Contributions to the Design of Asynchronous Macromodular Systems

Luis Angel Plana

Department of Computer Science

CUCS-001-99

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1998

ABSTRACT

Contributions to the Design of Asynchronous Macromodular Systems

Luis Angel Plana

In this thesis, I advocate the use of macromodules to design and build robust and performance-competitive asynchronous systems. The contributions of the work relate to different aspects of the design of asynchronous macromodular systems.

First, an architectural optimization for 4-phase systems is introduced. The goal of the optimization is to increase the performance of a system by increasing the level of concurrent activity in the sequencing of data processing stages. In particular, three new asynchronous sequencers are designed, which increase the throughput of the system. Existing asynchronous datapaths do not operate correctly at this increased level of concurrency: *data hazards* may result. Interlock mechanisms are introduced to insure correct operation. The technique can also be regarded as a low-power optimization: The increased throughput can be traded for a significant reduction in the power consumption of the entire system. SPICE simulation results show that the new sequencers allow roughly twice the throughput of non-concurrent sequencers. The simulations also show that, after voltage scaling, energy dissipation is reduced by a factor of 2.5.

Second, the use of pulses for efficient inter-module synchronization is introduced. The idea is complemented with the definition of a pulse-mode handshake protocol and the characterization of Pulse-Burst Operation (PBO), an important extension to traditional pulse-mode operation. Also, a basic set of macromodules, that efficiently implement control operations such as sequencing, selection, iteration, concurrency control, resource sharing, and arbitration is presented. Modules for interfacing pulse-mode circuits with traditional 2-phase and 4-phase circuits are also included in the set.

Finally, the design of a packet switch is used to demonstrate the viability of pulse-mode macromodules to implement complex, high performance systems. The switch organization, its asynchronous operation, and the low control overhead introduced by pulse-mode macromodules result in a design that can handle 2.4 times the target throughput of 155 Mbits/Sec. Also, the switch is characterized by very low input-to-output latency. These results suggest that pulse-mode macromodules can keep control overhead low without introducing complex, unsafe timing considerations, two necessary conditions to achieve robust, performance-competitive systems.

Contents

List of Figures v

List of Tables ix

Acknowledgments x

Chapter 1 Introduction 1

1.1 Challenge: Robust, High-Performance Systems 3

1.2 Contributions of the Thesis 5

1.3 Organization of the Thesis 6

Chapter 2 Asynchronous Macromodular Systems 8

2.1 Introduction 9

2.2 Macromodular System Design 10

2.2.1 System Optimization 12

2.3 Macromodule Specification and Synthesis 14

2.4 Control Signaling 15

2.4.1 2-Phase Handshaking 16

2.4.2 4-Phase Handshaking 17

2.4.3 Single-Track Handshaking 17

2.4.4 Pulse-Mode Handshaking 18

2.5 Data Communication and Processing 18

2.5.1 Dual-Rail Datapaths 18

2.5.2 Single-rail Datapaths 21

2.6 High-Performance or Low-Power Systems? 24

Chapter 3 Architectural Optimization for 4-Phase Systems 27

3.1 Previous Work on Asynchronous Sequencers 28

3.1.1 Sequential Protocol 29

3.1.2 Concurrent Protocol 33

3.2 New Concurrent Sequencers 36

3.2.1 Tightly-Coupled Sequencer 36

3.2.2 Loosely-Coupled Sequencer 40

3.2.3 Speed-Independent Sequencer 42

3.3 Data Hazards in Concurrent Operation 44

3.4 Data Hazard Elimination 47

3.4.1 Algorithm Modification 47

3.4.2 Architectural Solution 48

3.4.3 WAW Hazard Elimination 49

3.4.4 WAR Hazard Elimination in Dual-Rail Datapaths 49

3.4.5 WAR Hazard Elimination in Single-Rail Datapaths 52

3.4.6 Comparison with Kagotani/Nanya Approach 54

3.5 Comparison with True Four-Phase Operation 55

3.6 Results 58

3.6.1 Sequencer Comparison 58

3.6.2 Simulation: Performance and Energy Consumption 59

3.6.3 Comparison with True Four-Phase Operation 63

3.7 Conclusions 64

Chapter 4 Pulse-mode Macromodular Systems 65

4.1 Pulse-Mode Handshaking 66

4.1.1 Traditional Pulse-Mode Operation 67

4.1.2 Pulse-Burst Operation 68

4.1.3 Pulse-Mode Implementation Issues 69

4.2 Pulse-mode Control Macromodules 72

4.2.1 Sequencing 72

4.2.2 Selection 73

4.2.3 Conditional Iteration 77

4.2.4 Unconditional Iteration 79

4.2.5 Pulse-Mode Counters 80

4.2.6 Resource Sharing 83

4.2.7 Concurrency Control 84

4.3 Protocol Conversion 86

4.3.1 Transition Signaling/Pulse-mode Conversion 86

4.3.2 4-Phase/Pulse-mode Conversion 87

4.4 Arbitration 89

4.4.1 Background 89

4.4.2 4-Phase Arbiters 90

4.4.3 Event-Based Arbiters 91

4.4.4 Pulse-Mode Arbiter 92

4.5 Pulse-Mode Control of Datapaths 92

4.5.1 Single-Rail Datapaths 93

4.5.2 Dual-Rail Datapaths 94

4.5.3 Datapaths Implemented Using Pulse-Driven Logic 96

4.6 System Optimization 98

4.6.1 JOIN elimination 99

4.6.2 Module Substitution 100

4.7 Pulse-Mode Systems: Examples 101

4.7.1 Micropipelines 101

4.7.2 Iterative Multiplier 102

4.8 Conclusions 105

Chapter 5 Design of a Macromodular Packet Switch 108

5.1 Introduction 109

5.2 Switch Architecture 111

5.3 Switch Design 115

5.3.1 Asynchronous Bus (BUS) 118

5.3.2 Input Processor (IP) 122

5.3.3 Output Processor (OP) 130

5.3.4 FIFO Buffers 135

5.4 Switch Simulation 142

5.4.1 Operation of the Macromodular Switch 142

5.4.2 Characteristics of the Macromodular Switch 144

5.4.3 Comparison with a Synchronous System 146

5.5 Conclusions 147

Chapter 6 Conclusions 149

List of Figures

2.1 Macromodule interconnections: Channels and Ports. 9
2.2 Macromodular System Diagram. 10
2.3 Macromodular System High-Level Specification. 11
2.4 Optimized Macromodular System. 13
2.5 Re-synthesized Macromodular System. 14
2.6 Example of Macromodule and specification. 15
2.7 Handshaking Protocols for Module Communication. 16
2.8 Dual-Rail Latch. 19
2.9 Block Diagram of Dual-Rail Datapath Stage. 20
2.10 Block Diagram of Single-Rail Datapath Stage. 22

3.1 4-Way Sequencer: Sequential Protocol. 29
3.2 Tangram SEQ: (a) Symbol, (b) Circuit, (c) Tree Sequencer. 30
3.3 Josephs/Bailey Counter-Decoder Sequencer. 31
3.4 Josephs/Bailey Chain Sequencer. 32
3.5 4-Way Sequencer: Concurrent Protocol. 33
3.6 2-step Module, Circuit Implementation and Tree Sequencer. 34
3.7 Auto-Sweeping Module, Implementation and Chain Sequencer. . . . 35
3.8 Tightly-Coupled Sequencer Operation. 37
3.9 Tightly-Coupled Sequencer Burst-Mode Specification. 37

3.10 Tightly-Coupled Sequencer: (a) Block Diagram, (b) Modules. . . . 38
3.11 Tightly-Coupled Modules Controlling Processes. 39
3.12 Operation of the Loosely-Coupled Sequencer. 40
3.13 Loosely-Coupled Sequencer: (a) Block Diagram, (b) Modules. . . . 41
3.14 Loosely-Coupled Sequencer: Two Intermediate Stages. 42
3.15 Operation of the Speed-Independent Sequencer. 42
3.16 Speed-Independent Sequencer: (a) Block Diagram, (b) Modules. . . 43
3.17 Sequencer controlling 2 Dual-Rail Datapath Processes. 45
3.18 Scenario for a WAW Data Hazard. 46
3.19 Scenario for a WAR Data Hazard. 46
3.20 Macromodular System High-Level Specification. 47
3.21 Macromodular System High-Level Specification. 48
3.22 Interlock Circuit to Avoid WAW Hazard. 49
3.23 Interlock Circuit to Avoid WAR Hazard in Dual-Rail Datapaths. . . 50
3.24 Modified Dual-Rail Latch Gate-Level Implementation. 51
3.25 Modified Dual-Rail Latch Transistor-Level Implementation. 52
3.26 Robust Interlock to Avoid Single-Rail WAR Hazards. 53
3.27 Avoiding Single-Rail WAR Hazards: Fast Interlock. 53
3.28 Kagotani/Nanya Interlock Mechanism for Dual-Rail Datapaths. . . 54
3.29 Single-Rail Operation: Sequential, TFP and Concurrent. 56
3.30 Performance and Power Consumption of Dual-Rail System. 61
3.31 Power Consumption of Single-Rail System (High-Latency). 62
3.32 Power Consumption of Single-Rail System (Low-Latency). 62
3.33 Power Consumption of Multiplier/Adder Single-Rail Datapath. . . . 63

4.1 Pulse-Mode Handshaking for Module Communication. 67
4.2 Example of Pulse-Burst Operation: JOIN Module specification. . . 68
4.3 Pulse-Mode PULSER Module: Block Diagram and I-Net. 71

4.4 Single-Edge Implementation of the PULSER. 71

4.5 Pulse-Mode 2-STEP Module: Block Diagram and I-Net. 73

4.6 Pulse-Mode IF-ELSE Module: Block Diagram and I-Net. 74

4.7 Pulse-Mode IF-ELSE Module Implementation. 75

4.8 Pulse-Mode IF-ELSE Module (Single-Rail Variable X). 75

4.9 Transition Signaling and 4-Phase IF-ELSE Modules. 76

4.10 Pulse-Mode WHILE Module: Block Diagram and I-Net. 78

4.11 Pulse-Mode UNTIL Module: Block Diagram and I-Net. 79

4.12 Pulse-Mode DO_N Module. 80

4.13 Pulse-Mode TOGGLE Element. 81

4.14 Pulse-Mode 8-COUNTER using TOGGLE Elements. 82

4.15 Alternative Pulse-Mode 8-COUNTER Implementation. 82

4.16 Pulse-Mode CALL Module: Block Diagram and I-Net. 83

4.17 Pulse-Mode FORK/JOIN Module. 84

4.18 Pulse-Mode JOIN Element Specification. 85

4.19 JOIN Element Implementation. 85

4.20 TS/PM Module: Block Diagram and I-Net. 86

4.21 PM/4Φ Module: Block Diagram and I-Net. 88

4.22 Mutual Exclusion Element. 90

4.23 Arbiter Element and Tree Arbiter. 91

4.24 RGD Arbiter: Pulse-Mode Interface. 92

4.25 Pulse-Mode Control of Single-Rail Datapath. 93

4.26 Pulse-Mode Interface to Dual-Rail Datapath. 95

4.27 Pulse-Driven Dual-Rail Datapath. 97

4.28 Optimization: JOIN Elimination. 99

4.29 Optimization: Module Substitution. 101

4.30 Pulse-Mode Micropipeline. 102

4.31 Iterative Multiplier Specification. 103

4.32 Add-and-Shift Multiplier Datapath. 104

4.33 Add-and-shift Multiplier Control Logic. 104

4.34 Optimized Multiplier Control. 105

5.1 Switch Structure. 112

5.2 Switch Fabric Datapath. 113

5.3 Packet Switch Block Diagram. 115

5.4 Asynchronous Internal Bus. 120

5.5 Input Processor Module Interface. 123

5.6 Input Processor Specification. 123

5.7 Input Processor Macromodule Diagram. 125

5.8 *IP* Specification Segment: Two Concurrent Processes. 127

5.9 Segment of the *IP* Macromodular Diagram. 128

5.10 Optimized Input Processor Macromodule Diagram. 129

5.11 Fabric Input to Buffer Path (Data and Control). 130

5.12 Output Processor Module Interface. 131

5.13 Output Processor Specification. 132

5.14 Output Processor Macromodule Diagram. 133

5.15 Optimized Output Processor Macromodule Diagram. 134

5.16 Buffer to Fabric Output Path (Data and Control). 135

5.17 Buffer Module Interface. 136

5.18 FIFO Buffer: Micropipeline Implementation. 137

5.19 Low-Latency FIFO Buffer. 139

5.20 Low-Latency FIFO Buffer: Write Control. 140

5.21 Low-Latency FIFO Buffer: Read Control. 141

5.22 Functional Simulation: Packet Switching. 143

5.23 Simulation: (a) Input, and (b) Output Response Time. 144

List of Tables

3.1 Static Characteristics of N-way Sequencers. 59

3.2 Dynamic Behavior of N-way Sequencers. 60

5.1 Timing Parameters of Simulated Packet Switch. 145

5.2 Comparison with Fabricated Synchronous Switch. 147

Acknowledgments

I would like to start this list of acknowledgments with Steve Unger, my advisor. It has been a true privilege to work with him and to learn from his technical expertise, and also from his ethics and ideas about the social responsibility of engineers. He was a constant source of ideas, insights and thought-provoking comments. It was his encouragement that kept me “rolling” and helped me start and finish the dissertation. I hope I can transmit some of these experiences and learnings to my students in the future.

I’m also grateful to Steve Nowick, formally the “second reader” of the thesis, but in practice a “second advisor”. I had the opportunity to interact with him, not only while working in parts of the dissertation but in many other situations, and learn basic skills to become a researcher: to be thorough in finding the solution to a problem and careful and clear in writing it. I want to thank the other members of the thesis committee: Peter Allen, Charles Zukowski and Andrew Campbell, for the effort they put in judging this thesis, and their interesting comments and questions.

I must also thank the other members of the async group: Fu-Chiung John Cheng, Michael Theobald, Montek Singh, Robert Fuhrer and Tiberiu Chelcea. Our weekly seminar was a great opportunity to share ideas and to learn form each other. They also made life at Columbia a better one.

I’d like to mention my friends Paul Michelman, Bülent Yener, Tom O’Donnell

and Peter Karp. I didn't expect to make such good friends here. As Paul said, "I'm grateful that they were born and that we met".

I couldn't have done much at Columbia without the help and encouragement of Rosemary Addarich, Mel Francis, Martha Peres, Alice Cueba, and Ashutosh and Manu Dutta. They were always very nice and helpful. They showed me how to walk some difficult paths.

I would also like to thank some of my fellow PhD students who helped create a sense of community and with whom I enjoyed many interesting moments: Sushil Da Silva, Apostolos Dailianas, Andreas Prodromidis, Simon Baker, Kazi Zaman, Damianos Chatziantoniou and Akira Kawaguchi. The list is obviously incomplete, I hope that those I forgot will understand.

This work benefited from the support of Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICIT) and Universidad Nacional Experimental Politécnica Antonio José de Sucre (UNEXPO), both in Venezuela.

Agradecimiento

"Gracias a la Divina Pastora por favor concedido"

Tenemos la costumbre de dejar siempre lo más importante para el final y este agradecimiento no es la excepción. Quiero agradecer el apoyo constante y el cariño de la familia Plana Cabrera: Luis, Amparo, Amparito, Alicia, Carolina y Eduardo. Pocas personas han tenido la fortuna de contar con una familia como esta.

Ni este trabajo, ni los pasados 22 años de mi vida, hubieran sido posibles sin la compañía, el cariño y el apoyo de Carolina, Daniela y Santiago. Espero poder compensarlos por todo el tiempo que no les di para dedicárselo a los estudios y a este trabajo. Gracias.

Chapter 1

Introduction

Virtually all digital systems in use today are synchronous. The synchronous approach is based on the use of a single, global signal –the clock signal– to synchronize the operation of all system components. The basic design rule for a synchronous system is that all signals must be stable and valid when a clock pulse is produced.

This simple rule is one of the main reasons for the popularity of the synchronous approach but is also becoming one of its basic limitations, for several reasons: (*i*) The global clock signal must be distributed to every component of the system. As the clock frequency increases, the variation in the arrival time of the clock to different parts of the system –*clock skew*– becomes a problem. (*ii*) The circuit that distributes the clock signal is one of the major sources of power consumption in a system. Also, power is consumed in every clocked component of the system even in the case where no new information needs to be processed or stored. (*iii*) The clock frequency is designed so that every component stabilizes before a clock pulse arrives. This means, essentially, that the slowest components of a system have the largest impact on its performance, *i.e.,* synchronous systems tend to exhibit worst-case performance.

Many researchers are looking for solutions to these problems. Techniques to "synchronize" the arrival of the clock signal to different parts of the system have been devised and applied to reduce clock skew problems. Methods to save power by selectively shutting off the clock in idle sections of the system have also been implemented. Finding good solutions is not always easy. Sometimes, a solution to one problem creates another: introducing logic into the paths of the clock signal to save power increases the clock skew problem and, therefore, may incur a performance penalty.

These problems of the synchronous approach are increasingly more difficult to solve. Consequently, interest in asynchronous systems has grown considerably in recent years. In principle, an asynchronous approach is attractive for several reasons: (*i*) Asynchronous systems have no global clock, avoiding the clock skew and power consumption problems related to the distribution of this signal in synchronous systems. (*ii*) Asynchronous circuits have an inherent power-down operation: components are activated only when their operations are needed. (*iii*) Each system component operates at its own speed: if the components cooperate properly, systems may exhibit average-case performance. (*iv*) It is easier, in unclocked systems, to deal with metastability [44, 90], because there is no clock period imposing an upper bound on the length of time before a signal emerges from a metastable state.

Asynchronous design has been the focus of intense research activity and several design methods have been introduced recently [83, 45, 6, 64, 106, 63, 94, 29]. These methods have been used to design large scale systems that have worked correctly and exhibited good performance. Important examples are a zero-overhead divider [100], a DCC error corrector [92], an asynchronous differential equation solver [107], and the different versions of the Amulet processor [29].

Some of these design methods build asynchronous circuits as networks of cooperating modules. Such systems are called *macromodular* [14, 68, 82], since they are constructed by combining modules into a working system. Easily interconnected modules of small to moderate complexity are designed and optimized individually and used as basic building blocks.

In the context of complex VLSI systems, which have reached a point where design time and cost usually exceed fabrication time and cost, this modular approach has several potential advantages over *monolithic* systems: (*i*) Individual modules can be optimized separately, without the rest of the system requiring any changes. As a result, any performance improvement of a module can improve the performance of the system. (*ii*) Low-level engineering details are separated from the algorithm and organization levels of design. A designer can concentrate on the algorithmic aspects and can try different organizations at the macromodule level. Such an approach also allows the estimation of speed, area and power at the macromodule level. (*iii*) After the system is designed at the macromodule level, optimization techniques can be used to reduce the overhead introduced by the use of pre-designed modules.

1.1 Challenge: Robust, High-Performance Systems

After extensive experimentation to optimize the performance of a basic elastic FIFO structure [56], Molnar *et al.* conclude the following:

> “These results encourage us to believe that performance- and area-competitive asynchronous circuits can be achieved if one is willing to rely on control of delays in circuits and their interconnections to the same degree as in clocked systems.”

Their conclusion addresses one of the key issues that asynchronous circuit designers face: the trade-off between robustness and performance. Asynchronous circuits can be classified according to their choice in that trade-off.

Delay-Insensitive (DI) circuits [55] are the most robust. They operate correctly regardless of gate and wire delays. Unfortunately, this robustness has its price. Martin [48] and Brzozowski *et al.* [7] have shown that the class of DI circuits, implemented using simple gates, is very limited, and the performance of these system is often limited also.

Speed Independent (SI) circuits [52] assume unbounded, finite gate delays but zero delay in the wires. Quasi-delay-insensitive (QDI) circuits [45] assume unbounded gate and wire delays but introduce a timing assumption: certain forks are considered isochronic, *i.e.,* there is no difference in the propagation time of the branches. Many consider that these two classes of circuits are equivalent.

Unfortunately, the assumption of zero wire delay is less realistic as feature sizes decrease and gate delays become comparable with delays in the wires. Isochronic forks are not an easy assumption to comply with, as shown by van Berkel [95]. Even if it is met, the performance of the circuits is not always adequate. The need for better performance has led to the definition of extended isochronic forks [98], in which the equal delay assumption is applied to one or more levels of inverting CMOS gates connected to the branches of the fork.

Self-timed (ST) circuits [79] consist of networks of self-timed elements that are interconnected using delay-insensitive connections. The self-timed elements can be designed according to any desired delay model. An important feature of ST circuits is that any timing assumptions must relate local signals, since they must belong to the same module.

General asynchronous circuits (AC) assume bounded gate and wire delays. This is the largest class of circuits but also the least robust. Making timing assumptions about relative delays of gates or wires that are far from each other, or are fabricated separately may be risky.

Most definitely, high performance is the result of tight control overhead and use of timing assumptions. Robustness is the result of safe, conservative and verifiable timing assumptions. Clearly, a designer faces a delicate balancing act. The self-timed strategy of detailed local control of internal timing and delays in the modules combined with delay-insensitive external interfaces seems to be a step in the right direction. We are challenged to push it to the limits.

1.2 Contributions of the Thesis

The goal of the work described in this thesis is to contribute to the design of robust, performance-competitive asynchronous systems. Two different classes of asynchronous systems (defined later) are targeted: 4-phase and pulse-mode systems.

In 4-phase systems, the work consists of the design of efficient circuitry to "eliminate" one of the largest sources of control overhead in this type of systems, the return-to-zero phase. The main contributions of the work are the following:

- New designs for sequencing control (or "sequencers"), which greatly increase the performance of non-pipelined asynchronous systems. The sequencers overlap the redundant phase with the execution of productive work that otherwise would be delayed. The new sequencers have advantages over all existing sequence control elements.

- Interlock circuitry that guarantees that the system can operate correctly at the increased performance.

In pulse-mode systems, the main contributions of the work presented in this thesis are the following:

- The introduction of the use of pulses as an efficient handshaking protocol. Although the idea had been suggested previously, no systematic approach to the use of pulse-mode inter-module synchronization has been presented before.

- The introduction of a more concurrent form of pulse-mode operation. Traditional pulse-mode systems allow a single pulse to be active at any time. Pulse burst operation (PBO), introduced in this work, allows concurrent pulses as inputs to a module.
- The design of a large set of pulse-mode macromodules that can be used to build cost-effective asynchronous macromodular systems. Such a set has not been proposed before. The set includes control macromodules as well as arbiters and converter modules to interface pulse-mode to other handshaking protocols.
- The viability of the use of pulse-mode macromodules in the construction of a large, complex, high-performance system is demonstrated with the design of an asynchronous packet switch. A packet switch was chosen as a case study because it is a control-dominated system in which control overhead has a large impact on performance, stressing the need for an efficient design.

1.3 Organization of the Thesis

This chapter has presented the context in which the work reported in the thesis is inscribed. The rest of the thesis is organized as follows:

Chapter 2 reviews basic background and previous work on asynchronous macromodular systems. In particular, the design and optimization process of macromodular systems is examined, and the basic handshaking protocols and data encodings are presented. Given that high performance and low power are two important goals in the design of macromodular systems, this chapter also reviews an important technique for the reduction of power dissipation and shows how performance and power consumption can be traded to obtain the desired goals.

Chapter 3 focuses on the class of macromodular systems that use 4-phase handshaking. This chapter presents an architectural optimization that increases the performance of a system by increasing the level of concurrency in the sequencing of data processing stages. In this chapter, three new asynchronous concurrent sequencers are introduced, that increase the concurrent activity and throughput of the entire system. The chapter also discusses the impact of the new sequencers on the operation of existing datapaths and introduces modifications to safely handle the increased concurrent operation.

Chapter 4 introduces the use of pulses as handshake events in macromodular systems and defines a pulse-mode handshake protocol. Designs of pulse-mode modules that implement control operations such as sequencing, selection, iteration, concurrency control, resource sharing and arbitration are introduced and their characteristics compared to those of equivalent transition signaling and 4-phase implementations. Modules for interfacing pulse-mode circuits with traditional 2-phase and 4-phase circuits are also presented. The chapter also illustrates examples of the use of these pulse-mode modules in the implementation of macromodular systems and micropipelines.

In Chapter 5, the design of an asynchronous packet switch is presented, to demonstrate the feasibility of the use of pulse-mode handshaking to build large, complex systems. The packet switch is implemented as a pulse-mode macromodular system. The details of the design process as well as topics like the use of buses in asynchronous systems, arbitration, and optimizations based on timing assumptions are addressed in this chapter.

Finally, Chapter 6 summarizes the contributions of the work and presents conclusions.

Chapter 2

Asynchronous Macromodular Systems

In this chapter, background and previous work on macromodular systems is reviewed. In particular, the design and optimization process of macromodular systems is examined, and the basic handshaking protocols and data encodings are presented.

Even though the main goal of the work presented in the following chapters is to improve the performance of macromodular systems, sometimes a reduction in the power consumption of the system is desired. Therefore we also review an important technique for reduction of power dissipation in CMOS circuits.

This chapter is organized as follows. Section 2.1 presents the operation of elementary macromodules. Section 2.2 introduces the basic design methods and a simple example of a macromodular system. Macromodule specification methods and design procedures are presented in Section 2.3. Sections 2.4 and 2.5 review the details of the operation of the controller and datapath in macromodular systems. Finally, Section 2.6 reviews basic concepts related to power consumption in CMOS circuits and shows how performance and energy can be traded to achieve desired design goals.

2.1 Introduction

Asynchronous macromodules were introduced by Clark, Ornstein and Stucki in [14, 68, 82] as a set of simple, easily interconnected modules from which working systems can be readily assembled. Keller [40], Molnar *et al.* [55], Rosenberger *et al.* [78], Brunvand [6], Unger [87], and van Berkel *et al.* [94] among others, have continued to work on macromodules and have made important contributions. Macromodular systems are sometimes known by other names, such as "Handshake Circuits" [94].

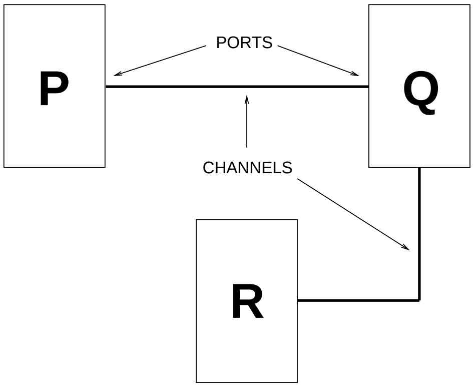


Figure 2.1: Macromodule interconnections: Channels and Ports.

Figure 2.1 illustrates a simple macromodular system, which consists of three modules: P, Q and R. Modules communicate with each other through channels. Channels consist of one or more wires that conduct data and control signals. Instead of a global clock signal, channels between modules use handshaking to synchronize their operation and data interchange. Usually, there are no timing assumptions related to the operation of the channels.

The channels attach to module ports, which provide a standard interface to the module. A module can have a single port (P and R) or several, as module Q in the figure. In some cases, Q must be able to interact with P and R concurrently.

If all the modules behave properly, *i.e.,* comply with the handshaking requirements of the channels, the system will operate correctly, independently of the speed of the modules. An important advantage of the system is its modularity. If Q is substituted by Q', which has the same functionality but faster response time, the system must still operate correctly and, in some cases, increase its performance due to the faster new module.

2.2 Macromodular System Design

A small set of macromodules is now used to illustrate the operation of a macromodular system and the basic steps in the design process.

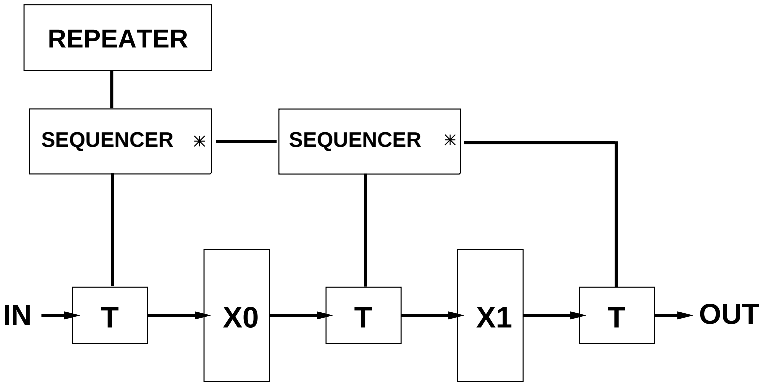


Figure 2.2: Macromodular System Diagram.

Figure 2.2 illustrates a simple macromodular system. The system represents a two-place ripple shift register. The system is built by connecting modules together. IN is the input to the system and OUT is its only output. $\boxed{\text{X0}}$ and $\boxed{\text{X1}}$

are the two registers that store data. The $\boxed{\text{T}}$ modules are transferrers, used to transfer (copy) information from their inputs to their outputs. A $\boxed{\text{SEQUENCER}}$ is used to activate two modules in a prescribed sequence (or order): the module connected to the $*$ output is activated first and, when this module completes its operation, the other module is activated. Finally, the $\boxed{\text{REPEATER}}$, as it names implies, is used to repeatedly activate a module. As indicated above, channels between modules use handshaking to synchronize their operation and data interchange.

Succinctly, the operation of the system consists of a sequence of three actions, repeated permanently: (*i*) Transfer the contents of X_1 to the output, (*ii*) transfer the contents of X_0 to X_1, and (*iii*) transfer the new input to X_0.

The system operates as follows. The REPEATER initiates the actions by activating the SEQUENCER on the left. This module activates the SEQUENCER on the right. The second SEQUENCER finally interacts with the datapath: it activates the rightmost transferrer, which copies data from X1 to *OUT*, completing action (*i*) above. The sequencer then activates the middle transferrer to copy data from X0 to X1. Finally, the first SEQUENCER activates the left transferrer to copy data from *IN* to X0, completing the sequence of actions. When the sequencer signals completion to the REPEATER, this module starts the process again.

```
begin
  REPEAT forever
    OUT = X1;
    X1 = X0;
    X0 = IN
end
```

Figure 2.3: Macromodular System High-Level Specification.

Macromodules are particularly well-suited for methods that approach circuit design as a programming activity. For example, Brunvand [6] and van Berkel *et al.* [94], have developed methods to automatically design asynchronous circuits from high-level programs. One possible high-level description of the two-place shift register above is shown in Figure 2.3. The programs are compiled, using syntax-directed translation, into a macromodular system, as an intermediate level representation. To obtain a gate or transistor-level circuit, the modules must be substituted by their pre-designed implementations. It is interesting to note that a macromodular system can be mapped into very different circuits, depending on the actual modules used and the handshake and data encoding options chosen.

This mapping process, and the lack of global synchronization, are the key elements that allow incremental improvement of a complete macromodular systems by improving each module separately. The macromodules are designed using different techniques, sometimes by hand or using automatic tools. Clearly, module design has a large impact on the performance of macromodular systems.

2.2.1 System Optimization

A potential weakness of the use of macromodules is the presence of inefficiencies in the design due to the use of pre-designed modules. It is possible to examine the macromodule diagram and optimize it in a number of ways.

This optimization process, sometimes called peephole optimization, has been studied by several researches. For example, Brunvand [6], van Berkel [96], and Peeters [71] have introduced optimizations that consist of structural transformations that maintain the functionality of the circuit but optimize it according to a desired metric. These optimizations are very similar to the peephole optimizations used by compilers to optimize code.

Figure 2.4 shows an example of such optimization. the circuit for the two-

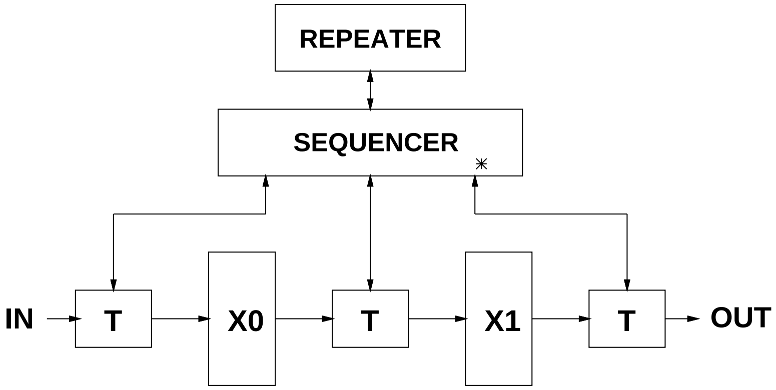


Figure 2.4: Optimized Macromodular System.

place ripple shift register above has been optimized: the two 2-way sequencers have been substituted by a single 3-way sequencer. The larger sequencer uses less area and has better performance than the combination of the two smaller ones (see Chapter 3 for a thorough design of new n-way sequencers).

A different approach is taken by Gopalakrishnan *et al.* [33] and Kolks *et al.* [41]. They optimize the macromodular system by re-synthesizing sections of the control path (possibly comprising several macromodules), using automatic tools. Figure 2.5 shows how this optimization may be applied to the ripple register: the specifications of the repeater and sequencers are combined and fed to an automatic synthesis tool. The tool generates a finite state machine (FSM) that substitutes the modules in the control of the register. The resulting macromodular circuits are very robust and usually have few timing assumptions.

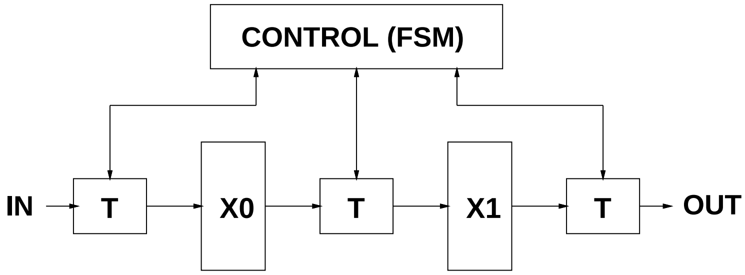


Figure 2.5: Re-synthesized Macromodular System.

2.3 Macromodule Specification and Synthesis

The previous section showed that macromodular systems are designed as compositions of pre-designed macromodules. An important question is: how are the macromodules themselves specified and designed? This section presents basic concepts to answer the question.

Several specification forms have been used to express the desired behavior of a module: Flow tables [87], state diagrams, burst-mode specifications [63, 106], CSP or similar language based forms [6, 45, 96], and Petri net based forms [6, 55]. In our case, flow tables and a restricted form of Petri nets, called Interface- or I-NETS, will be used. I-NETS will be used to represent the desired Input/Output behavior of a module and flow tables will be used to provide detailed specifications of module operation.

I-NETS were introduced by Molnar *et al.* [55] as a tool to specify the interface behavior of a module. Figure 2.6 shows an example of a simple I-NET, which corresponds to a pulse-mode JOIN Module, that will be described in detail in Chapter 4.

I-NETS consist of places (circles), transitions (bars), and arcs that connect places and transitions. Transitions are used to represent input/output (or inter-

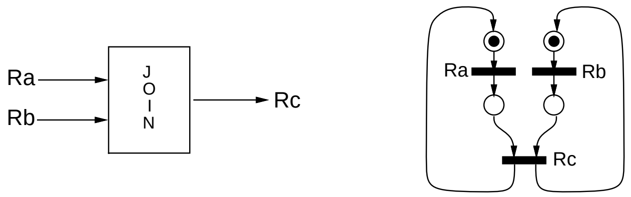


Figure 2.6: Example of Macromodule and specification.

face) events that take place in the module. In the case of the JOIN module, each transition corresponds to a pulse in an input or output signal of the module. Only *enabled* transitions can occur. A transition is enabled when all places that point to that transition contain a *token*.

Modules can be synthesized using several different procedures. Automatic synthesis tools can be used. Some of the tools are finite-state machine (FSM) synthesis tools, others are based on Petri net-like specifications and others are based on high-level programming languages.

In our case, the modules are of an order of complexity that is very well suited to design by flow-table based techniques. In many cases, hand synthesis results in very efficient implementations. Timing problems related to critical races, and combinational and essential hazards can be dealt with very effectively.

2.4 Control Signaling

In Figure 2.4 above, it is easy to identify two sections of the system: Datapath and Control. This is a helpful distinction due to the different nature of the data processing stages and the control sections. This difference will become clear in this and the following sections, in which details of handshaking and data communication are reviewed.

Figure 2.7 shows two macromodules that communicate with each other using *request* and *acknowledge* signals. Initially, the modules are idle and both signals are de-asserted. P uses signal r to request Q to start processing. Q uses signal a to indicate completion. Module P, that starts the handshake, is called the active component, and Q is the passive one.

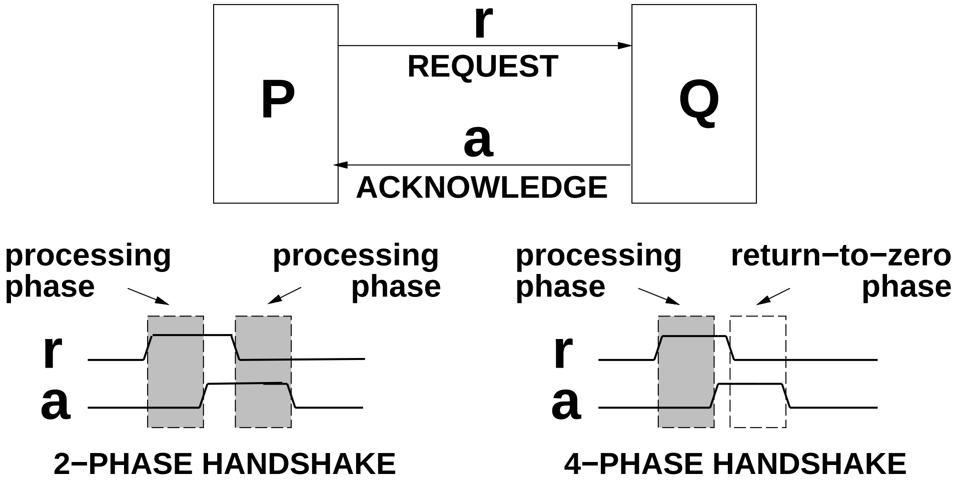


Figure 2.7: Handshaking Protocols for Module Communication.

Control signaling usually follows a *handshake protocol*. *2-phase* and *4-phase handshaking* are the two most commonly used protocols [79]. Other handshaking protocols are *single-track* and *pulse-mode*. These protocols are reviewed in the following sections.

2.4.1 2-Phase Handshaking

This is a simple protocol. Usually, signal transitions are used to represent events in 2-phase handshaking. A complete handshake consists of two events: a request (r), that starts the handshake, and an acknowledge (a), that completes it. For example, a first handshake consists of $r\uparrow; a\uparrow$. The following handshake will consist of $r\downarrow; a\downarrow$.

2-phase handshaking is commonly used. Some examples are the macromodules project [14], micropipelines [83], and OCCAM-based systems [6]. A disadvantage of transition signaling is that the levels of the wires after a complete handshake are different from the levels before that handshake, usually leading to more complex circuits than level signaling [97]. Also, many datapath components require level-sensitive control, so transition-to-level (or 2-phase to 4-phase) converters must be used.

2.4.2 4-Phase Handshaking

4-phase handshaking is level-based. This protocol forces the wires to return to their initial level at the end of the handshake. A complete 4-phase handshake consists of four events: $r\uparrow$, that starts the handshake; $a\uparrow$, that indicates the end of the processing phase; $r\downarrow$ starts the *return-to-zero phase* and $a\downarrow$ completes the handshake.

4-phase handshaking is also very common. The AMULET asynchronous microprocessor [29] and Tangram-based systems [94] use a 4-phase protocol and have successfully demonstrated the low-power potential of asynchronous systems. A disadvantage of 4-phase handshaking is the presence of the redundant return-to-zero phase, which usually degrades performance [1].

2.4.3 Single-Track Handshaking

Recently, van Berkel and Bink introduced *single-track handshaking* [97]. This 2-phase protocol uses a single wire (w) for both request and acknowledge, with the communicating processes alternating control over the wire. In general, the starting process initiates the handshake by $w\uparrow$ and then *releases* the wire to allow the other process to control it. This process completes the handshake by generating $w\downarrow$ and

[1] In some cases, novel data-processing schemes [71] and concurrent protocols [74] may reduce the effects of the unproductive return-to-zero phase.

releasing the wire.

Single-track handshaking has the advantage that after each handshake the wire is in its initial state, which solves one of the problems of the 2-wire protocol. Unfortunately, the *shared* control over the single wire may create problems [97]: *(i)* circuits may be less reliable, *(ii)* complex, non-standard, gates that can gain and release control of the wire are required, and *(iii)* circuits may be difficult to test.

2.4.4 Pulse-Mode Handshaking

An alternative approach, mentioned by Keller [40] but not fully explored so far, is the use of pulses, instead of transitions, to represent events in a 2-phase protocol. Chapter 4 introduces our contributions to the use of this handshaking protocol, including a more general form of pulse-mode operation and a large set of pulse-mode macromodules, some of which were originally presented in [75].

2.5 Data Communication and Processing

When data communication is involved, techniques must be used to represent and transmit data. Two schemes are most common: *dual-rail* and *single-rail*.

2.5.1 Dual-Rail Datapaths

Encoding Scheme

In dual-rail, data is encoded using two wires for each data bit [79]. Codes 01 and 10 represent "1" and "0" data values respectively, and code 00 represents the *spacer* or *idle* state. This is a robust and widely-used scheme, which guarantees correct operation with arbitrary delays in the circuit [92, 94, 45, 46, 62]. However, a disadvantage is that implementations typically require larger area than single-rail designs, due to the two-wire encoding scheme. Another disadvantage is that there are additional transitions per bit between consecutive data words, due to the need for the spacer state. As a result, there are performance and power penalties.

Datapath Operation

As an example, we now introduce a simple dual-rail datapath stage. We begin with a description of the operation of a typical dual-rail latch, which is the basic storage element. After that, the operation of the complete stage is presented.

A block diagram of a dual-rail latch is shown in Figure 2.8. The latch has separate read and write ports; only one port may be active at a time. Initially, all wires are low. A *read operation* begins by asserting the read request ($R_r \uparrow$). The latch responds with $R_0 \uparrow$ (if "0" is read) or $R_1 \uparrow$ (if "1" is read). Once data has been used, R_r is de-asserted, followed by $R_0 \downarrow$ or $R_1 \downarrow$. Similarly, a *write operation* begins by asserting either W_0 or W_1 as a write request, indicating the value to be stored. $W_a \uparrow$ acknowledges that data has been stored. The asserted request (W_0 or W_1) is de-asserted, then W_a is de-asserted.

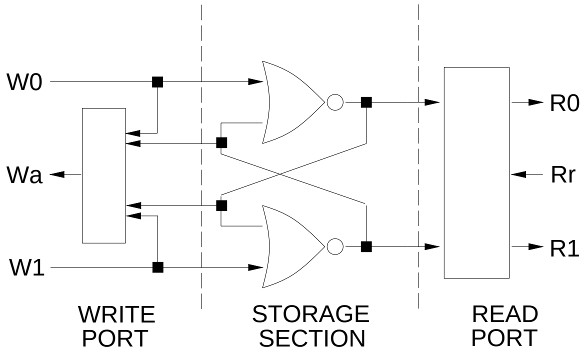


Figure 2.8: Dual-Rail Latch.

Figure 2.9 shows the interaction of datapath and control, in one stage of a dual-rail system. X, Y, and Z are registers built with dual-rail latches, and F is a combinational block which implements some function $F(X, Y)$ using dual-rail *hazard-free* logic (see, for example, [46, 62]).

Dual-Rail Datapaths/4-Phase handshaking

Usually, 4-phase handshaking is used to control dual-rail datapaths because the return-to-zero phase produces the necessary spacer or idle code.

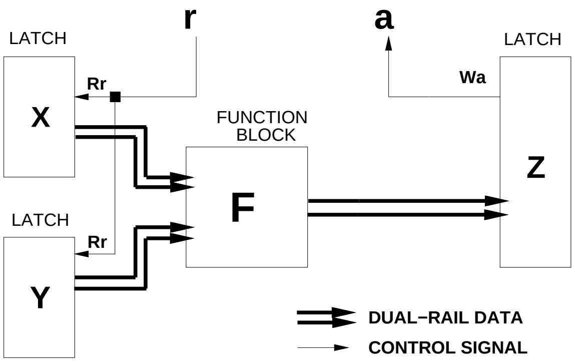


Figure 2.9: Block Diagram of Dual-Rail Datapath Stage.

The stage operates as follows. A request, $r\uparrow$, is sent to the stage, which initiates a *read request* of X and Y. The registers send dual-rail data to block F, which computes a dual-rail result. The result itself acts as a *write request* to register Z. Once data is written, the processing phase is complete, and Z sends acknowledge, $a\uparrow$, to the controller. The return-to-zero phase is initiated with $r\downarrow$. X and Y are then reset, the F outputs are de-asserted, and Z and a are reset, completing the operation.

Dual-Rail Datapaths/2-Phase handshaking

As mentioned above, dual-rail requires the presence of a spacer code between two consecutive valid data. This requires all data bits to return to their idle value, a phase that is not present in 2-phase handshaking. Thus, the use of dual-rail code with 2-phase handshaking is awkward, requiring special converter modules.

Dean *et al.* [18] introduced Level-Encoded 2-Phase Dual-Rail (LEDR) to address this problem. This is a dual-rail code that also uses 2 wires per data bit, but has no spacer or idle code.

2.5.2 Single-rail Datapaths

Encoding Scheme

A common alternative scheme is *single-rail*, which uses only one wire for each data bit, as in synchronous designs [83, 93, 6, 19, 72, 65]. An additional wire, called the data-valid signal, is used to indicate that the data in the wires is stable and valid. The set of all data wires together with the data-valid signal is called a *data bundle*. Correct operation of systems that use this encoding scheme relies on a local timing assumption: The data-valid signal must be asserted at the receiver end after all data signals are stable and valid. This is called a *bundling constraint*.

An advantage of single-rail encoding is that most existing synchronous (*i.e.*, non-hazard-free) function blocks can be used, so the datapath has good area characteristics. A disadvantage is that operation is less robust than dual-rail, due to the timing assumptions.

Datapath Operation

As an example, we now introduce a simple single-rail datapath stage. We begin with a description of the techniques most frequently used to comply with the bundling constraint. After that, the operation of the complete stage is presented.

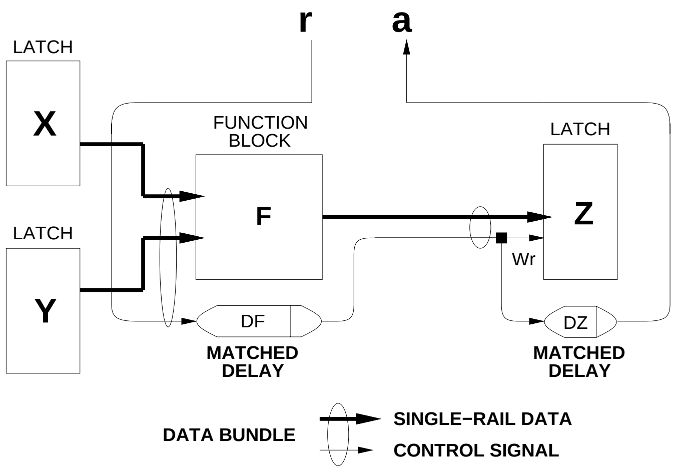


Figure 2.10: Block Diagram of Single-Rail Datapath Stage.

Figure 2.10 shows the interaction of datapath and control, in one stage of a system. To comply with the bundling constraint, *i.e.,* guarantee that all data wires are valid and stable before the data-valid signal is asserted, delays are inserted in the data-valid wires. These delays are designed to match the worst case delay of the corresponding datapath block; that is, DF must equal the worst case delay in the combinational logic block that implements function F, and DZ must be equal to the worst case delay in latch Z[2].

The operation of the single-rail stage in Figure 2.10 depends on the type of storage elements and the handshaking protocol used.

[2] If 4-phase handshaking is used to control single-rail datapaths, more novel data-processing schemes may avoid the unproductive return-to-zero phase. These schemes use different matching criteria for the delay elements (see [71] and Section 3.5 for details).

Single-Rail Datapaths/4-Phase handshaking

If a 4-phase protocol is used, standard D-latches (normally opaque) are typically used, with no special read port (data is always available at the outputs). In the case of the latches, the bundling constraint is similar to a set-up time requirement, insuring that data is valid and stable *before* the *latching*, or *data-valid*, signal W_r, is asserted.

In this case, the datapath operates as follows. The controller generates an initial request $r\uparrow$. Data from X and Y is already present as inputs to F. The request signal propagates through the matched delay DF while $F(X, Y)$ is computed. The output of the delay acts as the data-valid signal for the result. This data bundle is sent to Z. The arrival of the data-valid signal makes Z transparent and, after propagating through DZ, it is sent back to the controller as acknowledge signal a. At this point, the processing phase is complete and the controller starts the return-to-zero phase by de-asserting r, which propagates through DF and arrives at Z, making it opaque again. After propagation through DZ, a is de-asserted.

Single-Rail Datapaths/2-Phase handshaking

Alternatively, if 2-phase handshaking is used to control the datapath, event-based latches (also called capture-pass latches [83]) or dual-edge-triggered flip-flops (DETFFs) [89, 1, 105] can be used. These storage elements respond to transitions in the control signal, which make them more expensive than standard (level-based) latches. If D-latches are used in 2-phase systems, expensive 2-phase/4-phase converters are needed [83, 69, 16].

If DETFFs are used, the datapath operates as follows. The controller generates an initial request $r\uparrow$. Data from X and Y is already present as inputs to F. The transition in the request signal propagates through the matched delay DF while $F(X, Y)$ is computed. When computation is complete the data is stable and valid. The transition at the output of the delay acts as the data-valid signal for the result. The transition in the data-valid signal triggers the storage of the new data in Z and, after propagating through DZ, is sent back to the controller as acknowledge signal transition $a\uparrow$. At this point, the processing phase is complete. The next handshake will follow the same path with the negative transitions of the handshake signals.

Complying with the Bundling Constraint

The design of the matched delays is an important problem. A simple approach is to use an inverter chain as a rough matched delay. A better approach used in high-performance systems, where tight margins are critical, is to use a replicated portion of the critical path as the delay (see [19, 32, 65]). In CMOS implementations, delays depend heavily on the sizes of transistors and their loading, and also on the final routing and placement of modules, so safety margins are required for correct operation.

A potential weakness of the use of single-rail systems is that the bundled delays match the worst-case delay in the logic, eliminating the possibility of early completion of processes. To overcome this problem, Nowick *et al.* [65] introduced speculative completion. They use several bundled delays, of different lengths, and one of them, the one that fits the actual conditions of the circuit, is selected in every activation of the process.

2.6 High-Performance or Low-Power Systems?

Interest in low-power systems has grown considerably in recent years. This section shows how a performance increase can be traded for lower system energy consumption. The section reviews basic background on power consumption in CMOS circuits and introduces voltage scaling, an important power saving technique.

The constant increase in the use of battery-operated portable devices like cellular phones and notebook computers has made low power consumption a high priority. Low power is also becoming critical for non-portable systems because of its impact on packaging and component lifetime.

What can be done to reduce the power consumption of a system? To answer this question, we must understand how power is consumed in a circuit. There are three major sources of energy consumption in CMOS circuits. *Switching energy* is associated with transitions on gate outputs. *Short-circuit energy* consumption is caused by simultaneous conduction, during a transition, of pull-up and pull-down stacks, allowing current flow directly from the power supply to ground. Finally, *leakage energy* occurs in standby mode, and is caused by substrate currents and by sub-threshold conduction in off transistors. We only consider transition energy because in most CMOS circuits this energy dominates the other two and contributes up to 90% of the total energy [11].

In asynchronous systems there is no global clock, so the metric of interest is the *total energy of a computation.* A well-known expression [94] for this energy is:

$$\text{Energy of a computation} = \tfrac{1}{2} \cdot n \cdot C_L \cdot V_{dd}^2$$

where n is the total number of transitions in the computation, C_L is the load capacitance being charged/discharged, and V_{dd} is the power supply voltage.

A wide range of techniques is used to reduce circuit power consumption in CMOS circuits. Clearly, energy dissipation can be reduced by reducing the load capacitance, the number of transitions, or the supply voltage. Since energy depends quadratically on the supply voltage, *voltage scaling*, *i.e.,* the reduction of the supply voltage, is an especially attractive scheme for power reduction [11].

Unfortunately, voltage scaling has the undesirable effect of reducing the speed of the circuit. Several techniques are used to compensate for this performance penalty. In particular, Chandrakasan *et al.* [11, 10] propose *architecture-driven voltage scaling*, which combines architectural optimizations (to increase the throughput of the system) with voltage scaling (to reduce the power consumption). If the increase in performance is achieved without increasing the switching activity required for the computation, a substantial reduction in power is possible through voltage scaling, with no net loss in performance.

As a further optimization, Nielsen *et al.* [61] and van Berkel *et al.* [92] have shown how adaptive voltage scaling, *i.e.,* dynamic adjustments of the power supply voltage, can be used to reduce energy consumption and, at the same time, meet varying performance requirements.

We can now answer the question that starts this section. In CMOS circuits, optimizations that increase the performance of a system can also be regarded as power-saving techniques. The increased performance can be traded for lower energy consumption using voltage scaling.

Chapter 3

Architectural Optimization for 4-Phase Systems

In macromodular systems, the most basic control operation is the *sequencing* of computations or data processing actions. This chapter presents a new architectural optimization for 4-phase asynchronous systems. The goal of the optimization is to increase the performance of a system by increasing the level of concurrency in the sequencing of data processing stages.

In particular, the following contributions are presented. First, we introduce three new designs for asynchronous sequencers. Each design increases the throughput of the entire system. Second, we show that existing asynchronous datapaths will not operate correctly at this increased level of concurrency: *data hazards* may result. We therefore modify the datapath to insure correct operation. Specifically, we introduce interlock mechanisms that safely handle concurrent operation in both dual-rail and single-rail datapaths.

The use of these new sequencers can also be regarded as an optimization for low power. 4-phase macromodules are frequently used to implement non-pipelined asynchronous systems, such as low-power DSP circuits with moderate performance requirements [62, 92]. The increased throughput obtained through more concurrent operation can be traded for lower energy consumption by a reduction of the power supply voltage, as discussed in Section 2.6.

SPICE simulation results show that, for a dual-rail datapath, the new sequencers allow roughly twice the throughput of non-concurrent sequencers. After voltage scaling, energy dissipation of the system is reduced by a factor of 2.5. Similar results are obtained for a single-rail system.

Only two recent approaches attempt to achieve similar benefits. Kagotani and Nanya [39] introduce concurrent operation in dual-rail systems and deal with the presence of data hazards. Peters and van Berkel [71], without introducing concurrency, modify the usual operation of single-rail systems to obtain similar performance improvements. We compare our method to theirs, and indicate the performance and power advantages of our approach.

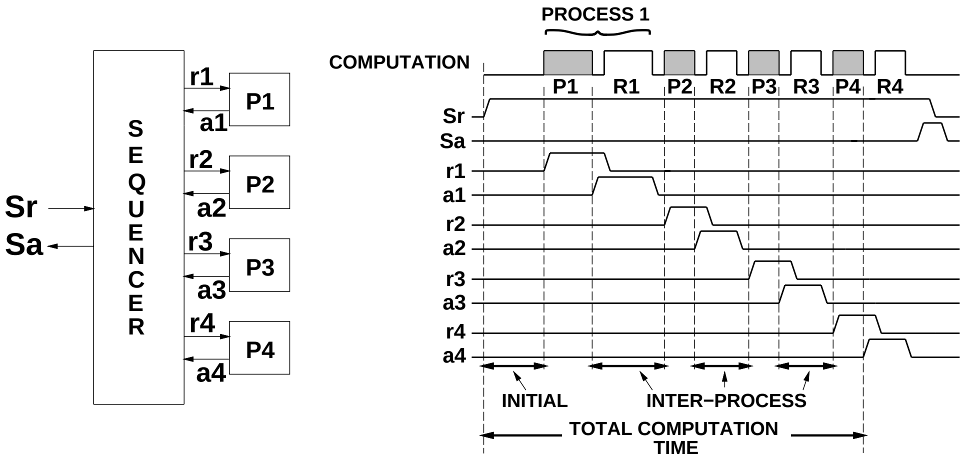
This chapter is organized as follows. Section 3.1 presents previous work on asynchronous sequencers, including a description of sequential and concurrent protocols. In Section 3.2, the three new low-latency sequencers are introduced, including details of their specification, design and operation. Data hazards in concurrent operation are analyzed in Section 3.3, and techniques to eliminate them in both dual-rail and single-rail datapaths are introduced in Section 3.4. Section 3.4.6 includes a detailed comparison with the dual-rail method of Kagotani and Nanya. Section 3.5 presents a thorough comparison with the single-rail approach introduced by Peeters and van Berkel. Section 3.6 presents results of analysis and SPICE simulations, and Section 3.7 presents conclusions.

3.1 Previous Work on Asynchronous Sequencers

In non-pipelined macromodular systems, the most basic operation is the *sequencing* of computations or data processing actions. Such sequences can be very long. For example, Bailey [4] reports that the longest sequence in the asynchronous error decoder circuit for a DCC player [92] consists of 48 processes. Two common protocols have been used in asynchronous sequencers: *sequential* and *concurrent*.

3.1.1 Sequential Protocol

The most common scheme is a *sequential protocol*. Figure 3.1(a) shows a sequencer controlling four processes: P_1, P_2, P_3, and P_4. In a sequential protocol, shown in Figure 3.1(b), the sequencer executes a complete 4-phase handshake with process P_i *before* starting the handshake with P_{i+1}. In this case, processing (P_i) and return-to-zero (R_i) phases alternate, resulting in a long dead time between computations, as shown in Figure 3.1(b). This non-computation time is called the *inter-process latency*. Other parameters, shown in Figure 3.1(b), are *initial latency* and *total computation time*.



(a) (b)

Figure 3.1: 4-Way Sequencer: Sequential Protocol.

A number of previous sequencer designs use a sequential protocol. A detailed quantitative comparison is presented in Section 3.6.

Tangram Sequencer

In Tangram, 2-way sequencing is implemented using the *SEQ* operator [94], shown in Figure 3.2(a). The sequencer is activated on its *passive* port, or channel, S (a passive port is indicated by a small white circle). The sequencer then communicates on *active* ports $P1$ and $P2$ to activate the first and second processes, respectively (an active port is indicated by a small black circle).

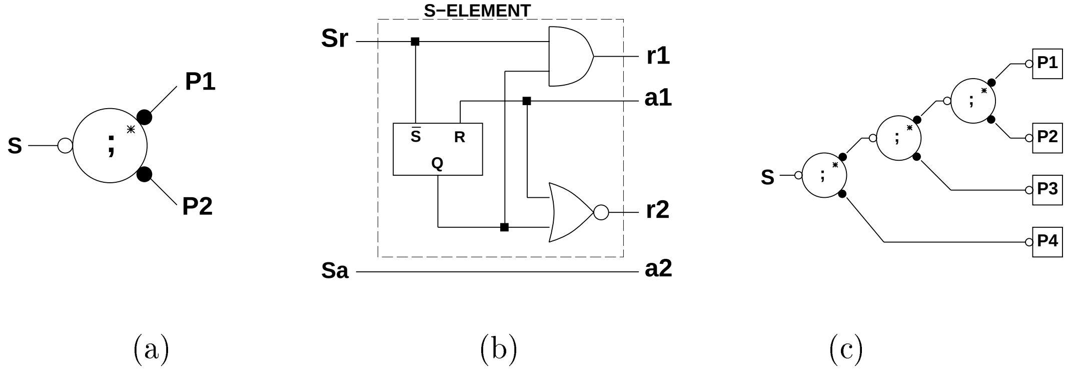


Figure 3.2: Tangram SEQ: (a) Symbol, (b) Circuit, (c) Tree Sequencer.

Channels are implemented using request and acknowledge wires (S_r and S_a for channel S, and r_i and a_i for channel Pi). A complete 4-phase handshaking occurs on port $P1$, followed by a complete 4-phase handshaking on port $P2$. The behavior of the SEQ operator can be described by the following expression:

$$*(s_r\uparrow; r_1\uparrow; a_1\uparrow; r_1\downarrow; a_1\downarrow; r_2\uparrow; a_2\uparrow; s_a\uparrow; s_r\downarrow; r_2\downarrow; a_2\downarrow; s_a\downarrow)$$

An implementation of the SEQ operator is shown in Figure 3.2(b). This circuit is *speed-independent* [52], *i.e.,* it operates correctly assuming arbitrary, finite, gate delays. An N-way sequencer consists of SEQ operators connected in a tree structure, as shown in Figure 3.2(c). There are two problems with the Tangram sequencer: (*i*) it has a long initial latency and (*ii*) it has long inter-process latencies.

Martin Sequencers

In [45], Martin presents two implementations of n-way sequencers. The first uses *Q-elements* and the other uses *D-elements*. A left-branching Tangram n-way sequencer corresponds exactly to a Q-element-based Martin n-way sequencer and it has the same performance problems discussed earlier. A D-element based sequencer reduces the initial latency but provides no overall performance improvement.

Josephs/Bailey Counter–Decoder Sequencer

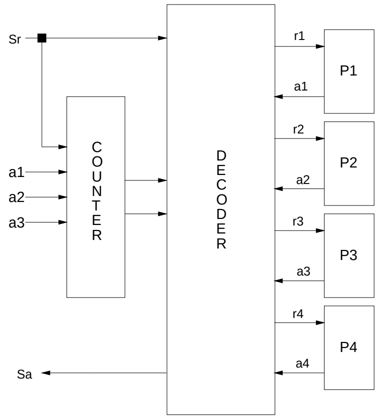


Figure 3.3: Josephs/Bailey Counter-Decoder Sequencer.

Seeking to improve on the Tangram circuit, Josephs and Bailey introduced a centralized sequencer [4]. Figure 3.3 shows a schematic *counter-decoder* sequencer. The counter centralizes the state of the sequencer and the decoder distributes the signals to the processes resulting in improved initial and phase inter-process latencies compared to the Tangram tree sequencer. The circuit is speed-independent and is currently used in handshake circuits [93]. Minor problems are that the circuit is not modular and is designed to work with an even number of processes only.

Josephs/Bailey Chain Sequencer

A different architecture, also introduced in [4], is shown in Figure 3.4. This n-way sequencer is built as a linear chain of n modules, each controlling a process. There are three different types of modules: X, Y, and Z. The sequencer uses one X module to control the first process, one Y module to control the last process, and n-2 Z modules to control the intermediate ones. The circuit implementations of the modules are also shown in the figure. The modules assume fundamental-mode operation and rely on reasonable timing assumptions to operate correctly.

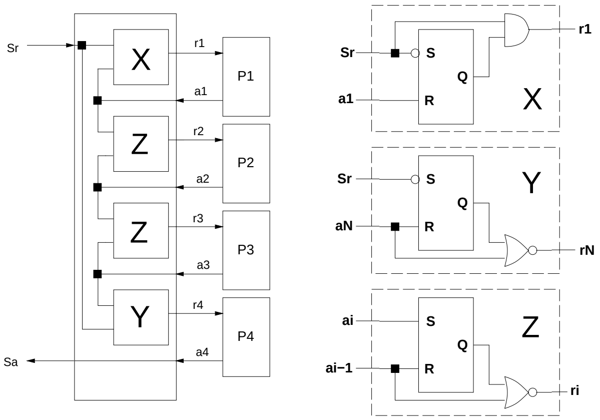


Figure 3.4: Josephs/Bailey Chain Sequencer.

Even though the implementations by Josephs and Bailey have improved area, performance, and power over the Tangram sequencers, They suffer from long inter-process latencies (see Section 3.6). To obtain better results, an alternative approach is needed.

3.1.2 Concurrent Protocol

A sequential protocol inherently has low throughput, *i.e.,* long inter-process latencies, due to the alternation of processing and return-to-zero phases. An attractive alternative approach is a *concurrent protocol*, which allows higher levels of concurrent activity in the system.

In a *concurrent protocol*, the sequencer can start process P_{i+1} *without* waiting for P_i to complete its return-to-zero phase. In this protocol, every processing phase P_{i+1} *overlaps* the return-to-zero phase R_i of the previous process, thus providing roughly twice the throughput, as shown in Figure 3.5.

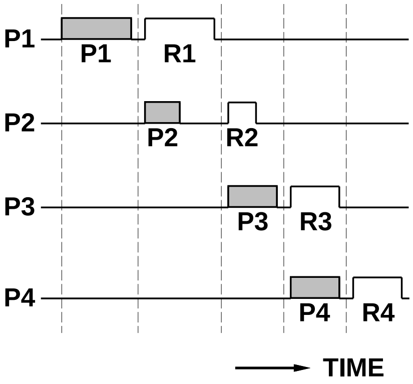


Figure 3.5: 4-Way Sequencer: Concurrent Protocol.

Only a few concurrent sequencers have been proposed:

Unger Tree Sequencer

As part of his building block approach [87], Unger presents a *2-step module* that implements a 2-way sequencer. Figure 3.6 shows the module symbol and its circuit implementation.

The 2-step is activated by a request in S_r. It starts a 4-phase handshake with

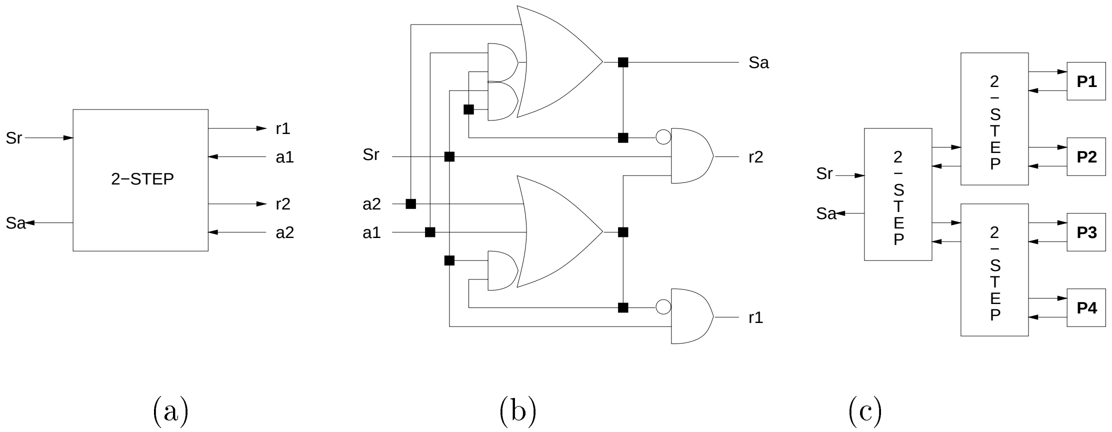


Figure 3.6: 2-step Module, Circuit Implementation and Tree Sequencer.

process P_1 using r_1. When P_1 signals the completion of the computation using a_1, the 2-step *concurrently* starts the processing phase of P_2 (by asserting r_2) *and* the return-to-zero phase of P_1 (by de-asserting r_1). Similarly, when a_2 is asserted, the 2-step *concurrently* de-asserts r_2 *and* asserts S_a. At this point, the module waits for $S_r\downarrow$, $r_1\downarrow$, and $r_2\downarrow$ to de-assert S_a to indicate the end of the sequence. This overlapped protocol can be described by the following expression, where '$||$' is the "parallel" operator, and ';' is the "sequential" operator:

$$*(s_r\uparrow; r_1\uparrow; a_1\uparrow; (r_1\downarrow; a_1\downarrow) \;||\; (r_2\uparrow; a_2\uparrow; (r_2\downarrow; a_2\downarrow) \;||\; (s_a\uparrow; s_r\downarrow)); s_a\downarrow)$$

The 2-step assumes fundamental-mode operation and relies on reasonable timing assumptions. In fundamental mode, no new inputs can arrive until the component has stabilized from a previous input change [88].

An n-way sequencer can be built as a balanced tree of n-1 2-step modules [87], as shown in Figure 3.6(c). There are several problems with this implementation. First, the sequencer has a long *initial latency*. Second, the *inter-process latency* is variable and can be several gate delays long. It depends on how far up and down the tree the signals have to propagate. Finally, the area and power consumption of this structure are significantly worse than previous designs (see Section 3.6).

Kagotani/Nanya Auto-Sweeping Module (ASM)

Kagotani and Nanya [39] introduced the Auto-Sweeping Module, which is the basic module used to implement a concurrent n-way sequencer. Figure 3.7 shows the ASM symbol and its implementation.

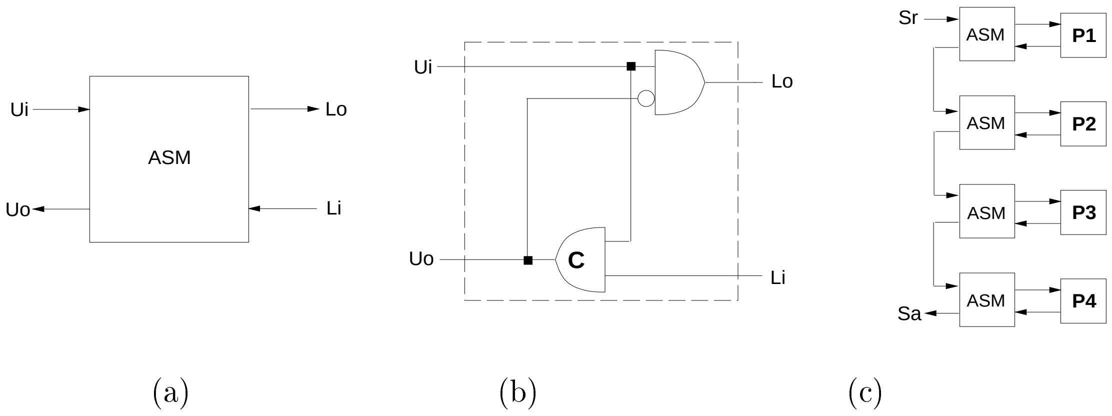


Figure 3.7: Auto-Sweeping Module, Implementation and Chain Sequencer.

An n-way sequencer is built as a linear chain of n ASM modules, as shown in Figure 3.7(b). The sequencer has a long inter-process latency due to the C-element[1] in series with the AND gate of the following stage. Also, and most important, $S_r \downarrow$ has to propagate through a chain of n C-elements to generate $S_a \downarrow$ and complete the 4-phase handshake. This may have a large impact on the throughput of the system.

Farnsworth/Edwards/Liu/Sikand Sequencer

It may be interesting to note that Farnsworth *et al.* [25] used a concurrent 2-way sequencer as part of a FIFO control unit, but did not discuss N-way extensions.

[1] A C-element is a basic asynchronous primitive; when both inputs are 0 (1), the output is 0 (1); otherwise, the output holds its prior value [83].

The 2-way sequencer is very similar to the ASM.

Each of the above sequencers has some drawbacks. First, the sequencers have either long initial latency or long inter-process latency. A detailed comparison is presented in Section 3.6. Second, and most important, the use of concurrent sequencers may introduce *data hazards* in the datapath (see Section 3.3). Of the above methods, *only* Kagotani and Nanya address this problem and present a solution. A detailed discussion of the Kagotani/Nanya approach is presented in Section 3.4.6.

3.2 New Concurrent Sequencers

We now introduce three new concurrent sequencer designs, which are faster, more compact and more energy-efficient than existing designs.

3.2.1 Tightly-Coupled Sequencer

The first new sequencer implements a *tightly-coupled* concurrent protocol: processing phase P_i overlaps *exactly* the return-to-zero phase R_{i-1}. The key point is that this sequencer waits until *both* concurrently operating phases, R_{i-1} and P_i, complete before starting the next two overlapped phases, R_i and P_{i+1}, as shown in Figure 3.8. The arrows indicate that, for example, processing phase P_3 starts after both P_2 and R_1 have finished.

Figure 3.9 shows a *burst-mode* specification [66] for the behavior of the n-way sequencer. The concurrent start and completion of the overlapped phases is clear in this specification. Each arc in the specification indicates an input burst (*i.e.,* input change) followed by output burst. A "+" means a rising transition, and a "-" means a falling transition.

An n-way burst-mode sequencer circuit was synthesized using an existing

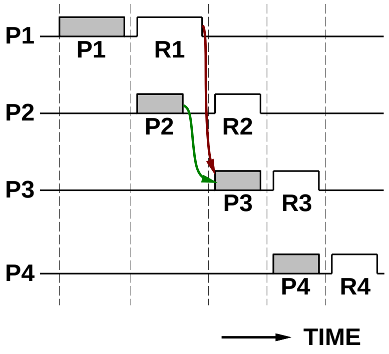


Figure 3.8: Tightly-Coupled Sequencer Operation.

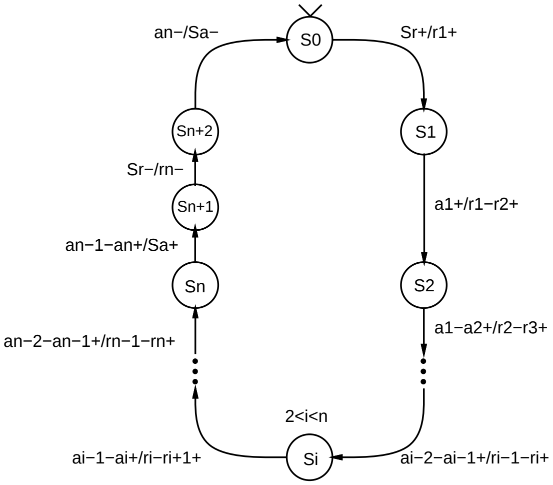
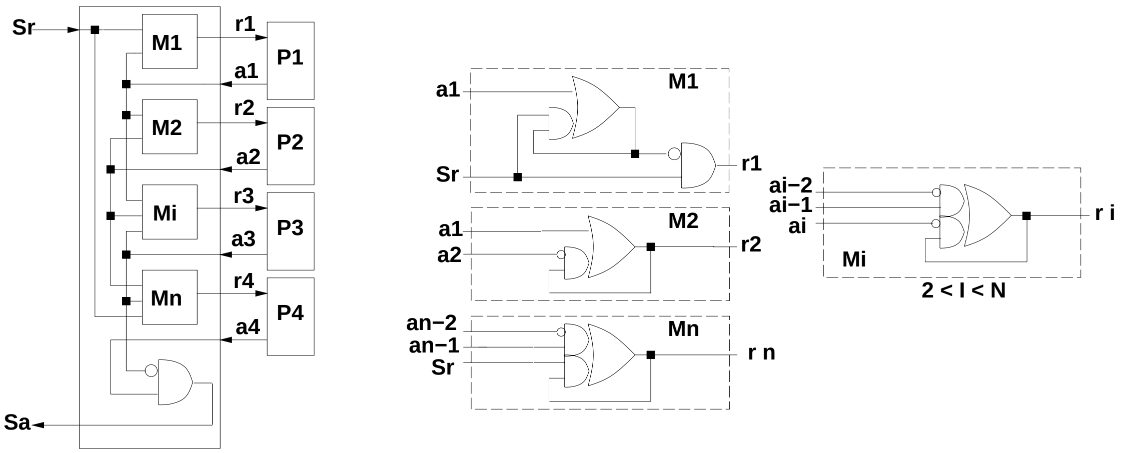


Figure 3.9: Tightly-Coupled Sequencer Burst-Mode Specification.

burst-mode asynchronous tool (UCLOCK) [63]. The result is a modular design, well suited for distributed control. Our N-way sequencer has N modules organized into 4 types as shown in Figure 3.10(a): module M_1 controls process P_1, M_2 controls P_2, M_i modules control P_3 to P_{N-1}, and M_n controls P_N. The sequencer is very efficient in terms of speed, area and power: the inter-process latency is only 2 CMOS gate delays.



(a) (b)

Figure 3.10: Tightly-Coupled Sequencer: (a) Block Diagram, (b) Modules.

The sequencer operates as follows. A request on S_r activates module M1 which starts a 4-phase handshake with process P_1 by $r_1\uparrow$. P_1 then responds with $a_1\uparrow$; modules M1 and M2 both receive this signal. BM1 will respond with $r_1\downarrow$ while, *concurrently*, M2 will start a 4-phase handshaking with with P_2 by $r_2\uparrow$. As a result, the reset phase of the first process ($R1$) overlaps the next computation ($P2$). The sequencer then waits for the completion of *both* phases to proceed: once $a_1\downarrow$ and $a_2\uparrow$ have both arrived, M2 continues the handshaking with P_2 concurrently with starting a 4-phase handshake with P_3. As a result, R_2 overlaps P_3. The same behavior continues until the end of the sequence.

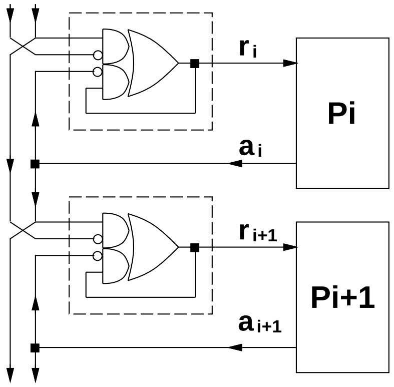


Figure 3.11: Tightly-Coupled Modules Controlling Processes.

Figure 3.11 shows two M_i modules, each controlling a process. The modules have good latency, area and power. In typical computation, the inter-process latency, —the time from completion of P_i's processing phase ($a_{i-1}\uparrow$) to the start of P_{i+1}'s processing phase ($r_i\uparrow$)—, is *only 2 CMOS gate delays* (an AOI-gate followed by an inverter).[2] Also, each module contributes only 8 gate output transitions and 14 transistors to the energy consumption and area, respectively, of the system.

The correct operation of the sequencer relies on modest timing assumptions, due to the fact that the acknowledge signal of each process, a_i, is forked to two different modules. In particular, $a_i\uparrow$ generates, concurrently, $r_i\downarrow$;$a_i\downarrow$ in module i, and $r_{i+1}\uparrow$ in module $i+1$. The change in r_{i+1} must propagate back to the input of the complex gate before $a_i\downarrow$ arrives at this gate, to avoid an unspecified change in r_{i+1}. Since r_{i+1} has to propagate through a short wire while r_i has to propagate through process P_i, this restriction is quite reasonable in practice.

One limitation of our tightly-coupled sequencer is that a long return-to-zero

[2] If a return-to-zero phase is unusually long, the inter-process latency may increase due to synchronization dependencies. In particular, P_i can be delayed by R_{i-2}.

phase, such as R_1 in Figure 3.8, may unnecessarily delay the start of the next processing phase (P_3). This observation leads to our second design.

3.2.2 Loosely-Coupled Sequencer

We now introduce a second concurrent sequencer which allows greater concurrency than the previous one.

In the tightly-coupled protocol implemented above, each processing phase exactly overlaps the previous return-to-zero phase. Our second sequencer allows greater concurrency by using a more relaxed synchronization requirement. By starting P_{i+1} as soon as P_i is finished, *independently* of the status of R_i, a faster sequence of processing phases is allowed. The operation of this loosely-coupled sequencer is shown in Figure 3.12.

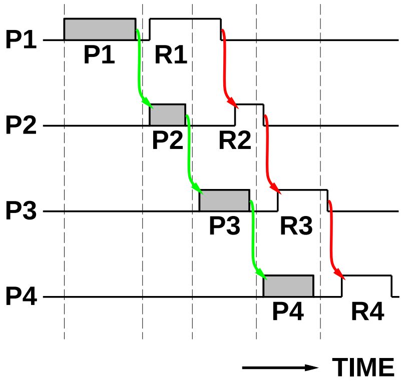
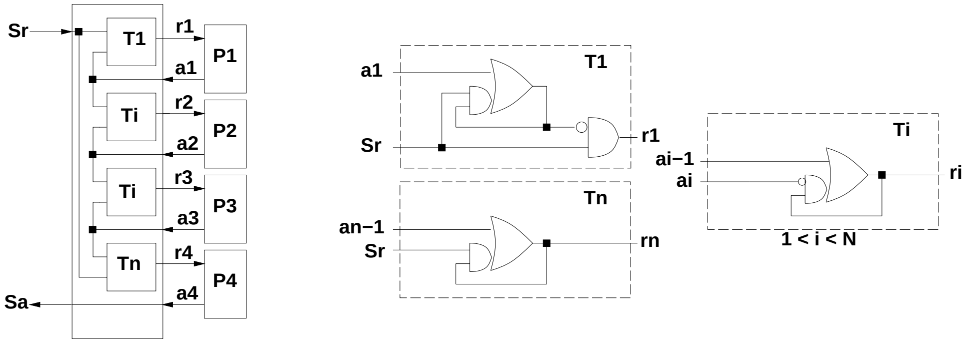


Figure 3.12: Operation of the Loosely-Coupled Sequencer.

The sequencer has a modular design, well suited for distributed control, and the inter-process latency, from $a_i\uparrow$ to $r_{i+1}\uparrow$, is 2 CMOS gate delays (an AOI gate followed by an inverter). The circuit consists of 3 different types of modules ($T1$, Ti, and Tn) organized in a chain, as shown in Figure 3.13(a). Figure 3.14 shows two adjacent Ti modules, each controlling a process. The modules are efficient in terms of speed, power and area (see details in Section 3.6).



(a) (b)

Figure 3.13: Loosely-Coupled Sequencer: (a) Block Diagram, (b) Modules.

The correct operation of this sequencer also relies on a modest timing assumption. Once signal $a_i\uparrow$ is generated, $a_i\downarrow$ cannot occur until the $i+1$st controller stage is stable. This is a *fundamental-mode* assumption [88]: no new input can arrive until the sequencer has processed the previous input change. This restriction is quite reasonable in practice: signal $a_i\uparrow$ must propagate through the $i+1$st controller stage, before its fork propagates through the ith controller stage ($r_i\downarrow$) *and* process P_i ($a_i\downarrow$).

Module T_i requires an external reset signal. This reset mechanism can be implemented with 2 transistors and has a small impact on the performance of the circuit (See Section 3.6).

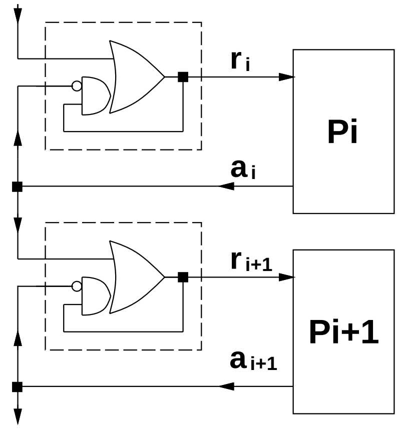


Figure 3.14: Loosely-Coupled Sequencer: Two Intermediate Stages.

3.2.3 Speed-Independent Sequencer

The tightly- and loosely-coupled sequencers introduced above operate in fundamental mode. In some environments, the timing assumptions related to this operation mode may be difficult to insure. In such cases, a speed-independent system may be used. In this section we introduce a new concurrent *speed-independent* sequencer.

A speed-independent circuit is one which operates correctly assuming arbitrary, finite, gate delays [52]. Our speed-independent design has more robust operation, at the cost of a slight increase in power and area.

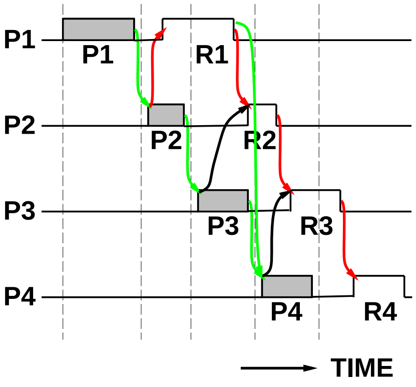
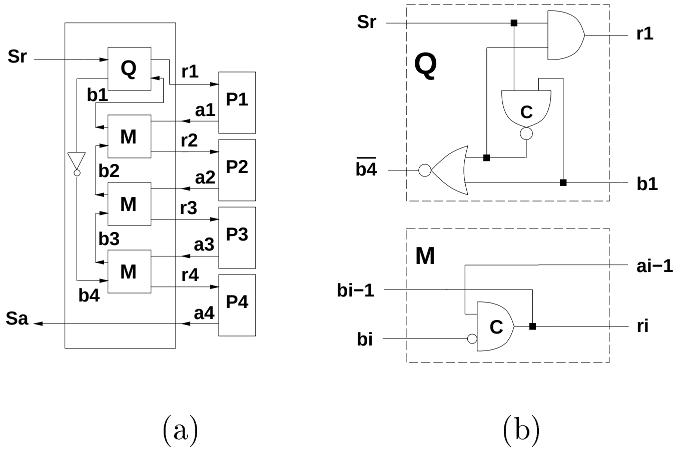


Figure 3.15: Operation of the Speed-Independent Sequencer.

Figure 3.15 shows the operation of our speed-independent sequencer, and Figure 3.16(a) shows the implementation. The first process, P_1, is controlled by a Q module [45]; remaining processes are controlled by M modules. Module implementations are shown in Figure 3.16(b). An M module has an efficient implementation: a single C-element[3].



(a) (b)

Figure 3.16: Speed-Independent Sequencer: (a) Block Diagram, (b) Modules.

A speed-independent circuit *acknowledges* every input change to indicate that it is stable and ready to accept new changes [45, 52]. This is true for our speed-independent sequencer. An M module acknowledges changes in both inputs using r_i. Clearly, when r_i changes, the module is stable and can accept further input changes. In the Q module, a change in b_1 produces a change in r_1 after the module is stable. A change in signal S_r produces a series of changes in module Q. When Q is ready to accept a new change in S_r, it produces a change in signal $\overline{b_4}$ which in turn enables a change in S_a (in the last M module), thus allowing new

[3] This sequencer works for $N > 2$ (the circuit will deadlock for $N = 2$). A 2-way sequencer can be built using an S-element to control the first process, an M-module to control the second process and an additional M-module to generate S_a.

changes in S_r.

A reset input is required for the C-element in the M module. Bailey [3] introduced a C-element with reset that uses only 2 additional transistors, with no significant impact on performance (based on SPICE simulations).

3.3 Data Hazards in Concurrent Operation

The use of concurrent sequencers increases the throughput of the entire system. Unfortunately, existing circuits may not operate correctly at the increased level of performance. In this section we show that data hazards may appear. We then introduce mechanisms to guarantee that both dual- and single-rail systems operate correctly, without hazards, using our new sequencers.

In the context of datapath operation, a *data hazard* [35] represents the possibility that a wrong data value is used in a computation or is stored in a register. Data hazards cannot occur in a sequential protocol: in this case, a datapath operation starts only after the previous one has completed. On the other hand, in a concurrent protocol, two or more operations may execute concurrently. This overlapped operation introduces the possibility of data hazards. In particular, if two concurrent operations involve the same register, data hazards may arise during concurrent accesses to that register.

Figure 3.17 shows schematically a 2-stage dual-rail datapath. In this example, a sequencer controls two processes, where *Process 1* implements $Z = F(Y)$ and *Process 2* implements $X = G(W)$. There are four possible forms of interaction when concurrent processing and return-to-zero phases access the same latch (each phase can read or write the latch).

Read after Read (RAR). [$W = Y$ in Figure 3.17] In this case, data in the common latch does not change and remains stable. Thus, no data hazard can occur.

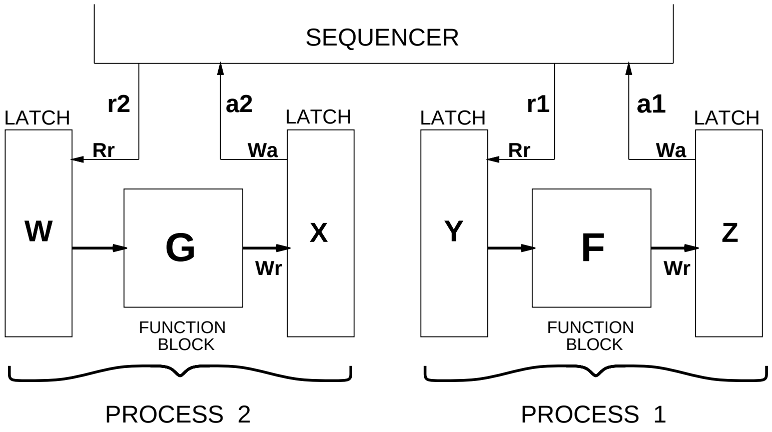


Figure 3.17: Sequencer controlling 2 Dual-Rail Datapath Processes.

Read after Write (RAW). [$W = Z$ in Figure 3.17] In this case, the read (second) operation overlaps the return-to-zero phase of the write (first) operation, *i.e.,* the second computation reads data that has already been written to the latch and is stable. Again, no data hazard occurs.

Write after Write (WAW). [$X = Z$ in Figure 3.17] When two different sources can write to the same latch, a multiplexer module [72, 94] is typically used, as shown in Figure 3.18. In a sequential protocol, no data hazard occurs since the first write is completed before the second write begins. However, in a concurrent protocol, the two writes overlap, causing a conflict in the multiplexer. Thus, a WAW hazard may occur. To avoid the hazard, the second write operation must be stalled until the first one is complete.

Write after Read (WAR). [$X = Y$ in Figure 3.17] This situation is illustrated in Figure 3.19.

A WAR hazard occurs when a read is first initiated in X, making the read port transparent. Before the read operation is completed, a write is initiated. New data in the write port can propagate through the latch to the read port, causing undesired changes in the output data.

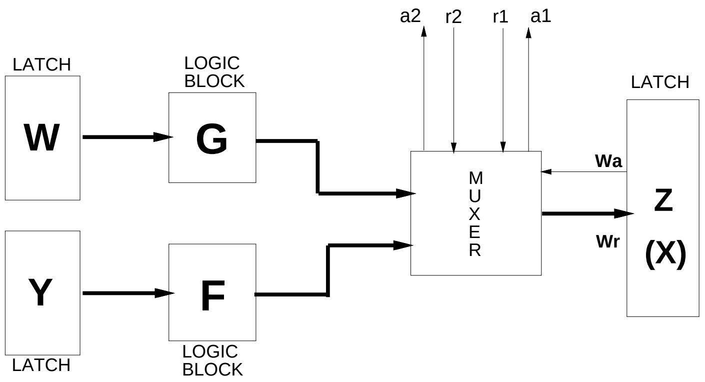


Figure 3.18: Scenario for a WAW Data Hazard.

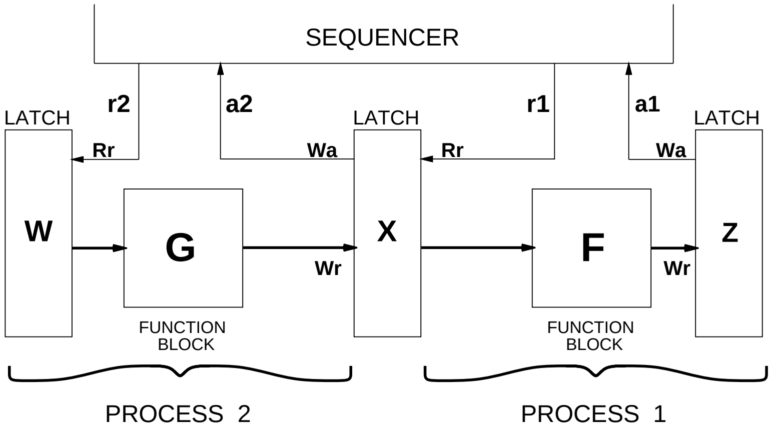


Figure 3.19: Scenario for a WAR Data Hazard.

Summarizing, of the four possible forms of concurrent accesses to a register, two (RAR and RAW) are hazard-free and only two (WAW and WAR) can introduce data hazards. We must introduce mechanisms to avoid the presence of hazards in these cases.

3.4 Data Hazard Elimination

Data hazards can be eliminated by avoiding concurrent accesses to the shared latch. This can be achieved at the algorithm level, by introducing changes in the order of the processes, or at the architecture level, by introducing additional circuitry. The following sections examine the solutions in detail.

3.4.1 Algorithm Modification

At the algorithm level, a designer (or a compiler) can easily identify a WAW or a WAR hazard between two *consecutive* computations. For example, Figure 3.20 shows the algorithm for the two-place ripple shift register introduced in Chapter 2. There is a potential WAR hazard between instructions labeled A and B. The shared register is $X0$ (there is a similar data hazard involving $X1$).

```
begin
  REPEAT forever
    OUT = X1;
    X1 = X0;      ⟵ A
    X0 = IN       ⟵ B
end
```

Figure 3.20: Macromodular System High-Level Specification.

To avoid the hazard, the two operations must execute sequentially, not concurrently. An unrelated operation can be inserted between the two conflicting instructions to eliminate the hazard. This technique requires the use of our tightly-coupled sequencer, which allows WAW and WAR interactions only between two consecutive computations.

```
begin
    REPEAT forever
        OUT = X1;
        null;
        X1 = X0;
        null;
        X0 = IN
end
```

Figure 3.21: Macromodular System High-Level Specification.

In a case in which such reordering of processes is not possible, a special, *null*, operation is inserted, as shown in Figure 3.21. A null operation is a process that simply returns an acknowledge whenever a request is asserted.

3.4.2 Architectural Solution

Data hazards can also be eliminated at the circuit level, without modifications to the algorithm. The basic idea is to introduce circuitry that detects conflicting concurrent accesses to a latch and stalls the second operation until it is safe to proceed. The additional circuits are called *interlock circuitry.*

The interlock circuitry has an effect on system performance. The interlocks stall an operation, effectively delaying its completion. It is important to design the interlocks so that *only* potentially hazardous operations are stalled, and only for the minimum time necessary to guarantee correct operation.

The following subsections present the proposed architectural solutions for WAW and WAR data hazards. Elimination of WAR hazards requires different techniques for dual-rail and single-rail datapaths. We propose solutions for both types of datapaths.

3.4.3 WAW Hazard Elimination

In the case of WAW hazards, the second write operation must be stalled until the first one is complete. Figure 3.22 shows the datapath with multiplexer, and an added interlock: an AND gate is used to stall the second write operation until the first one is complete.

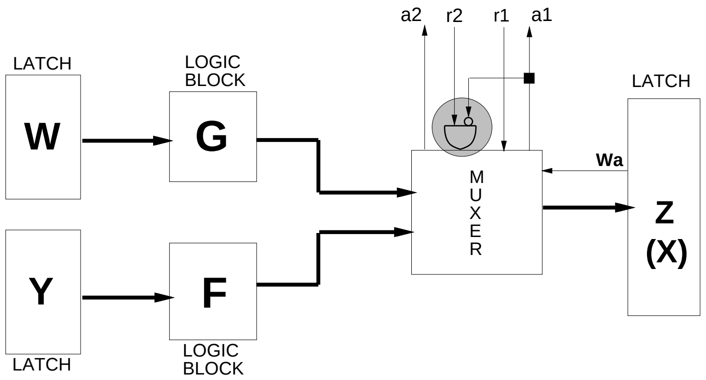


Figure 3.22: Interlock Circuit to Avoid WAW Hazard.

3.4.4 WAR Hazard Elimination in Dual-Rail Datapaths

Basic Solution

To eliminate WAR data hazards, the write operation must be *stalled* until it can be safely executed, *i.e.,* until the read port of the latch is closed (opaque). The read request signal ($\overline{R_r}$) controls the state of the read port and can be used as an enable signal for the write operation.

In dual-rail datapaths, WAR hazards can be eliminated using the interlock circuitry shown in Figure 3.23. The addition of the *inverter* and *AND* gates makes $\overline{R_r}$ into an enable signal for the write port. When the read port is transparent, R_r is high ($\overline{R_r}$ is low), thus preventing a write operation.

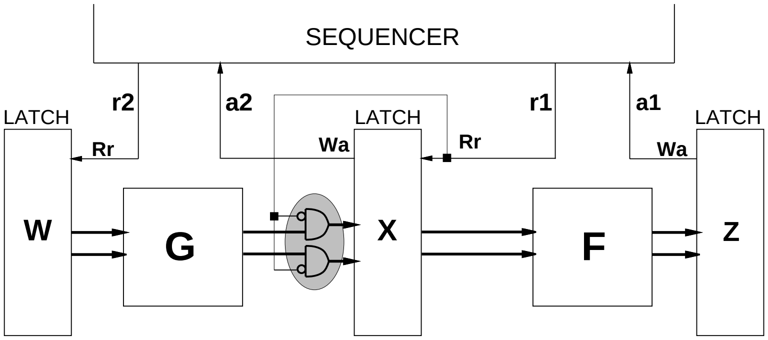


Figure 3.23: Interlock Circuit to Avoid WAR Hazard in Dual-Rail Datapaths.

In practice, this interlock circuitry should have minimal impact on performance. First, the interlock mechanism will rarely be activated, because $r_1\downarrow$ propagates through a wire to produce $R_r\downarrow$, while $r_2\uparrow$ must propagate through W and G to generate a write request ($W_0\uparrow$ or $W_1\uparrow$). Second, even if the interlock is activated, the write will be stalled only for the duration of the race and not for the entire phase. Finally, even though the AND gates are on the critical path, the increased delay is small compared to the performance benefits of concurrent operation.

Optimized Solution

The interlock mechanism can be optimized, when incorporated directly into an existing dual-rail latch, as shown in Figure 3.24. The shaded region indicates our added interlock. In this latch, if the data to be written is *the same* as the currently stored data, *no stall is required,* since no data hazard can occur. In this case, the circuit can immediately acknowledge the write request. This is a safe optimization: if the same data is written, no signals will change inside the latch.

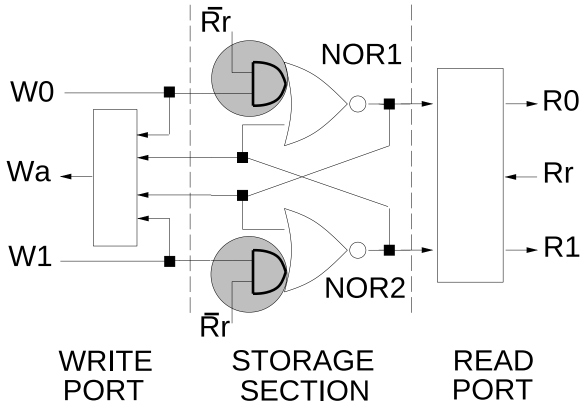


Figure 3.24: Modified Dual-Rail Latch Gate-Level Implementation.

Figure 3.25 highlights our changes at the transistor-level of the latch. This solution requires the addition of only 2 transistors.

The correct operation of the circuit relies on reasonable timing assumptions. In a WAR interaction, the read operation is always started first, so $\overline{R_r}\downarrow$ will safely disable any write operation. When the read is complete, $\overline{R_r}\uparrow$ must make the latch opaque (one gate delay), before the new write data arrives (three gate delays).

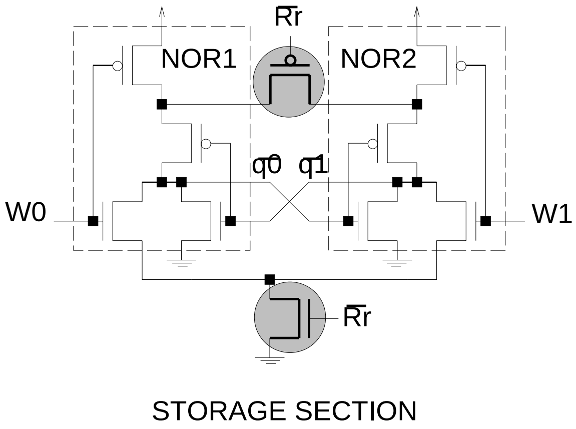


Figure 3.25: Modified Dual-Rail Latch Transistor-Level Implementation.

3.4.5 WAR Hazard Elimination in Single-Rail Datapaths

A key difference between single-rail and dual-rail datapaths is that single-rail latches have no read port. Therefore, a different approach is used to eliminate WAR hazards: the write operation must be stalled until the entire first computation is complete, *i.e.,* until the destination latch Z is opaque.

Basic Solution

Figure 3.26 shows the basic interlock mechanism to stall the write operation. In this case, the *acknowledge* signal from the destination latch (a_1) is used as an enable to the source latch write request. This mechanism guarantees correct operation, but limits the performance improvement obtained.

Optimized Solution

Two optimized schemes can be used, each of which assume certain timing constraints. First, if stages have little or no computation (*e.g.,* shift registers), very small or no matched delays are used, and control overhead tends to dominate. In this case, the request signal itself ($r_1 \uparrow$) can be used to stall the write request.

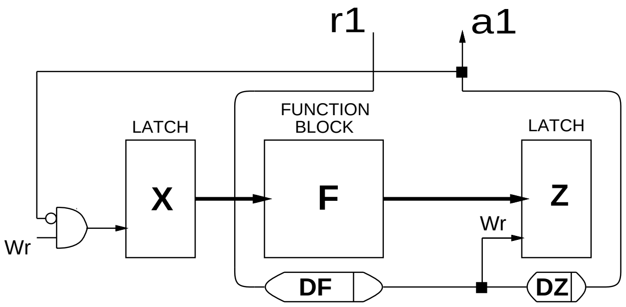


Figure 3.26: Robust Interlock to Avoid Single-Rail WAR Hazards.

Figure 3.27 shows this *fast interlock*, which provides better performance than the *robust interlock* of Figure 3.26. For correct operation, this circuit assumes that delay DF is faster than the delay of the AND-gate plus latch X.

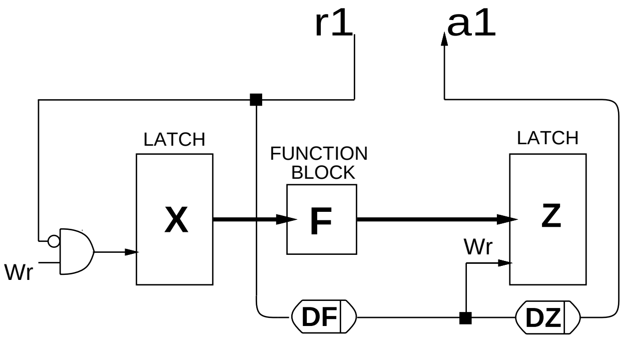


Figure 3.27: Avoiding Single-Rail WAR Hazards: Fast Interlock.

Second, an *asymmetric matched delay* [79] DF can be used. In this case, no interlock is needed. An asymmetric delay matches the function block in the processing phase, but has a fast reset in the return-to-zero phase. Therefore, $W_r\downarrow$ arrives quickly at the *destination latch* Z, which becomes opaque before new data arrives. The write operation is never stalled, and performance is improved. For correct operation, $W_r\downarrow$ must arrive at the destination latch, Z, before the source latch, X, becomes transparent and new data propagates through F.

3.4.6 Comparison with Kagotani/Nanya Approach

Of existing work on concurrent operation in dual-rail datapaths, only Kagotani and Nanya address the issue of data hazards [39]. Their work appeared at the same time as ours [73].

Section 3.1.2 above presented the ASM module introduced by Kagotani and Nanya as a concurrent sequencer. Kagotani and Nanya also developed an interlock mechanism to avoid WAR hazards in dual-rail datapaths, shown in Figure 3.28. Their interlock is similar to ours, in that they use latch X read request signal as an enable for the write operation. However, unlike our approach, it actually stalls the read of the source register (W) of the second operation, stalling the processing phase, thus stalling the entire write operation.

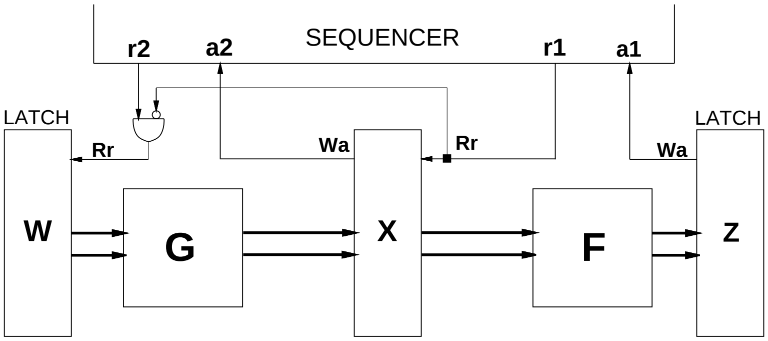


Figure 3.28: Kagotani/Nanya Interlock Mechanism for Dual-Rail Datapaths.

The Kagotani and Nanya approach has several drawbacks. First, only dual-rail datapaths are considered. Second, their concurrent sequencer has greater area, power, and inter-process latency than ours (see Tables 3.1 and 3.2). Finally, their interlock scheme has much greater impact on performance than ours, since it stalls a write operation at the beginning of the processing phase (*i.e.,* reading of source register W) rather than at the end (*i.e.,* writing of destination register X; see Figure 3.23).

3.5 Comparison with True Four-Phase Operation

No other concurrent sequencing approach has been proposed, for single-rail datapaths, which addresses the problem of data hazards. However, a novel sequential scheme, limited to single-rail systems, was introduced by Peeters and van Berkel [72], which attempts to achieve similar high throughput as that obtained by concurrent sequencing, without introducing data hazards. In this section we compare the operation of our new sequencers with this scheme and show that our approach has several advantages.

The Peeters and van Berkel *true four-phase (TFP) scheme* uses a sequential protocol. However, unlike a typical sequential scheme, delay elements are designed to match only *half* the worst-case delay in the functional blocks. During the processing phase, only half of the computation occurs. The second half of the computation takes place in the return-to-zero phase. Therefore, the function block computes throughout the entire handshaking cycle, and there is no dead time.

Figure 3.29 shows timing diagrams for sequential, TFP and concurrent operation. Clearly, the length of the processing and return-to-zero phases is the same in sequential and concurrent operation, but is reduced to a half in TFP. Another key difference, also shown in the figure, is that in sequential and concurrent operation the destination latches remain opaque during computation while in TFP the latches are transparent during the second half of the computation.

The advantage of the TFP scheme is that, as in our approach, it can obtain roughly twice the throughput of a normal sequential protocol. However, there are several key drawbacks:

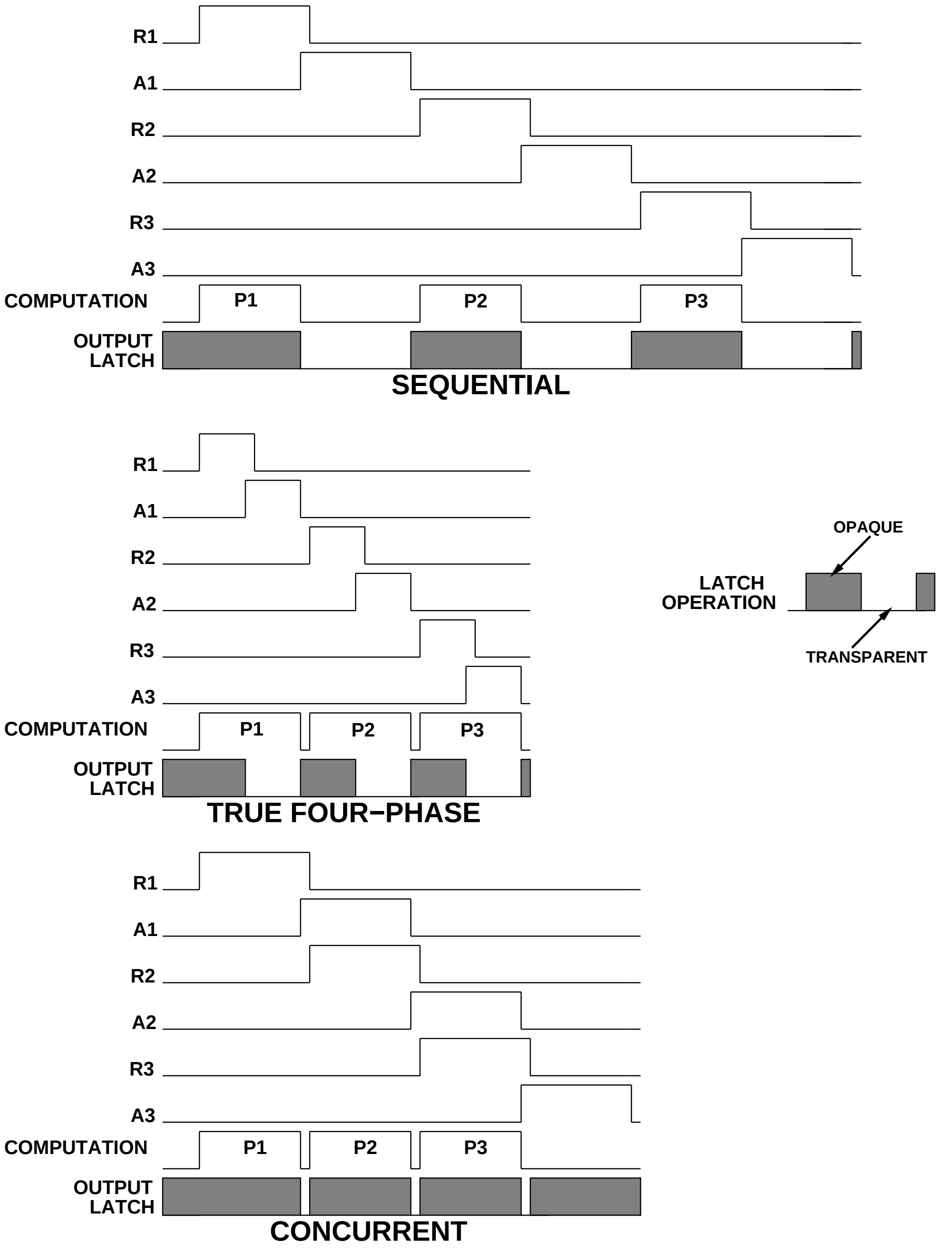


Figure 3.29: Single-Rail Operation: Sequential, TFP and Concurrent.

Glitch Propagation. Glitches can cause significant power consumption, especially if they can propagate through latches connected to deep combinational circuits [27, 57]. In our approach, glitch propagation does not occur, since the destination latch is opaque during computation. In TFP, glitch propagation *can* occur, since the destination latch is transparent throughout the entire return-to-zero phase while computation is taking place. Figure 3.29 illustrates the difference in latch operation in the two schemes. SPICE simulations indicate that glitch propagation may contribute up to 40% more energy consumption in TFP than using our scheme (see Section 3.6).

Performance and Design Overhead. TFP has several overheads compared to our scheme. First, TFP uses sequential sequencers; these have greater interprocess latency, area and power than our concurrent sequencers (*cf.* Josephs/Bailey sequencers in Tables 3.1 and 3.2).

Second, TFP is not easily applicable to *dynamic-logic implementations.* Typically, in an asynchronous dynamic implementation, the processing phase corresponds to the *evaluate phase*, and the return-to-zero phase corresponds to the *precharge phase* [31, 26]. However, in TFP, both processing and return-to-zero phases will be used to match the *evaluate phase.* To implement a precharge phase, either (i) separate control signals must be introduced, or (ii) the processing phase must match the evaluate phase, and return-to-zero must match the precharge phase, reducing the operation to a simple sequential scheme, with degraded performance.

Finally, for the special case of *fine-grained stages*, which have little or no computation, TFP may have significant control overhead: 2 cycles through control in TFP vs. 1 cycle through control in our scheme. In this case, our system may be up to 40% faster (see Section 3.6).

Delay Matching. In high-performance single-rail datapaths, tight margins in delay matching are necessary. Matched delays that accurately model the logic are usually built by using a single extracted portion of the critical path [19, 32, 65], with similar layout and loading. However, this approach is not directly applicable to *half-matched delays* in TFP, which may therefore require larger safety margins, thus degrading performance.

3.6 Results

In this section, we present the results of detailed comparisons of the characteristics of our new sequencers with existing ones. We also show the results of SPICE simulations to compare the performance and energy consumption of an entire system, using our new concurrent sequencers versus a sequential system. Finally, we use SPICE simulations to compare the operation of our concurrent sequencers with True Four-Phase operation.

3.6.1 Sequencer Comparison

We first compare the static characteristics of several sequencers in Table 3.1. N is the number of processing stages being sequenced. Transistor and gate-output transition counts are used as approximations to area and energy consumption, respectively. The table also shows the timing model (fundamental mode or speed-independent) of each design. The table indicates that our *loosely-coupled* and *speed-independent sequencers* have better area and energy consumption than existing designs.

Table 3.2 compares the *dynamic behavior* of different sequencers. Each sequencer is assumed to control N identical processes. g is roughly the delay associated with a CMOS complex gate or an inverter, P represents the length of a processing phase, and R is the length of a return-to-zero phase. For symmetric delays, P and R are roughly equal; for asymmetric delays, R is smaller than

	AREA	**ENERGY**	**TIMING**
SEQUENCER	# transistors	# gate output transitions	**MODEL**
Previous Designs			
van Berkel	18N–18	10N–10	SI
Martin	18N–18	10N–10	SI
Josephs/Bailey Counter-Decoder	15N–6	7N–2	SI
Josephs/Bailey Chain	12N+4	8N–2	FM
Unger Tree	36N–36	16N–16	FM
Kagotani/Nanya ASM chain	18N	10N	SI
New Designs			
Tightly-Coupled	14N–6	8N–4	FM
Loosely-Coupled	10N+2	6N	FM
Speed-independent	12N+2	6N+6	SI

Table 3.1: Static Characteristics of N-way Sequencers.

P. The table indicates that the our new sequencers have the shortest initial and inter-process latencies, and the best total computation times.

3.6.2 Simulation: Performance and Energy Consumption

We next use SPICE simulations to compare total system performance and energy consumption, using concurrent and sequential sequencers. The simulations use MOSIS 1.2μ technology parameters with a 5V power supply. Both dual-rail and single-rail systems are simulated. In each case, a datapath consisting of 4 generic stages is used (see Figures 2.9 and 2.10). Inverter chains are used to model function blocks (1 chain for single-rail blocks, 2 for dual-rail ones). The length of the chain determines the latency of the block (8 and 24 inverter chains were used in the simulations).

Figure 3.30 shows SPICE simulations of a *dual-rail system.* Figure 3.30(a) shows total computation time (represented by $S_a\uparrow$) and power consumption of a sequential implementation of the system, using the Josephs/Bailey chain sequencer [4] and van Berkel dual-rail latches [94]. Figure 3.30(b) also shows the total computa-

SEQUENCER	INITIAL LATENCY	INTER-PROCESS LATENCY	TOTAL COMPUTATION TIME
Previous Designs			
van Berkel	(2N-2)g	5g + R	(6N-8)g+NP+(N-1)R
Martin	(2N-2)g	5g + R	(6N-8)g+NP+(N-1)R
Josephs/Bailey Counter-Decoder	2g	4g + R	(4N-4)g+NP+(N-1)R
Josephs/Bailey Chain	2g	3g + R	(3N-2)g+NP+(N-1)R

Unger Tree	2(logN)g	MIN=2g
		MAX=[4(logN)-2]g
			(6N-6)g+NP

Kagotani/Nanya ASM chain	2g	4g	4Ng+NP
New Designs			
Tightly-Coupled	2g	2g†	(2N+2)g+NP
Loosely-Coupled	2g	2g	2Ng+NP
Speed-Independent	2g	2g	2Ng+NP

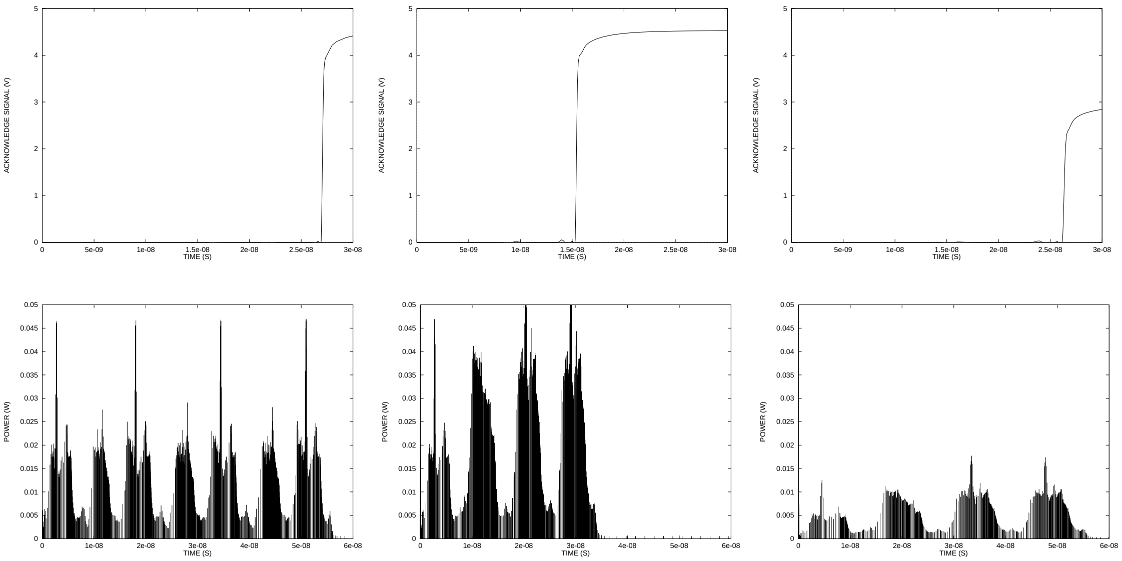
† If a r-t-z phase is unusually long, the inter-process latency may increase due to synchronization dependencies

Table 3.2: Dynamic Behavior of N-way Sequencers.

tion time and power consumption of our concurrent implementation of the system, using the loosely-coupled sequencer and modified dual-rail latches. Our system obtains a *67% improvement* in total computation time, and a 4% reduction in total energy consumption.[4]

Figure 3.31 shows the simulation results for a *single-rail system.* A *high-latency* system is modeled, where each stage has a 24-inverter matched delay. Figure 3.31(a) shows power consumption of a sequential implementation, using single-rail latches [71] and asymmetric matched delays. Figure 3.31(b) shows the power consumption of our concurrent design, using the loosely-coupled sequencer and symmetric matched delays. Asymmetric delays were used for the sequential protocol, since they yielded the best total computation time: 35.75 ns using asymmetric vs. 43.10 ns using symmetric delays. Symmetric delays were used for the concurrent protocol, since asymmetric delays have no benefit: the fast reset phase is hidden.

[4] If a reset mechanism is added to the sequencers, our system shows a 7% performance loss. The impact of the reset on the sequential system was not simulated.



(a) (b) (c)

SEQUENTIAL DESIGN **OUR DESIGN**

No Voltage Scaling **After Voltage Scaling**

Josephs/Bailey sequencer
+ Tangram dual-rail latches

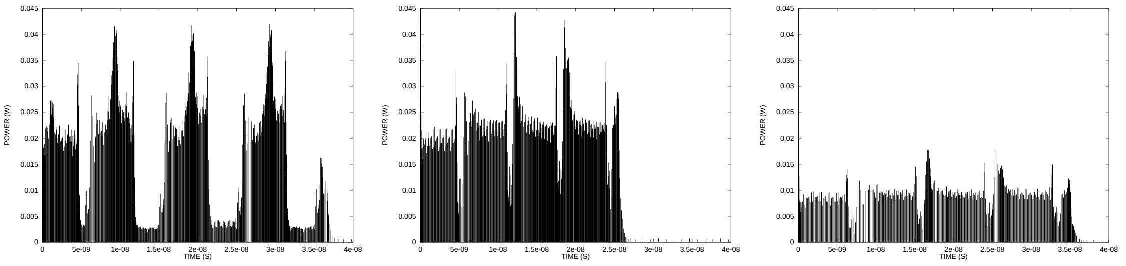
Loosely-coupled sequencer + modified dual-rail latches

Figure 3.30: Performance and Power Consumption of Dual-Rail System.

In this case, our design obtains a *45% improvement* in total computation time.

Finally, Figure 3.32 shows simulation results for a *low-latency* single-rail system, where each stage has an 8-inverter matched delay. This case indicates that the use of asymmetric delays is not always beneficial. Figure 3.32(a) shows the sequential implementation using symmetric matched delays, and Figure 3.32(b) shows the concurrent system (also using symmetric delays). Here, we used symmetric delays for the sequential protocol, since they yielded the best total computation time: 22.75 ns using symmetric vs. 24.25 ns using asymmetric delays. The simulations suggest that, for fine-grained computation, the increased loading and switching activity of asymmetric delays may dominate (see Figure 7.8 in [79]). In this case, our system obtains a *73% improvement* in total computation time.

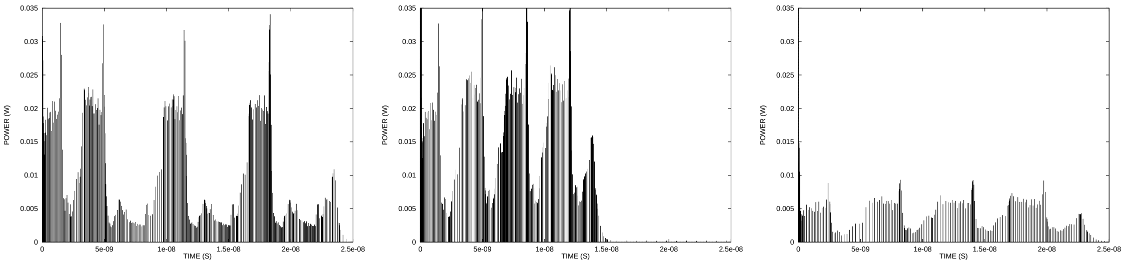
We also simulated the effect of applying *voltage scaling* to each of the 3



(a) **SEQUENTIAL DESIGN** Josephs/Bailey sequencer + asymmetric matched delays

(b) (c) **OUR DESIGN** **No Voltage Scaling** **After Voltage Scaling** Loosely-coupled sequencer + symmetric matched delays

Figure 3.31: Power Consumption of Single-Rail System (High-Latency).



(a) **SEQUENTIAL DESIGN** Josephs/Bailey sequencer + symmetric matched delays

(b) (c) **OUR DESIGN** **No Voltage Scaling** **After Voltage Scaling** Loosely-Coupled sequencer + symmetric matched delays

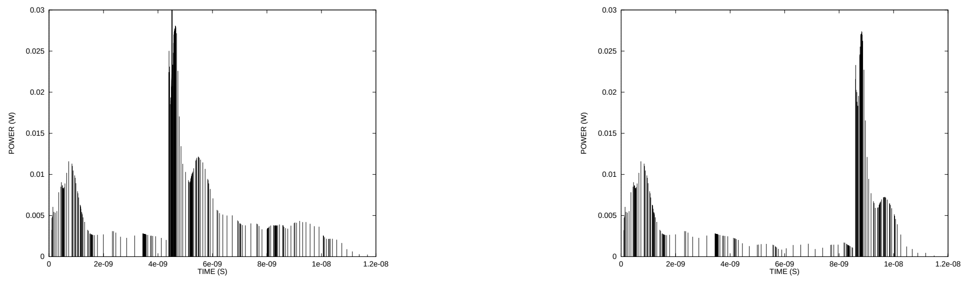
Figure 3.32: Power Consumption of Single-Rail System (Low-Latency).

systems: the power supply voltage is dropped until the total computation time is the same as the corresponding sequential system. In the dual-rail system, shown in Figure 3.30(c), the energy consumption of the entire system is reduced *by a factor of 2.4* compared to the sequential design. Figure 3.31(c) shows the simulation of our first single-rail system. In this case, energy consumption of the entire system is reduced *by a factor of 1.9*. Finally, our fast single-rail system, shown in Figure 3.32, reduces energy by a factor of 2.3.

3.6.3 Comparison with True Four-Phase Operation

A final comparison is between our concurrent scheme and the TFP scheme of Peeters and van Berkel.

Glitch Propagation. First, the impact of glitch propagation is analyzed. We simulated a 2-stage single-rail system: a 4-bit array multiplier followed by an 8-bit ripple-carry adder (with inputs $15 * 9 + 240$). Figure 3.33(a) shows a SPICE simulation of the datapath using *half-matched* delays, as in the TFP scheme. Figure 3.33(b) shows the simulation using *full-matched* delays, as in our scheme. Energy consumption using the TFP matching scheme was 40% more than using our full-matched delays, indicating the impact of glitch propagation in TFP.



(a)
TRUE FOUR PHASE
Half-Matched delays

(b)
OUR DESIGN
Full-Matched delays

Figure 3.33: Power Consumption of Multiplier/Adder Single-Rail Datapath.

Fine-Grained Stages. An 8-bit shift register was used to compare our approach with the TFP scheme in datapaths with little or no computation. SPICE simulations show that our system (using the fast interlock mechanism) was 40% faster than using TFP scheme, since control overhead in the return-to-zero phase is hidden in our scheme.

3.7 Conclusions

In this chapter we presented an architectural optimization for 4-phase macromodular systems. The goal of the optimization is to increase the throughput of a system by increasing its level of concurrent activity. This optimization can be applied to low-power applications such as the DCC error corrector in [92] and the FIR filter bank in [62].

Three new concurrent sequencers were introduced, which increase the concurrent activity and throughput of a system. The new sequencers have better speed, area and power characteristics than existing sequential and concurrent sequencers.

We showed that, when the new sequencers are used, data hazards may arise in existing datapaths. To avoid data hazards, several interlock schemes were proposed, for both dual-rail and single-rail implementations. SPICE simulations showed up to 73% performance improvement when the new concurrent sequencers are used, compared to sequential ones.

The technique can also be regarded as a low-power optimization. SPICE simulations show total energy reductions up to a factor of 2.4 after voltage scaling, using our approach over a sequential approach.

Finally, simulations also indicated that True Four-Phase operation, a novel sequential approach that attempts to obtain similar throughput and power benefits as our concurrent operation, may in some cases consume up to 40% more energy than ours, due to glitch propagation in the datapath.

Chapter 4

Pulse-mode Macromodular Systems

In this chapter, we explore the use of pulses as handshake events and investigate the use of pulse-mode modules in the implementation of macromodular systems and micropipelines.

In particular, the following contributions are presented. First, the use of pulses as events in an efficient handshaking protocol. Although the idea had been suggested previously, no systematic approach to the use of pulse-mode inter-module synchronization has been presented before. Second, the introduction of a more concurrent form of pulse-mode operation. Traditional pulse-mode systems allow a single pulse to be active at any time. Pulse burst operation (PBO), introduced in this work, allows concurrent pulses as inputs to a module. Finally, the design of a large set of pulse-mode macromodules that can be used to build cost-effective asynchronous macromodular systems. Such a set has not been proposed before. The set includes control macromodules as well as arbiters and converter modules to interface pulse-mode to traditional 2-phase and 4-phase circuits. The modules have very efficient implementation using standard CMOS gates.

This chapter is organized as follows. Section 4.1 introduces pulse-mode handshaking. This section also reviews traditional pulse-mode operation and introduces pulse-burst operation, including several implementation issues. Section 4.2 presents a large set of pulse-mode macromodules that efficiently implement control operations such as sequencing, selection, iteration, concurrency control and resource sharing. Macromodules that interface pulse-mode systems with 2-phase and 4-phase systems are introduced in Section 4.3. Section 4.4 discusses arbitration and introduces a pulse-mode arbiter. The use of pulse-mode macromodules to control dual-rail and single-rail datapaths is presented in Section 4.5. This section reviews both level and pulse-mode datapaths. Section 4.6 introduces optimizations that can be applied to macromodular systems. These optimizations usually improve area and performance. Section 4.7 presents the design of a micropipeline and an iterative multiplier, as application examples of the pulse-mode macromodules introduced in previous sections. Finally, Section 4.8 presents conclusions.

4.1 Pulse-Mode Handshaking

Pulses are used frequently in digital circuits and are intuitively associated with events. Pulse-mode handshaking is a form of 2-phase handshaking that uses pulses, instead of transitions, to represent events.

Figure 4.1 shows two macromodules that communicate with each other using *req* and *ack* wires. Initially, the two modules are idle and both signals are deasserted. *Module P* sends a pulse on *req* to start the operation of *module Q*. When *Q* finishes processing, it sends a pulse on *ack* to complete the handshake. After the handshake cycle, the state of the wires is the same as in the initial state and the modules can start a new handshake.

Pulse-mode handshaking combines the conceptual simplicity of the 2-phase protocol (only two events per handshake) with the level-based approach of 4-phase

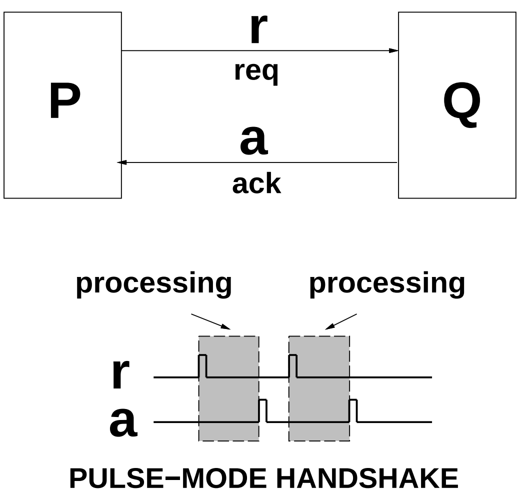


Figure 4.1: Pulse-Mode Handshaking for Module Communication.

handshaking. It has potential advantages over transition signaling and single-track implementations. As pointed out before, the initial and final levels of the request and acknowledge wires are the same. This, in general, leads to simpler circuits. Another important aspect of this protocol is that standard gates can be used to implement robust and efficient circuits. It has also potential advantages over 4-phase handshaking due to the absence of the return-to-zero phase.

4.1.1 Traditional Pulse-Mode Operation

Pulse-mode sequential circuits have been used since the very first digital designs. As a matter of fact, most existing sequential circuits correspond to this class of circuits: synchronous circuits are pulse-mode circuits that have a pulse input designated as the global clock, used for synchronization of the entire system. Detailed analysis and synthesis methods for pulse-mode sequential circuits are presented in [88, 50, 91].

Traditional pulse-mode operation is characterized by two properties [50, 91]: (*i*) at most one pulse input is in its active state at any given time, and (*ii*) other inputs are not expected to change while a pulse input is in its active state.

These two properties greatly simplify the design of these circuits. However, they allow only a limited form of operation. Pulse-mode handshaking requires higher levels of concurrency than traditional pulse-mode circuits. In the following section, a more general type of pulse-mode operation is introduced.

4.1.2 Pulse-Burst Operation

In macromodular systems, a module interacts with other modules through handshake channels. Every module operates at its own speed due to the lack of global synchronization. This means that handshaking can take place concurrently in different channels, violating property *(i)* above. A more general form of pulse-mode operation is required to accommodate the increased concurrency. We call it *Pulse Burst operation* (PBO), since it has similar properties to transition-based burst mode [66].

In this operation mode, multiple specified input pulses are allowed to arrive concurrently to a module. The input pulses can arrive in any order and at any time. This is called a pulse burst. When all the expected pulses have arrived the module can respond, generating the specified output pulses. A second pulse should not arrive at any input until the output pulses have occurred. To avoid ambiguous behavior, no specified input burst should be a subset of another in the same state of the module. Also, specified input bursts must be non-empty.

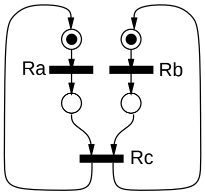


Figure 4.2: Example of Pulse-Burst Operation: JOIN Module specification.

Figure 4.2 shows an I-NET for a JOIN Module (described below). The JOIN module has two inputs (R_a and R_b) and one output (R_c). The JOIN operates as follows: it produces an output pulse only after it has received a pulse on each input. The key issue is that there are no timing conditions between the input pulses: they arrive concurrently. The inputs can come from completely unrelated, un-synchronized sources. Clearly, this behavior is not possible in traditional pulse-mode operation, which only allows a single pulse in its active state at any given time.

In this case, the input pulse burst is R_aR_b, meaning that these two pulses can arrive in any order and at any time. The JOIN module will produce an output pulse burst R_c after the input pulse burst has been received.

4.1.3 Pulse-Mode Implementation Issues

The use of pulses introduces issues that must be dealt with in order to successfully implement circuits. In this section we review some of these important issues.

In pulse-mode handshaking, a pulse is used to represent a single handshake event. Unfortunately, CMOS technology is level-driven and a pulse cannot be considered an atomic event. The situation would be different if we were considering a pulse-driven technology, like RSFQ [43], in which a pulse can be treated as an atomic event.

In our case, a pulse consists of two transitions: one from 0 to 1, called the rising edge, and one from 1 to 0, called the falling edge. The time between the rising and falling edges is called the *pulse width*.

Pulse Width and Separation

Given that a pulse actually consists of 2 edges (events), a first important observation is that circuits cannot be expected to respond properly to pulses of arbitrarily short width. The circuit has to have enough time to respond to the first event, the rising edge, before the second one is produced.

A minimum pulse width requirement must be met to guarantee correct operation. This requirement is analogous to the minimum clock pulse width requirement in synchronous systems. This minimum value depends on the actual technology used.

A similar requirement may arise for a minimum separation between pulses on the same wire. If two pulses are too close to each other, the system may start to respond to the falling edge of the first pulse and, before it is ready, the rising edge of the second one will arrive, possibly causing erroneous operation.

Single-Edge and Dual-Edge Modes

Pulse-mode modules can operate in two different modes: (*i*) The module can "react" to only one edge of the input pulse (usually the rising edge), ignoring the other ("single-edge mode"). Output pulses are generated as a consequence of the single meaningful event. Alternatively, (*ii*) the module can react to both edges ("dual-edge mode"). In this case, each input edge is used to generate the corresponding edge of the output pulses.

In single-edge mode, a single input transition causes an output pulse, *i.e.*, two output transitions. Clearly, this mode leads to multiple output change (MOC) circuits. The width of the output pulse depends on internal delays in the module. On the other hand, in dual-edge mode, each input change will produce a single output transition, and single output change (SOC) implementations can be used. In this case, the width of the output pulse will depend basically on the width of the input pulse.

A simple example can clarify the difference. A PULSER is a basic module that produces an output pulse whenever an input pulse arrives. Figure 4.3 shows

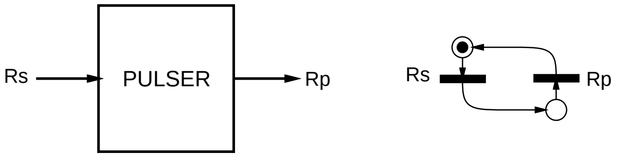


Figure 4.3: Pulse-Mode PULSER Module: Block Diagram and I-Net.

the block diagram of the module and an I-net that models its interface (channel) behavior.

R_s

	0	1
1	$\boxed{1}$,0	2 ,-
2	3 ,1	3 ,1
3	1 ,0	$\boxed{3}$,0

R_p

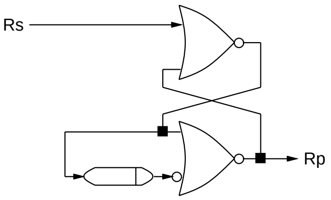


Figure 4.4: Single-Edge Implementation of the PULSER.

The single-edge flow table is shown in Figure 4.4. The module produces an output pulse as a response to the input rising edge (transitions 1-0 → 2-1 → 3-1). The falling edge generates no output change. The output pulse width is determined by the delay in getting from state 2 to state 3. A possible implementation is also shown in the figure, in which the delay element controls the width of the output pulse. On the other hand, it is clear that a single wire can be used to implement a PULSER in dual-edge mode.

The pulse-mode control modules introduced in the following section operate in dual-edge mode because, in general, dual-edge mode leads to simpler, faster circuits. We use single-edge mode mainly in modules that require control of the width of the output pulse, such as the PULSER described above. Single-Edge mode is also used in the design of modules that interface pulse-mode channels to transition signaling and 4-phase channels.

4.2 Pulse-mode Control Macromodules

The goal of the work presented in this section is the design of a basic set of efficient macromodules. Such a set has not been proposed before and is a basic requirement to design pulse-mode macromodular systems. In particular, we design pulse-mode modules that efficiently implement control operations such as sequencing, selection, iteration, concurrency control, resource sharing, and arbitration. Modules for interfacing pulse-mode circuits with traditional 2-phase and 4-phase circuits and to implement arbitration are presented in the following sections.

The modules have very efficient implementation using standard CMOS gates. Timing problems related to critical races, and combinational and essential hazards are constrained within individual modules and can be dealt with very effectively.

In all the control modules shown here there is a a signal R_s that is used to initiate the action of the module and an A_s signal that is generated to acknowledge R_s when the action is complete. The modules control one or more processes using a pair of signals R_p and A_p to communicate with each of them.

4.2.1 Sequencing

In macromodular systems, the most basic operation is the sequencing of processes. The 2-STEP module, also known as sequencer [94], is the basic sequencing element. This module transforms a single handshake cycle into a consecutive pair of such cycles. Figure 4.5 presents a block diagram of the module showing the input and output signals, and an I-net that models the interface behavior.

A flow table that provides a detailed specification of the module and the logic expressions for its proper realization are as follows:

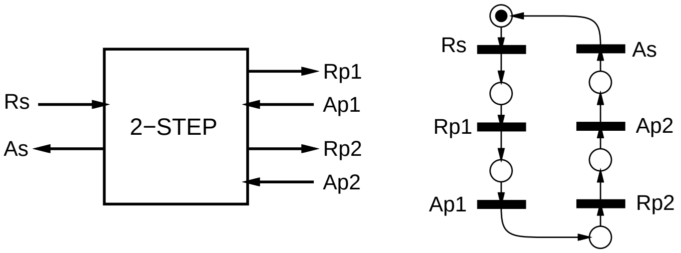


Figure 4.5: Pulse-Mode 2-STEP Module: Block Diagram and I-Net.

$R_s A_{p1} A_{p2}$

	000	001	011	010	110	111	101	100
1	$\boxed{1}$,000	$\boxed{1}$,001	$\boxed{1}$,011	$\boxed{1}$,010	$\boxed{1}$,110	$\boxed{1}$,111	- ,—	$\boxed{1}$,100

$R_{p1} R_{p2} A_s$

$$\boxed{R_{p1} = R_s, \quad R_{p2} = A_{p1}, \quad A_s = A_{p2}}$$

Clearly, the 2-STEP module requires *only wires* to be implemented and there are no timing problems related to hazards or critical races. A tree of $n-1$ 2-STEP modules can be used to control sequences of n processes.

4.2.2 Selection

The IF-ELSE module, also called *decision unit* [68], *Selector* [6], and *IF component* [94], is a basic decision-making element. The value of a boolean signal X is used to select between two processes to be activated. If $X = 1$, process P_1 will be activated through R_{p1}. If $X = 0$ then process P_2 will be activated through R_{p2}. Handshakes are completed through A_{p1} and A_{p2}, respectively.

There are two basic implementations of the IF-ELSE module, for dual-rail and single-rail encodings of the selection signal. The following sections introduce these implementations. All version of the IF-ELSE module have very efficient implementations.

Dual-Rail Selection Variable

Figure 4.6 shows the block diagram and interface behavior of the IF-ELSE module to be used with a dual-rail encoding of variable X. In this case, R_x is used to inquire the value of X. If $X = 1$ a pulse in X^1 will be returned. A pulse on X^0 represents $X = 0$.

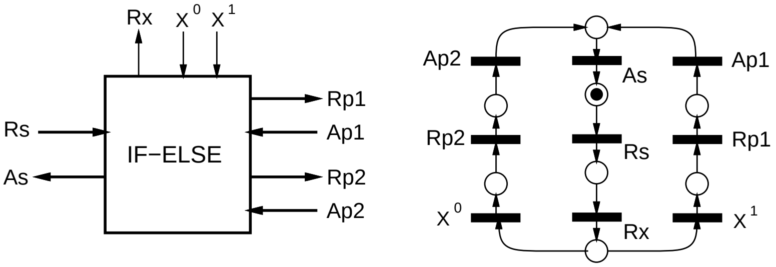


Figure 4.6: Pulse-Mode IF-ELSE Module: Block Diagram and I-Net.

Logic expressions to realize the module are:

$$\boxed{R_x = R_s, \quad R_{p1} = X^1, \quad R_{p2} = X^0, \quad A_s = A_{p1} + A_{p2}}$$

Figure 4.7 shows an implementation of the IF-ELSE module with dual-rail variable X for selection. As the 2-STEP, this module can be implemented very efficiently, using a single OR gate.

Single-Rail Selection Variable

When a single-rail selection variable is used, several options are available. In general, the single-rail selection signal X is a level signal, and its value directly indicates the selection value, *i.e.,* $X = 1$ is indicated by a stable '1' level (as opposed to a

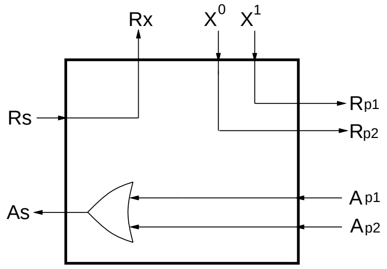


Figure 4.7: Pulse-Mode IF-ELSE Module Implementation.

pulse used in dual-rail implementations). Under certain conditions, the handshake operation to read the value of X can be eliminated, speeding up the overall operation. In this case, X must be stable before R_s arrives and must remain constant while the pulse is active. This set-up requirement for X is typical of single-rail systems and is called a *bundling constraint*.

An implementation of the IF-ELSE module for single-rail signal X, without the need for handshaking, is shown in Figure 4.8.

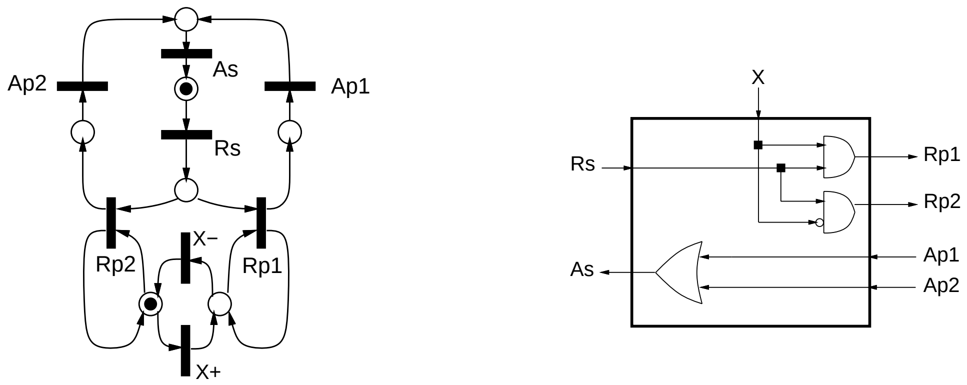


Figure 4.8: Pulse-Mode IF-ELSE Module (Single-Rail Variable X).

In the case of the IF-ELSE module, the use of a single-rail signal results in a slightly more complex module. The fact that dual-rail variables automatically provide the complement of the signal and a spacer code simplifies the logic.

Comparison with Traditional Modules

The efficient characteristics of the pulse-mode macromodules can be appreciated when compared to equivalent macromodules that use transition signaling and 4-phase handshaking. In this section we compare the transition signaling, 4-phase and pulse-mode implementations of the IF-ELSE module as a representative example. In all cases a single-rail selection signal is used.

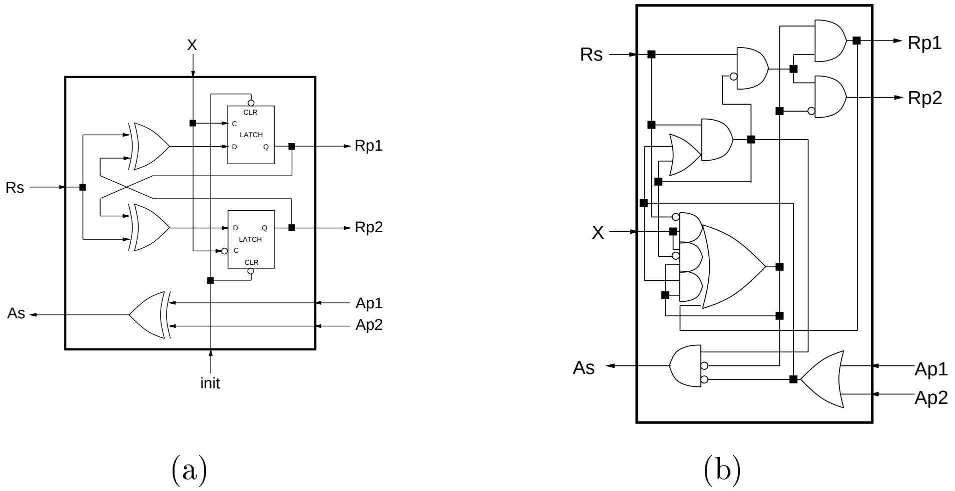


Figure 4.9: Transition Signaling and 4-Phase IF-ELSE Modules.

Figure 4.8 above shows the implementation of the IF-ELSE module using pulse-mode handshaking. The corresponding transition signaling and 4-phase implementations are shown in Figures 4.9(a) and 4.9(b), respectively.

The transition signaling module is based on the SELECTOR module by Brunvand [6]. An XOR was added to generate A_s from A_{p1} and A_{p2}. In this case, the complexity of the circuit eliminates the potential power and speed advantages of transition signaling. The 4-phase module corresponds to the IF-ELSE module by Unger[86]. This implementation also has less attractive characteristics than the pulse-mode module.

It is important to note that the three implementations above rely on timing assumptions related to changes in the value of X as a result of R_{p1} or R_{p2}. If X changes *too soon*, the circuits may not operate correctly. In that case a different version of the module, that temporarily latch the value of X, must be used.

IF Module

Sometimes a simpler component, the IF module, is needed. Instead of selecting among two processes as the IF-ELSE modules, the IF module selectively activates a process depending on the value of the select variable. Clearly, all versions of the IF-ELSE module can be transformed into version of the IF module by connecting R_{p2} directly to A_{p2}, *i.e.,* by using a null process as P_2. It is interesting to point out that the same IF module implementations are obtained if the complete design process, starting with a flow table, is followed, *i.e.,* no further optimization is possible.

4.2.3 Conditional Iteration

Iteration is also an important operation in macromodular systems: a process or sequence of processes is repeatedly executed a number of times. There are two forms of iteration: conditional and unconditional. This section presents macromodules that implement conditional iteration. Unconditional iteration will be introduced later.

Conditional iteration occurs when a process is repeatedly activated while a condition holds. Iteration stops when the condition becomes false. In general, the condition is represented by a boolean signal. Two different modules are used to implement conditional iteration, the WHILE and UNTIL modules. Both modules iterate conditionally, according to the value of a decision signal X. The decision signal can be encoded using dual-rail or single-rail. All versions of these two modules are very efficient. Only some of the versions are shown here.

The WHILE module implements the well known programming feature, in that, after R_s arrives, repeated executions of a process P are initiated (by turning on R_p) as long as the decision variable X is 1. If $X = 0$ at the start then P is not executed at all.

Figure 4.10 shows the block diagram and interface behavior of the WHILE module to be used with a dual-rail encoding of variable X.

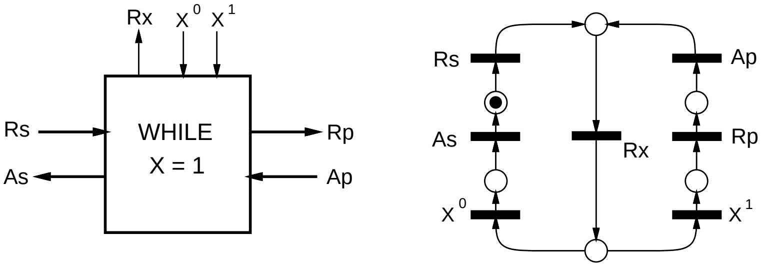


Figure 4.10: Pulse-Mode WHILE Module: Block Diagram and I-Net.

Logic expressions realizing this module are:

$$\boxed{R_x = R_s + A_p, \quad R_p = X^1, \quad A_s = X^0}$$

The UNTIL module also controls repeated executions of process P and is used in cases where it is desired to execute P at least once. To maintain the semantics of the WHILE, this module will iterate until the decision variable X is 0. The module is very similar to the WHILE. Figure 4.11 shows the block diagram and interface behavior of the UNTIL module to be used in environments where variable X is guaranteed to be stable while R_s is active, thus avoiding the X-handshake. Note that X is a level signal and both positive and negative transitions appear in the I-net.

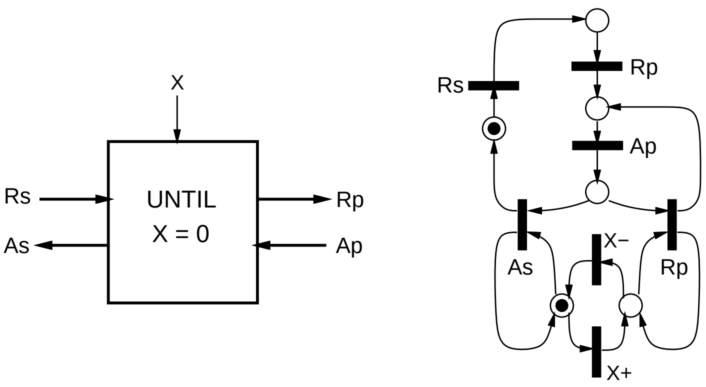


Figure 4.11: Pulse-Mode UNTIL Module: Block Diagram and I-Net.

Logic expressions for the proper realization of the module are as follows:

$$\boxed{R_p = R_s + A_p X, \quad A_s = A_p \overline{X}}$$

It is important to note that the WHILE and UNTIL module implementations shown above rely on the assumption that process P will take longer to complete than the width of R_s, so that pulses on different inputs to the OR gates will not overlap. This is almost always a reasonable assumption. In the rare case when P is a somewhat trivial (short) process, a PULSER that responds to the falling edge of A_p must be used.

4.2.4 Unconditional Iteration

Unconditional iteration occurs when a process or a sequence of processes is repeatedly activated a pre-defined number of times. Repetition is a special case of unconditinal iteration in which a process is activated continuously, *i.e.,* an infinite number of times.

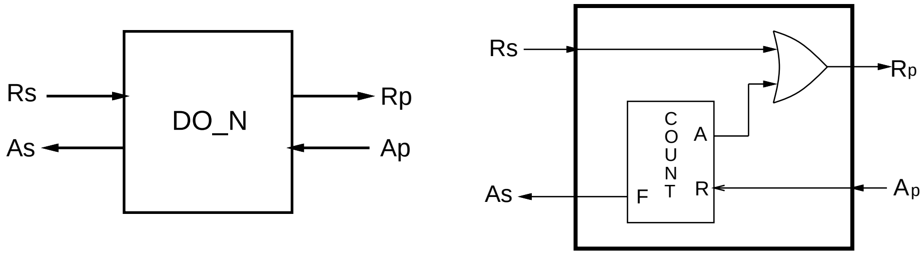


Figure 4.12: Pulse-Mode DO_N Module.

Unconditional iteration can be implemented using the DO_N module, illustrated in Figure 4.12. This module activates process P n consecutive times, using R_p. The module can be regarded as the composition of an UNTIL module and a counter, as shown in Figure 4.12. The operation of the counter will be examined in the following section.

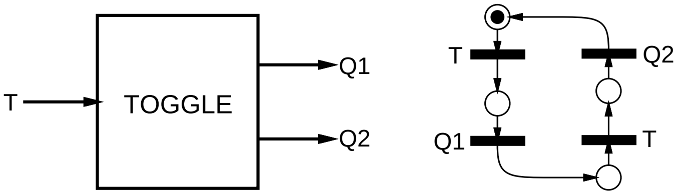
4.2.5 Pulse-Mode Counters

The DO_N module uses a counter to keep track of the number of iterations. This section introduces two basic implementations of pulse-mode counters.

As shown in Figure 4.12, a pulse-mode counter receives pulses in the R input and acknowledges the first n-1 pulses through output A and the nth pulse through F.

TOGGLE Element

The TOGGLE element can be used to implement the counter. This element, shown in Figure 4.13, accepts pulses through input T and acknowledges them alternative through outputs Q_1 and Q_2. The flow table for the TOGGLE is shown also in Figure 4.13.



	T	
	0	1
1	$\boxed{1}$,00	2 ,01
2	3 ,00	$\boxed{2}$,01
3	$\boxed{3}$,00	4 ,10
4	1 ,00	$\boxed{4}$,10
		Q_1Q_2

Figure 4.13: Pulse-Mode TOGGLE Element.

Logic expressions for the proper realization of the element are as follows:

$$\boxed{Y_1 = y_1T + y_2\overline{T} + y_1y_2, \quad Y_2 = \overline{y_1}T + y_2\overline{T} + \overline{y_1}y_2, \quad Q_1 = y_1T, \quad Q_2 = \overline{y_1}T}$$

This is a sequential circuit and there are timing considerations associated with its operation. The relative delays within the element must be controlled because there are essential hazards for every transition in the flow table. Note that, as has been the case so far, output pulses are generated by input pulses through a single gate. In this case y_1 and $\overline{y_1}$ change only as a result of $T\downarrow$ so that the pulses are not cut short. It is interesting to note that the TOGGLE can be implemented using two cross-coupled latches, an inverter, and a 1-2 decoder.

Pulse-Mode Ripple Counter

Figure 4.14 shows how to use 3 TOGGLE elements to implement an 8-COUNTER. It can be extended, by adding more stages, to build any 2^n-COUNTER. This implementation is equivalent to a transition signaling counter proposed by Ebergen and Peeters [24], and their designs for other count values can be adapted for pulse-mode operation also.

The response time $(R \rightarrow A)$ of this counter is not constant for all request. It depends on how many TOGGLE elements change state in each activation. For

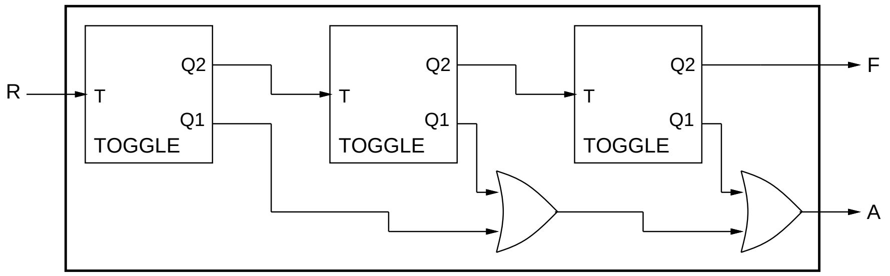


Figure 4.14: Pulse-Mode 8-COUNTER using TOGGLE Elements.

the 8-COUNTER, the best case is 1 TOGGLE while the worst case is 3 TOGGLE elements.

Alternative Pulse-Mode Counter

An alternative implementation of the same counter is shown in Figure 4.15. This implementation is more expensive in terms of area but has higher concurrent activity and, under reasonable timing assumptions, has a constant response time, similar to the best case of the previous implementation.

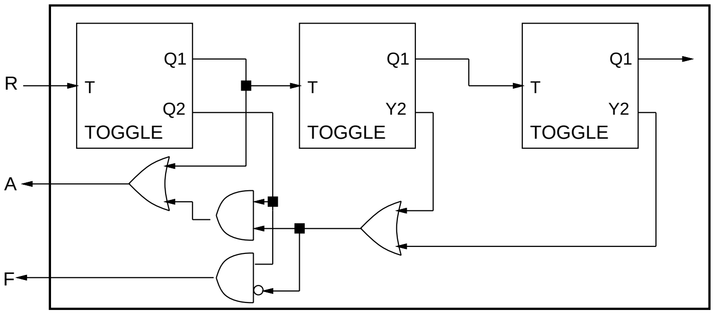


Figure 4.15: Alternative Pulse-Mode 8-COUNTER Implementation.

In this circuit, some of the toggle elements provide Y_2 as a level output. The analysis of the operation of the circuit is not simple but note that *not* all of Y_2 level outputs have to be stable when the input pulse arrives. They are all 0 *only* when the nth pulse arrives, producing a pulse on F.

4.2.6 Resource Sharing

The CALL module, also known as mixer [94], is a basic element that allows two different components to access the same resource. The components request access through R_1 and R_2. The CALL component "forwards" the request to the shared resource and routes the acknowledge back to the proper requester. Figure 4.16 shows a block diagram of the module showing the input and output signals and an I-net that models its interfaces.

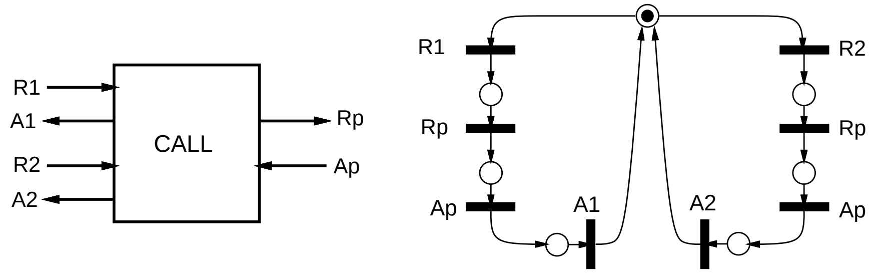


Figure 4.16: Pulse-Mode CALL Module: Block Diagram and I-Net.

A flow table that provides a detailed specification of the module and the logic expressions for its proper realization are as follows:

$R_1R_2A_p$

	000	001	011	010	110	111	101	100
1	$\boxed{1}$,000	$\boxed{1}$,010	- ,—	$\boxed{2}$,100	- ,—	- ,—	$\boxed{1}$,110	$\boxed{1}$,100
2	$\boxed{2}$,000	$\boxed{2}$,001	$\boxed{2}$,101	$\boxed{2}$,100	- ,—	- ,—	- ,—	$\boxed{1}$,100

$R_pA_1A_2$

$$\boxed{R_p = R_1 + R_2, \quad A_1 = A_p\overline{y}, \quad A_2 = A_p y, \quad Y = R_2 + \overline{R_1}y}$$

There are no essential hazards or races in this flow table. The module assumes that requests R_1 and R_2 are mutually exclusive for proper operation.

4.2.7 Concurrency Control

The FORK/JOIN module, also known as PAR element [94], implements the well known "fork/join" concurrent programming feature. This module, shown in Figure 4.17, is used to concurrently activate two processes, using R_{p1} and R_{p2}. The module waits until both processes have indicated completion (through A_{p1} and A_{p2}) to acknowledge the initial request.

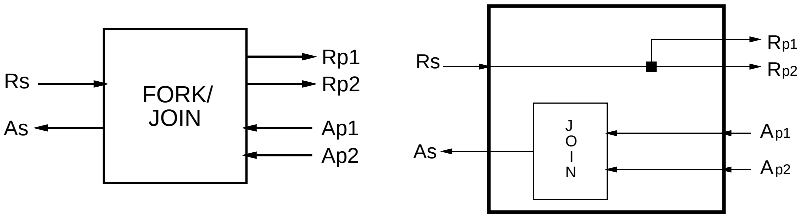
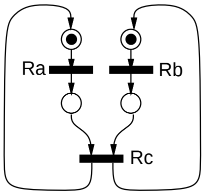


Figure 4.17: Pulse-Mode FORK/JOIN Module.

The JOIN element, also known as *rendezvous* [82], is a basic synchronization element and is used to implement the FORK/JOIN module, as shown in Figure 4.17. Figure 4.18 shows an I-net that models its interface (channel), and a flow table that provides a detailed specification of the module.

This element produces an output event when it has received events in both inputs.t The input pulses are allowed to arrive in any order and without any timing constraint, although a second pulse should not arrive at the same input until the



	R_aR_b			
	00	01	11	10
1	$\boxed{1}$,0	4 ,0	3 ,-	2 ,0
2	$\boxed{2}$,0	3 ,-	3 ,-	$\boxed{2}$,0
3	1 ,0	$\boxed{3}$,1	$\boxed{3}$,1	$\boxed{3}$,1
4	$\boxed{4}$,0	$\boxed{4}$,0	3 ,-	3 ,-

R_c

Figure 4.18: Pulse-Mode JOIN Element Specification.

output pulse has occurred.

Figure 4.19 shows a possible implementation of a pulse-mode 2-input JOIN module, using C elements. Alternative implementations can use SR flip-flops.

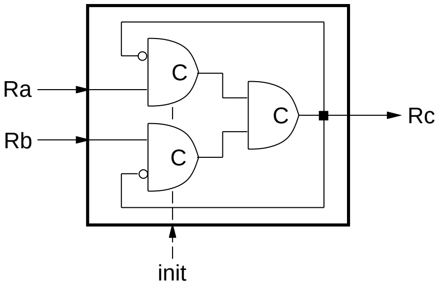


Figure 4.19: JOIN Element Implementation.

An interesting property of this circuit is that it "averages" the width of the input pulses, *i.e.,* the output pulse is not narrower than the narrowest input and not wider than the widest one.

The JOIN element shown in Figure 4.18 needs an initialization signal because the first two C elements have their inputs at opposite levels in the idle or reset state. In general, the outputs of these C elements should be initialized to 0 for proper operation. On the other hand, if one of the C elements is initialized to 1, a *primed JOIN* (pJOIN) element is obtained. The pJOIN acts as if it had initially received an event (pulse) in the primed input. This variation of the JOIN will be used in the control circuit of a micropipeline (see section 4.7).

4.3 Protocol Conversion

In order to interface pulse-mode macromodules with modules that use a different handshake protocol, conversion modules must be used. A complete set of conversion modules, that covers all protocols in both directions, must be designed. This section introduces converters thar interface pulse-mode to transition signaling and converters that interface pulse-mode systems to 4-phase.

4.3.1 Transition Signaling/Pulse-mode Conversion

The TS/PM module interfaces transition signaling (TS) and pulse-mode (PM) channels. In this case, TS is the input (passive) channel and PM is the output. Figure 4.20 presents a block diagram of the module showing the input and output signals, and an I-net that models the interface behavior.

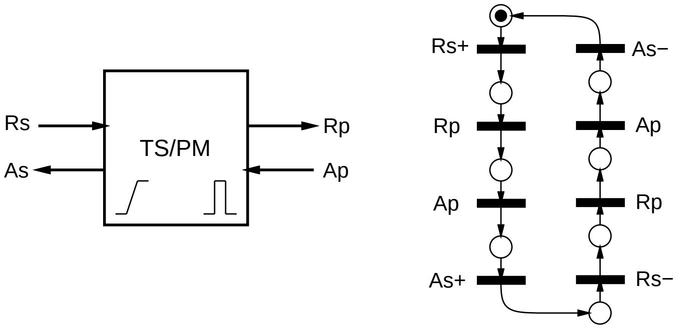


Figure 4.20: TS/PM Module: Block Diagram and I-Net.

This module must operate in single-edge mode, resulting in a MOC flow table, shown below. In this case, the width of the output pulse is determined by the time the circuit remains in transient states 1-10 (response to $R_s\uparrow$) and 3-00 (response to $R_s\downarrow$).

R_sA_p

	00	01	11	10
1	$\boxed{1}$,00	$\boxed{1}$,00	- ,–	2 ,10
2	- ,–	- ,–	3 ,01	$\boxed{2}$,00
3	4 ,11	- ,–	$\boxed{3}$,01	$\boxed{3}$,01
4	$\boxed{4}$,01	1 ,00	- ,–	- ,–

R_pA_s

The logic expressions for the proper realization of the table are as follows:

$$\boxed{Y_1 = R_sA_p + \overline{A_p}y_1 + R_sy_1, \quad A_s = Y_1, \quad Y_2 = \mathit{delayed}(R_s), \quad R_p = R_s \oplus Y_2}$$

The width of the output pulse is determined by the delay from R_s to Y_2. It is interesting to note that this circuit is “reversible”, *i.e.*, it can be used in the opposite direction (PM as the input and TS as the output). The terminals must be re-labeled ($R_p \leftrightarrow A_p$, $R_s \leftrightarrow A_s$).

4.3.2 4-Phase/Pulse-mode Conversion

The PM/4Φ module interfaces 4-phase (4Φ) and pulse-mode (PM) channels. 4Φ is the output channel and PM the input. Two different conversions, following narrow and broad protocols [6], are possible. Figure 4.21 presents a block diagram of the module showing the input and output signals, and an I-net that models the interface behavior of the narrow protocol.

This module also operates in single-edge mode, to produce the A_s output pulse as a response to the rising edge of A_p. A flow table is shown below. In this

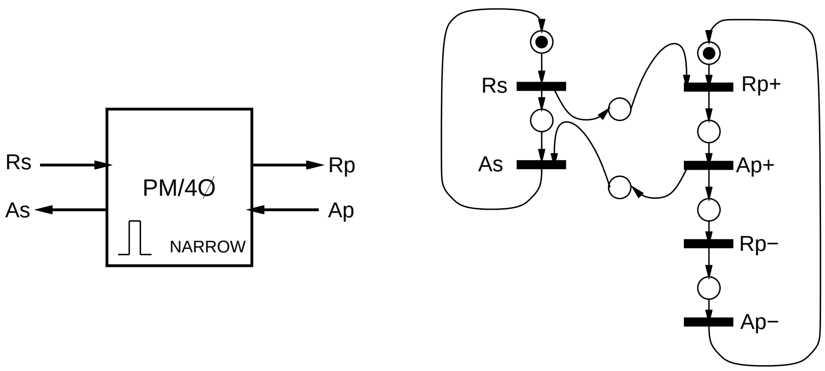


Figure 4.21: PM/4Φ Module: Block Diagram and I-Net.

case, the width of the output pulse is determined by the time the circuit remains in transient states 2-01 and 2-11.

R_sA_p

	00	01	11	10
1	$\boxed{1}$,00	- ,–	- ,–	2 ,10
2	2 ,10	3 ,01	3 ,01	$\boxed{2}$,10
3	1 ,00	$\boxed{3}$,00	$\boxed{3}$,00	$\boxed{3}$,00

R_pA_s

The logic expressions for the proper realization of the table are as follows:

$$\boxed{Y_1 = A_p + R_s y_1, \quad Y_2 = \overline{A_p} y_2 + R_s \overline{A_p}\,\overline{y_1}, \quad R_p = Y_2, \quad A_s = A_p delayed(\overline{Y_1})}$$

The width of the output pulse is determined by the time Y_1 is delayed in reaching A_s. If a broad protocol is needed, A_s must respond to the falling edge of A_p. The only change needed in the circuit is: $A_s = \overline{A_p} delayed(Y_1)$.

4.4 Arbitration

Asynchronous systems, including pulse-mode ones, require arbiters to guarantee mutual exclusion between two (or more) processes that can access a shared resource. Although many designs have been published, no pulse-mode arbiter has been presented before. This section introduces the design of a pulse-mode arbiter. This section also reviews basic background on arbiters and presents existing 4-phase and transition signaling arbiters.

4.4.1 Background

The design of asynchronous arbiters is an interesting and challenging problem. Plummer [77], and Pearce *et al.* [70] introduced arbiter designs over 20 years ago. A problem with these designs, as with the more recent by Calvo *et al.* [9] is that they assume that arbitration between two concurrent requests can be solved in a bounded amount of time. Chaney and Molnar [12], Marino [44], and Unger [90], among others, have shown that changes in asynchronous inputs to a circuit (like concurrent arbitration requests) can lead the circuit into metastability, an operation region in which circuit outputs can have an "intermediate" value between 0 and 1, or can oscillate between the two. Circuits can remain in the metastable state for an unbounded amount of time.

A robust arbiter must wait until the circuit leaves metastability before resolving any pending requests. Seitz [81, 79], Martin [45] and Unger [90], among others, have presented analog circuits that can be used to "filter" metastable states, *i.e.,* guarantee that no wrong output will be produced while the circuit is in a metastable state.

4.4.2 4-Phase Arbiters

Martin [45] introduced a mutual exclusion element (ME), shown in Figure 4.22. It is a basic two-way arbiter. The circuit guarantees that only one grant signal (g_1 or g_2) will be active at any time. If the circuit goes into metastability, both outputs will remain low until the metastable state is resolved.

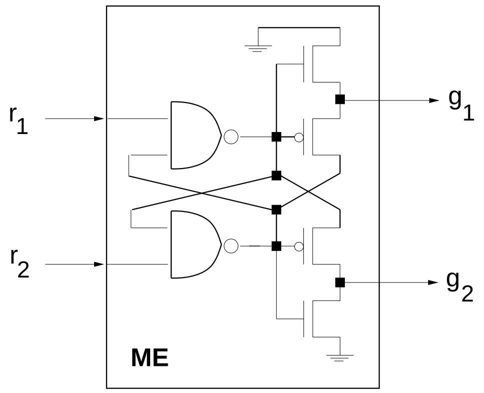


Figure 4.22: Mutual Exclusion Element.

Two types of 4-phase n-way arbiters have been the focus of most published work. Martin [47], Ebergen [23], and Ebergen *et al.* [22] have worked on "token ring" arbiters, formed by a linear chain of modules that "circulate" a special token, the authorization to grant requests. Seitz [81], Yakovlev *et al.* [101], and Josephs *et al.* [38] have presented designs for "tree" arbiters, formed by a tree of Tree Arbiter Elements (TAE). Figure 4.23(a) shows the TAE and its interface signals. A tree arbiter, constructed using TAEs and an ME is shown in Figure 4.23(b). A balanced tree gives equal priorities to all requestors.

The level-based approach of the 4-phase protocol fits nicely with the need to maintain a request, for an unknown period of time, until granted. Also, the four events in the protocol are meaningful: $req\uparrow$ indicates a request, $grant\uparrow$ indicates

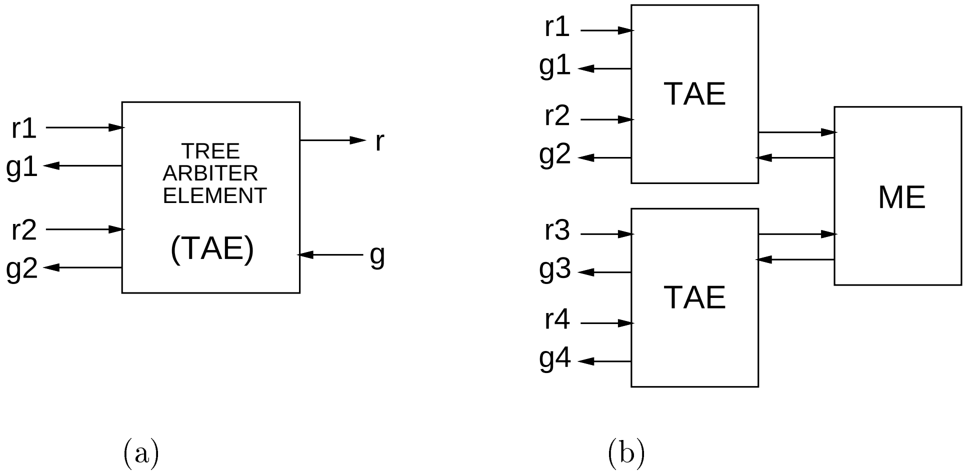


Figure 4.23: Arbiter Element and Tree Arbiter.

that the request has been granted, when the requestor completes the use of the shared resource, it issues *req*↓ to indicate it, and, finally, the arbiter issues *grant*↓ to cancel the grant.

4.4.3 Event-Based Arbiters

The transient nature of transitions and pulses requires a somewhat different approach. In [83], Sutherland introduced an *RGD* arbiter, which follows a 3-phase protocol: an event on *req* indicates a request, an event on *grant* indicates that the request has been granted, and, finally, the requestor produces an event on *done* to indicate that it has completed the use of the resource. It is not necessary to acknowledge the last event since the requestor is authorized to issue a new request immediately. This type of arbiter is better suited to both transition signaling and pulse-mode operation. Brunvand [6] showed that a 4-phase arbiter can be converted to a transition signaling *RGD* arbiter by adding two latches and an exclusive-OR gate to every channel.

4.4.4 Pulse-Mode Arbiter

The arbiters introduced above cannot be used in pulse-mode systems directly. In pulse-mode, the modules send pulses as requests. These pulses are lost if not granted immediately.

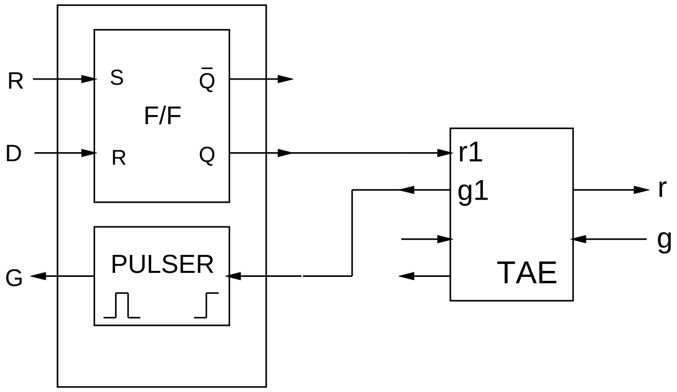


Figure 4.24: RGD Arbiter: Pulse-Mode Interface.

Figure 4.24 shows how a pulse-mode *RGD* arbiter can be constructed, based on a 4-phase tree arbiter (such as the one in Figure 4.23 above). Pulse-mode request (R), grant (G), and done (D) are connected through an interface (converter) module. A single SR flip-flop and a pulser are used to implement the interface module.

4.5 Pulse-Mode Control of Datapaths

This section examines how pulse-mode control macromodules are used to control datapaths. Both single-rail and dual-rail datapaths are considered. This section also examines how pulse-mode macromodules could be used to control datapaths implemented using pulse-driven logic.

4.5.1 Single-Rail Datapaths

Pulse-mode macromodules can easily be used to control single-rail datapaths, for several reasons: (*i*) The absence of a spacer or idle code fits nicely with the 2-phase nature of the pulse-mode handshake protocol, (*ii*) Pulse-mode handshaking allows the use of economical processing logic blocks like the ones used in synchronous systems and in existing single-rail systems since the pulses propagate through the bundled delays while the function block processes the data, (*iii*) the symmetric delay elements (bundled delays) used in traditional single-rail systems can also be used as bundled delays in pulse-mode systems. There is no need for asymmetric delays, used in 4-phase systems to reduce the return-to-zero phase, and (*iv*) The fact that a pulse is used to control the registers allows the use of level-based latches or edge-triggered flip-flops as storage elements. There is no need for expensive event-based latches or dual-edge-triggered flip-flops required in transition signaling datapaths or for interface circuitry needed in single-track implementations.

Datapath Operation

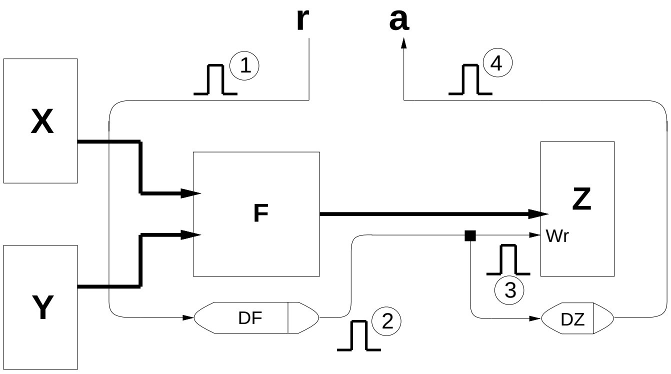


Figure 4.25: Pulse-Mode Control of Single-Rail Datapath.

Figure 4.25 illustrates, schematically, how a single-rail datapath stage can operate under pulse-mode control. The operation is as follows. A request pulse $\boxed{1}$ activates the stage. The pulse propagates through the matched delay (DF) $\boxed{2}$ while computation in F takes place. When the computation completes, the result is sent to the latch with the pulse as a write request $\boxed{3}$. While the data is written to the latch the pulse propagates through DZ, to be sent, finally, as the acknowledge pulse $\boxed{4}$ that signals completion of processing.

The critical aspect of the interaction with the datapath is the use of the handshake pulses to control the activity of the latches. The latches are transparent only during the time when the *write request* pulse is active. If the pulse is too short, the latch may fail to store the new data (see, for example, [71]). It is important to enforce design rules that establish a minimum pulse width that guarantees correct operation of the latches. In some cases, PULSER, which locally controls the width of the output pulse, can be used to drive the latches. The PULSER itself cannot be expected to respond to arbitrarily narrow pulses, but it is less sensitive than the latch to pulse width variations.

4.5.2 Dual-Rail Datapaths

Pulse-mode can also be used to implement dual-rail datapaths. However, the use of dual-rail code and latches usually requires a spacer or idle code between consecutive data values. This spacer state fits nicely with the return-to-zero phase of the 4-phase protocol but is not easily mapped to 2-phase handshaking.

Pulse-Mode control of dual-rail datapaths requires the use of interface macromodules (PM/4Φ) to convert the pulse-mode signals to standard 4-phase level signals. Two different conventions can be used to convert pulse-mode handshake signals to 4-phase ones: *broad* and *narrow* conventions [6]. The broad convention is roughly equivalent to a 4-phase sequential protocol while the narrow convention implements a concurrent one (see [74]). The use of the narrow convention requires the use of interlock mechanisms described in Section 3.4.4 to avoid data hazards.

Datapath Operation

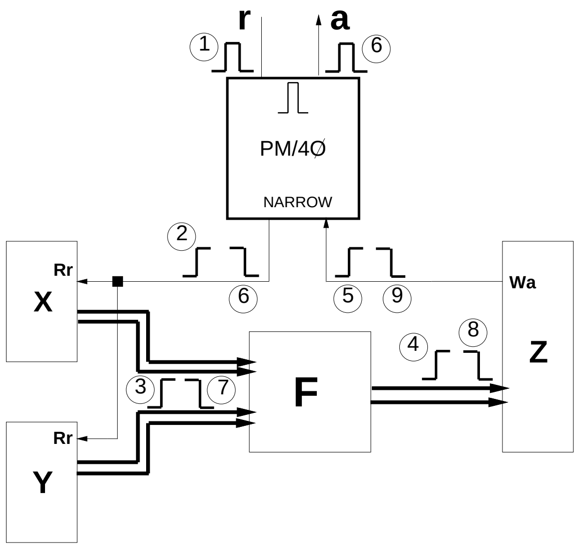


Figure 4.26: Pulse-Mode Interface to Dual-Rail Datapath.

The operation of a dual-rail stage with a narrow converter is illustrated in Figure 4.26. A request pulse feeds the converter $\boxed{1}$, which generates a request rising transition $\boxed{2}$ to activate the stage. The latches receive the request and send dual-rail data to the logic $\boxed{3}$. The logic computes the result and sends dual-rail data to the output latch $\boxed{4}$. When the latch has stored the data, it generates an acknowledge rising transition $\boxed{5}$ that feeds the converter. The converter generates, concurrently, an acknowledge pulse $\boxed{6}$ and a falling transition in the read request $\boxed{6}$. Data returns to its idle (spacer) value $\boxed{7}$ & $\boxed{8}$. This causes the write acknowledge of the latch to fall $\boxed{9}$, completing the operation of the stage. If a broad convention is used, the only difference is that the acknowledge pulse is generated by the converter as a response to the falling write acknowledge $\boxed{9}$.

Alternative Dual-Rail Encoding

A different approach to pulse-mode control of dual-rail datapath is the use of an alternative dual-rail code that requires no spacer or idle code, *i.e.,* is directly compatible with pulse-mode handshaking components. Dean *et al.* [18] introduced Level-Encoded Dual-Rail (LEDR) code which is a dual-rail code that requires no spacer. In this case, a 2-phase protocol can be used directly, without interface circuitry. This code requires the use of specilized latches or storage elements, and also requires specially designed logic circuits. This alternative has to be further explore to determine if it produces good results and is cost-effective.

4.5.3 Datapaths Implemented Using Pulse-Driven Logic

Both single-rail and dual-rail datapaths use levels to represent data. An alternative approach is the use of pulse-driven logic to implement the datapath itself. This section examines a pulse-driven code and the use of pulse-mode control of a datapath implemented using such a code.

A pulsed code uses two wires per data bit, and data is transmitted using pulses instead of levels. A pulse in the first wire represents a value of '1' and a pulse in the other wire represents a '0' value. There is no special spacer code, only the absence of pulses. This fits nicely with pulse-mode handshaking.

Self-resetting CMOS (SRCMOS) [13] devices can be used to implement pulse-driven logic. This precharged, unipolar switching logic family is similar to Domino logic, the main difference being that SRCMOS precharging is not governed by a global clock. Instead, reset signals are generated locally. The self-resetting nature of the devices results in output signals being pulses and not levels. SRCMOS, being a precharged logic, responds only to rising transitions of the inputs, operating in a mode similar to the single-edge mode presented earlier. The width of the output pulses is controlled by the local reset timing.

Datapath Operation

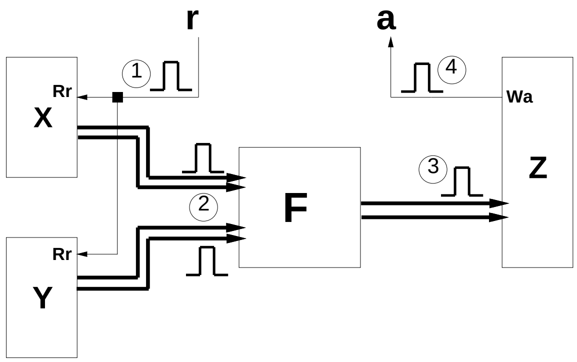


Figure 4.27: Pulse-Driven Dual-Rail Datapath.

Figure 4.27 illustrates the operation of a pulse-driven stage under pulse-mode control. A request pulse activates the stage $\boxed{1}$. This causes the latches to send pulsed dual-rail data to the logic $\boxed{2}$. F processes the data and generates pulse-mode results that are sent to the output latch. Finally, the output latch generates the acknowledge pulse $\boxed{4}$ that indicates completion of stage processing.

If pulse-driven logic is used, pulses will be used to drive the latches and the width of the pulses becomes critical for the correct operation of the system. Similar considerations as those for single-rail systems also apply in this case. Haring *et al.* [34] introduced a pulse-driven register for self resetting circuits that can be used as illustrated above. The correct operation of the pulse-driven logic requires that several timing constraint be satisfied [60]: (*i*) Input pulses must have a minimum width, and (*ii*) input pulses that are to act together must overlap by an amount of time that depends on the size and topology of the devices in the logic.

4.6 System Optimization

A potential weakness of the use of macromodules is the presence of inefficiencies in the design due to the use of pre-designed modules. In some cases, it is possible to examine the macromodule diagram and optimize it in a number of ways. This section introduces two techniques than can be used to optimize pulse-mode macromodular systems.

The process of examining a macromodular system and applying techniques to opimize it according to a given metric, sometimes called peephole optimization, has been studied by several researchers. These optimizations are very similar to the peephole optimizations used by compilers to optimize code.

Brunvand [6], van Berkel [96], and Peeters [71] have introduced optimizations that consist of structural transformations that maintain the functionality of the circuit but optimize it according to a desired metric. Example of these optimizations are: null module elimination, call element reduction, channel reduction, use of multi-way components instead of simple compositions of 2-way ones. A different approach is taken by Gopalakrishnan *at al.* [33] and Kolks *et al.* [41]. They optimize the macromodular system by re-synthesizing sections of the control path, using automatic tools.

In many cases, these optimizations are independent of the actual handshake protocol used to implement the system and, therefore, can also be applied to pulse-mode macromodular systems.

A different type of optimization, based on timing assumptions, is introduced here and usually obtains good results. In particular, reasonable timing assumptions about the relative completion time of different processes may lead to the simplification of a module, optimizing the performance of the system. Two basic optimizations are introduced in the following sections: JOIN elimination and module substitution.

4.6.1 JOIN elimination

Synchronizing two processes is an expensive operation, both in terms of performance and area. One of the most effective ways to improve the performance of a system is to eliminate synchronization modules. A FORK/JOIN macromodule is commonly used to synchronize the operation of two or more processes that must execute concurrently. The JOIN element is used to “wait” for the completion of all processes before generating its completion signal.

Figure 4.28(a) shows the typical scenario in which two processes execute concurrently. The two processes are activated together by the same signal (forked as request to both processes) and a JOIN element is used to synchronize the completion of the two processes. The JOIN element is unavoidable if the operation of the circuit must be guaranteed correct assuming unbounded delays in the two processes. This results in a very robust circuit. Unfortunately, the JOIN is an expensive element, both in performance and area. We can eliminate it if certain timing conditions are met.

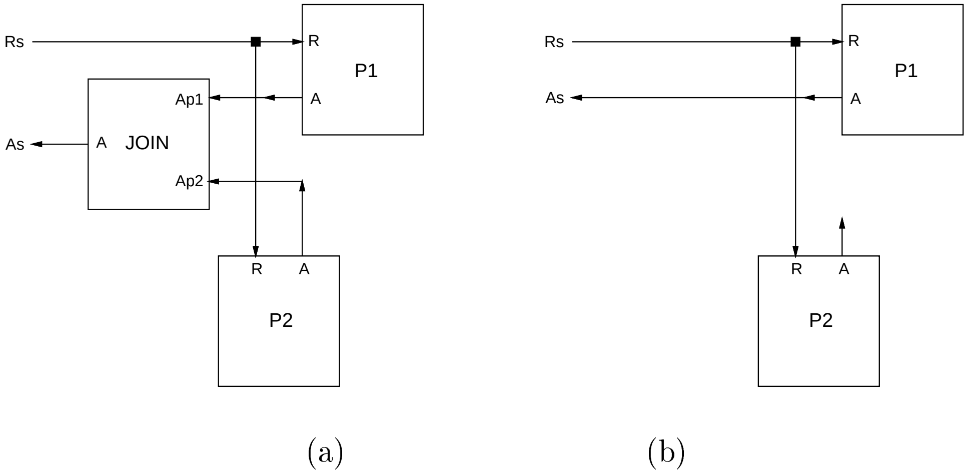


Figure 4.28: Optimization: JOIN Elimination.

If one of the processes, say P_1, is guaranteed to take longer to complete than the other (P_2), under reasonable timing assumptions, then the JOIN element can be eliminated. The completion signal of the "slow" process can be used to indicate completion of the two processes and the structure of Figure 4.28(b) can be used. This improves the performance of the system. Clearly, this substitution is not possible under the unbounded (gate and wire) delay model. On the other hand, do we really want to use a component with unbounded response time?

4.6.2 Module Substitution

Some macromodules have several "versions", which are used in different environments. For example, different IF-ELSE macromodules are used when the selection signal is encoded in single-rail or dual-rail. A second problem that can affect the performance of a system is the use of a complex version of a macromodule, which operates correctly under strict delay models, when a simpler module could be used if reasonable timing assumption are met.

Module substitution is applied to identify changes in the system that allow the use of a simpler version of a macromodule, with the same functionality. The substitution works correctly only if the required timing assumptions are met. For example, Figure 4.29 shows a segment of a macromodule diagram. Variable X is the selection variable for the IF-ELSE module. X is reset by a pulse sent from the IF-ELSE module. In this situation, an IF-ELSE module that allows X and $\overline{X}$ to change while its output pulse is still active must be used.

A simpler version of the IF-ELSE module could be used if X is guaranteed to remain stable while the IF-ELSE output pulse is active. To use this version of the module in our system, variable X cannot be reset directly by the IF-ELSE. A different process must do it. As shown in Figure 4.29 above, the resetting of X is done concurrently with two other processes. If process P_1 is guaranteed to take

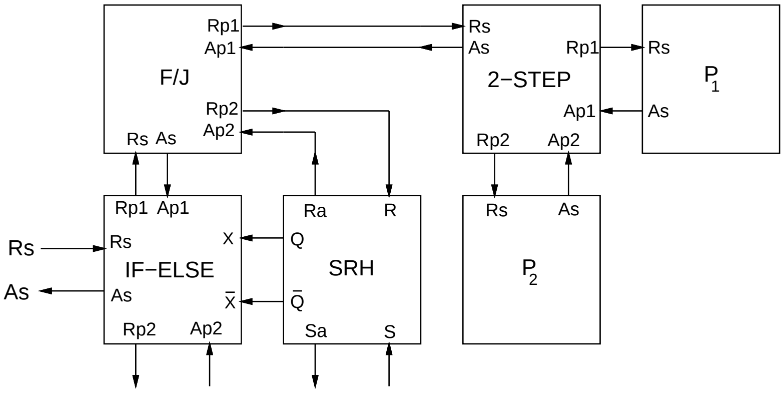


Figure 4.29: Optimization: Module Substitution.

longer to complete than the width of the R_s pulse (a reasonable assumption in most cases), then the resetting of X can be done concurrently with P_2, thus X and $\overline{X}$ will remain stable and the simpler IF-ELSE module can be used safely.

The combination of these two optimizations usually results in a more efficient implementation of a system.

4.7 Pulse-Mode Systems: Examples

In this section we present two macromodular designs using pulse-mode handshaking: (*i*) an asynchronous micropipeline, and (*ii*) an add-and-shift multiplier. These designs illustrate how pulse-mode components can be used to assemble a large system.

4.7.1 Micropipelines

Pipelines are a common way to organize datapath sections to increase throughput, at a relatively low cost in area and latency. Sutherland [83] introduced micropipelines as a simple and elegant way to implement asynchronous pipelines.

Although Sutherland proposed the use of 2-phase handshaking and event-driven ("capture-pass") latches, many different implementations have been presented [16, 30, 105], with varying results. In general, 2-phase micropipelines have to use expensive latches or flip-flops while 4-phase micropipelines have higher controller costs and less concurrent activity.

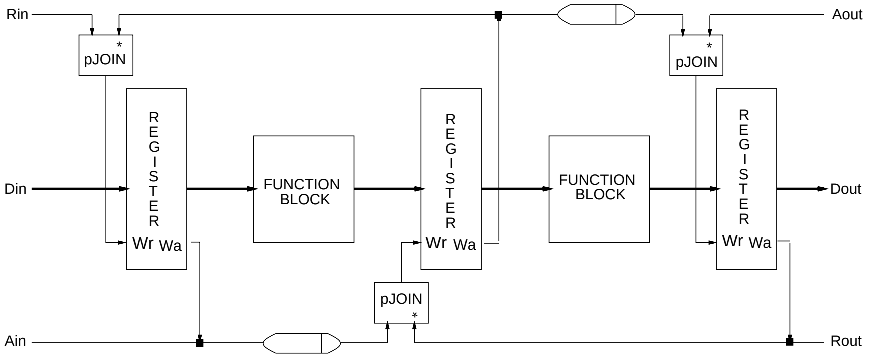


Figure 4.30: Pulse-Mode Micropipeline.

Figure 4.30 shows schematically how a micropipeline can be organized using pulse-mode macromodules to control it. The output pulses of the pJOIN modules are used to control the operation of the registers. This pulse-mode implementation has several desirable features: (*i*) Use of standard latches or edge-triggered flip-flops, (*ii*) all stages of the micropipeline can compute concurrently, and (*iii*) all registers of the micropipeline can store valid data concurrently.

As in other cases, there are robustness/performance trade-offs in this circuit related to set-up, hold, and propagation times of the storage elements, the width of the control pulses, and the bundled delay elements.

4.7.2 Iterative Multiplier

Multiplication is an important operation in both general-purpose and dedicated digital signal processors. It must be implemented carefully and efficiently. In this section we show how to implement an add-and-shift multiplier using pulse-mode handshaking.

The classic computer algorithm for multiplying two n-bit binary numbers works as follows: one operand is stored in a register called $MPCND$, and the other is stored in the MQ register. The accumulator (ACC) is connected as an extension ("concatenated") to the most significant end of MQ. ACC is initially cleared to 0. At each step, if the current least significant bit of MQ (MQ_0) is 1, $MPCND$ is added to ACC and the combined register ACC;MQ is shifted one position to the right. This process is repeated n times. At the end, the product is stored as a 2-n bit number spanning ACC and MQ. This process is described in the specification shown in Figure 4.31.

```
initialize registers ($MPCND$, $MQ$, $ACC$);
DO_N
  IF ($MQ_0 = 1$)
    FORK
      $ACC = ACC + MPCND$
      SHIFT RIGHT ($MQ$)
    JOIN
  ELSE
    SHIFT RIGHT ($ACC$;$MQ$)
```

Figure 4.31: Iterative Multiplier Specification.

Figure 4.32 shows a block diagram of the datapath used to implement the pulse-mode multiplier. A simple optimization has been performed by directly shifting right the outputs of the adder so that ACC need not be shifted after the addition. This allows MQ to be shifted concurrently with the add operation.

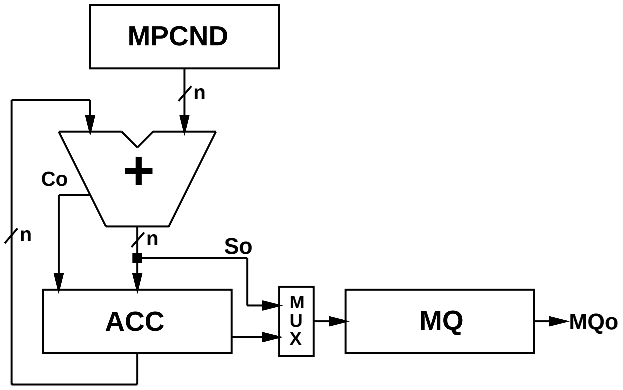


Figure 4.32: Add-and-Shift Multiplier Datapath.

Basic Implementation

Using the macromodules presented above, the specification can be directly translated into a simple macromodular diagram. The block diagram of the control logic of the multiplier is shown in Figure 4.33. The logic clearly follows from the description above, where a module directly implements each feature. Every ";" in the above specification maps into a 2-STEP module, which are used for sequencing.

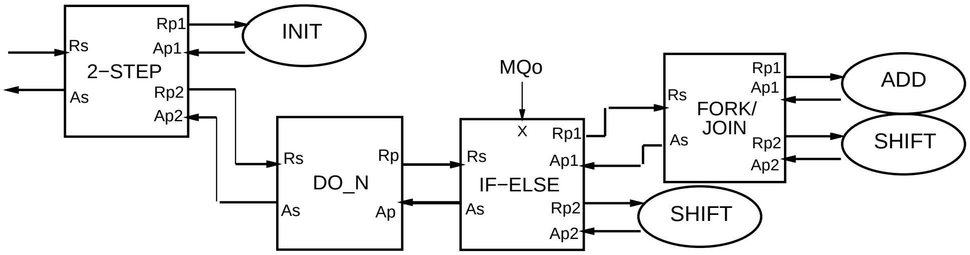


Figure 4.33: Add-and-shift Multiplier Control Logic.

The control overhead is kept to a minimum. The 2-STEP corresponds to wires only. The DO_N represents a counter and, if the concurrent version shown in Figure 4.15 above is used, the response time is that of a single TOGGLE element.

The IF-ELSE is, essentially, one basic gate. The FORK/JOIN is the module that introduces the largest overhead.

Optimized Implementation

The control path of the multiplier can be optimized by the application of JOIN elimination and module substitution. In particular, given that the SHIFT RIGHT operation for register MQ takes a constant time and is faster than the ADD operation, which is data-dependent, the FORK/JOIN module can be eliminated and the completion signal from the ADD operation used directly to feed the IF-ELSE module. Also, given that the datapath processing takes longer than the width of the R_S pulse in the IF-ELSE module, its simpler version can be used.

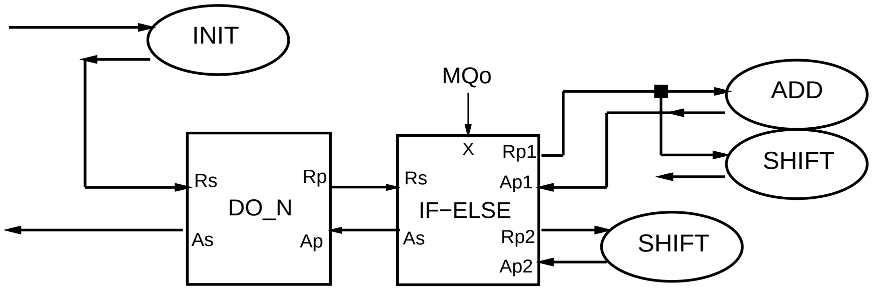


Figure 4.34: Optimized Multiplier Control.

The optimized control path of the multiplier is shown in Figure 4.34. Clearly, the optimizations result in reduced control overhead and better overall performance of the system.

4.8 Conclusions

In this chapter, pulse-mode macromodular systems were introduced. The goal of the work is to produce robust, performance–competitive asynchronous systems.

In particular, the following contributions were presented. First, the use of pulses to implement an efficient handshaking protocol. Second, the introduction of Pulse-Burst Operation (PBO), a more concurrent form of pulse-mode operation. Third, the design of a large set of pulse-mode macromodules that can be used to build cost-effective asynchronous macromodular systems. The set includes control macromodules as well as arbiters and converter modules to interface pulse-mode to traditional 2-phase and 4-phase circuits. Finally, the design of a pulse-mode micropipeline and a pulse-mode add-and-shift multiplier, to illustrate that pulse-mode macromodules can be used to assemble relevant, large systems.

Pulse-Mode handshaking combines the conceptual simplicity of the 2-phase protocol (only two events per handshake) with the level based approach of 4-phase handshaking. Pulse-mode macromodules are efficiently implemented using standard gates in CMOS technology.

The modules are of an order of complexity that is very well suited to design by flow-table based techniques. Timing problems corresponding to critical races, combinational hazards and essential hazards are constrained within individual modules and can be dealt with very effectively.

Pulse-mode control modules fit nicely with single-rail datapaths. They can also be used with dual-rail datapaths but adjustments must be made. An alternative to be further explored is the use of pulse-driven logic, like SRCMOS, to implement the combinational logic in the datapaths. In all cases, the critical aspect of the control/datapath interaction is the use of handshake pulses to control the activity of the latches.

Although further work is needed, early results suggest that the use of pulse-mode macromodules can keep control overhead low, without introducing complex timing considerations. Several issues have yet to be investigated such as how to test pulse-mode systems or the effect on pulses of different circuit factors including, for example, supply voltage and noise. None of them seems to represent a major hurdle that would prevent the application of pulse-mode macromodular systems.

Chapter 5

Design of a Macromodular Packet Switch

If asynchronous designs are to be widely used, it is critical to identify application domains where asynchronous techniques are of practical interest and to demonstrate their potential advantages using real designs. The zero-overhead divider [100], the high-performance cache controller [67], the DCC error corrector [92], the asynchronous differential equation solver [107], and the different versions of the Amulet processor [29] constitute major steps in this direction.

In this chapter, the design of an asynchronous packet switch is presented. A packet switch was chosen as a case study for several reasons: (*i*) It is, clearly, a relevant application, (*ii*) it is complex enough to show that the macromodular approach can be used to design large-scale systems, and (*iii*) it is a control-dominated system in which control overhead has a large impact on performance, stressing the need for an efficient design.

This chapter is organized as follows. Section 5.1 presents basic concepts related to packet switching in general and ATM in particular. In Section 5.2, the architecture of the macromodular packet switch is introduced, including a description of its key features and trade-offs. The complete design of the switch is introduced in Section 5.3. This section presents detailed descriptions of the different blocks of the switch including the Input and Output processors, Buffers, and the asynchronous bus. Section 5.4 presents the results of the simulation of the switch. This section also includes a comparison with a synchronous system. Finally, Section 5.5 presents conclusions.

5.1 Introduction

Recent years have witnessed an increased interest in high-speed networks supporting efficient protocols, such as ATM (Asynchronous Transfer Mode). These networks require high-performance switches to achieve the desired levels of increased bandwidth. The main goal of the work presented here is to demonstrate that a real system can be designed and built efficiently as a macromodular system. In particular, pulse-mode macromodules are used to design a highly modular, flexible, high-performance ATM switching component.

The basic idea behind ATM is to transmit small, fixed-size packets, called cells. The cells are 53 bytes long, of which 5 bytes are a header that contains destination and error checking information, and the remaining 48 bytes are data. ATM is a connection-oriented protocol, so a transmission path is established before actual data is sent. The protocol guarantees that cells arrive at their destination in the same order in which they were sent. The primary data rate in ATM networks is 155 Mbps, enough to transmit high-definition TV over the net. A second rate of 622 Mbps is also used to support transmission of four 155 Mbps channels.

ATM networks are organized like traditional Wide Area Networks (WANs), with links and switches. All links are point-to-point, and run between a computer and a switch, or between two switches. A general model of an ATM switch is simple: the switch has some number of input ports and some number of output ports. Cells arrive asynchronously on the input ports, each at its own speed. The switch examines each cell header, determines the required output port, updates the header information, and delivers the cells to proper output. Finally, cells leave the switch through the output ports, each operating at its own speed also.

One of the critical problems in ATM switch design is the need to buffer cells inside the switch. When two incoming cells have the same output port as destination, one of them will be directed to that port and the other cell will have to be buffered inside the switch and transmitted when the port is free. Many switch designs have been described in the literature proposing different places to locate the buffers: (*i*) at the input ports, (*ii*) at the output ports, (*iii*) associated with the physical switching elements (when a switching matrix is used), or (*iv*) in a shared memory, accessible by all input and output ports. The problems and benefits of each option have been thoroughly discussed elsewhere [85, 36, 17, 21].

Independently of the buffer placement, a second problem arises. What happens when a cell arrives at an input port and there is no free buffer space to use? In ATM, switches simply drop the cell. The protocol, at a higher level, will identify the situation and arrange for cell retransmission. This is a costly operation, so cell dropping must be avoided as much as possible.

The rest of the chapter presents the architecture and design of an asynchronous fast packet switch. The macromodular design results in a simple, efficient system. The parameters used to evaluate the design are the latency of a packet traversing the switch and the maximum throughput that each input and output port can handle. Flexibility and modularity are also considered important, so that the switch can be "tailored" to the needs of particular applications. Area and energy consumption are secondary concerns.

5.2 Switch Architecture

High-performance packet switching has been a very active area of research for several years. The introduction of ATM, with small fixed-size cells, had a large impact on switching architectures. A large number of ATM switch designs has been proposed and there are many commercial ATM switches available. Research is still very active, with higher performance objectives.

Synchronous designers have proposed a number of alternative switching architectures for packet and ATM switching, such as the Knockout [104] and the Coprin switches and Prelude architecture [99], the Switch-on-a-chip [20, 42], Tiny-Tera [51] and several others [53, 76]. Due to the asynchronous nature of the inputs and the internal synchronous operation, most of these switches need to synchronize the cells internally and operate in fixed-length "cell cycles". The need for internal synchronization is a major problem in synchronous implementations and has a large impact on their performance.

Research on asynchronous packet switches has not been as active. Although not entirely the same application, several asynchronous packet or message routing chips for multiprocessors have been designed and built. Examples are the Torus Routing Chip [15] and the Mesh Routing Chip [28]. In [37], Josephs *et al.* presented an interesting high-level design of an asynchronous packet-routing chip. An algebraic formalism is used to specify its operation and for synthesis and verification. Yantchev and Nedelchev [103] also presented a packet switching device, designed as a delay-insensitive circuit. Recently, an asynchronous ATM high-speed switch was introduced by Budde *et al.* [8]. This is an interesting implementation which combines clocked modules that operate asynchronously. Five different clocks were used in the design.

The design presented in this chapter is an asynchronous packet switch. The actual design is geared to ATM switching, although nothing in the architecture prevents it from being used in other packet switching environments. Figure 5.1 shows the basic structure of the switch. Every input port has an associated input adapter and every output port has a corresponding output adapter. The actual cell switching takes place in the switch fabric. All input and output adapters can operate concurrently, at their own speed. Cells must traverse the fabric as soon and as fast as possible.

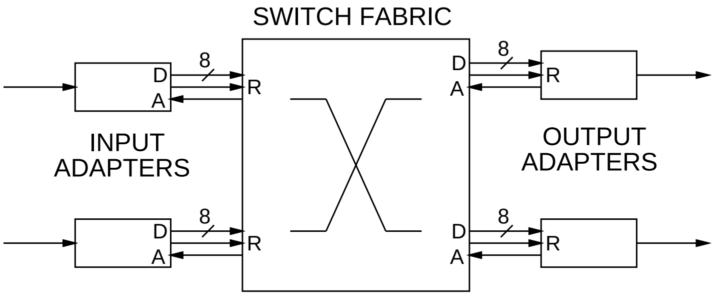


Figure 5.1: Switch Structure.

The switching process is as follows: Cells arrive at the inputs asynchronously, *i. e.* either a clock line is associated with every data channel or clock and data are encoded together in the bit-serial input stream. The input adapters are responsible for decoding the data from the input lines. The figure also shows that the input adapters send byte-sized data to the fabric. The switch has to be able to handle several concurrent links and must analyze incoming data, modify it and send it to the correct output. For this reason, the internal switch bandwidth must be larger than the external link bandwidth. This is accomplished by transforming the bit-serial input links into wider internal paths.

The input adapters also manage header-related tasks. When a cell is received, its Header Error Check (HEC) field is checked. If an error is found in the header the cell is dropped. If there is no error, the rest of the header is examined to determine the destination output port and is updated with new information. The input adapter prepends an internal self-routing header (SRH) to the cell. The SRH is used by the switch fabric to route the cell to the proper output port without having to examine the cell header.

Finally, the input adapters transmit the cells to the switch fabric. The communication between the input adapters and the fabric is synchronized using pulse-mode handshaking. The handshaking works also as a back-pressure mechanism to allow the fabric to refuse cells when its buffering space is full.

The function of the output adapters is simpler than that of the input ones. The fabric transmits cells to the output adapters using byte-sized data channels. The channels use pulse-mode handshaking for synchronization. The output adapters transform the parallel data into bit-serial streams, at the rate required by the external output link. When the output adapters are idle, *i.e.,* no cells arrive from the fabric, they generate empty cells to fill the unused bandwidth. These cells are automatically dropped by the input adapters in the destination switch.

The main component of the switch, which is the focus of our work, is the switch fabric. The basic structure of the fabric datapath is shown in Figure 5.2. The switch has I inputs, O outputs and B buffers. It is common for the fabric to have the same number of inputs and outputs ($I = O$). In most cases, the number of buffers exceeds the number of inputs and the number of outputs ($B \geq I \wedge B \geq O$).

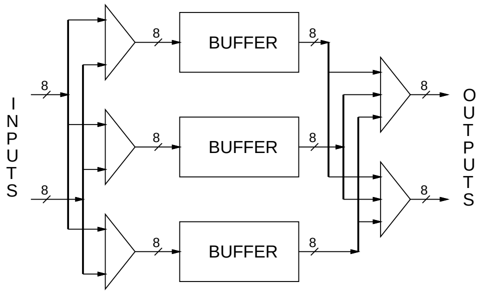


Figure 5.2: Switch Fabric Datapath.

The fabric datapath consists of three stages: input distributors, buffers, and output concentrators. The input distributors route data from the fabric inputs to the buffers. Each buffer stores a complete cell, organized as 53 bytes. The output concentrators provide paths from the buffers to the outputs of the fabric. As shown in the figure, all data paths in the fabric are 8 bits wide.

The architecture of the switch allows the concurrent operation of up to I paths from inputs to buffers and up to O paths from buffers to outputs. Due to the asynchronous implementation of the fabric, each path can operate at its own speed and there is no need for synchronization between the paths.

The switch architecture was chosen mainly to optimize buffer usage. With this architecture, every input and every output have access to all buffers, thus eliminating situations, present in other architectures, in which cells have to be dropped for lack of buffer access even though there may be empty (free) buffers. No cells are dropped in the fabric: handshaking with the input adapters is used as a back-pressure mechanism to stop cell transmission when all buffers are full (used).

Modularity and flexibility are also key features of the architecture. The switch fabric can easily scale both in number of inputs and outputs and in buffer space.

It is interesting to note that, due to the datapath organization, every cell is routed from input to output through a buffer. There are no "straight-through" paths. This is not a problem, for two reasons. First, it is unlikely that a cell will arrive at an input at the precise time when the destination output is idle, which would be the only scenario in which the straight-through paths could be used. Most cells would be buffered anyway. Second, the path through the buffer is operated in an optimized way: the buffer can transmit data to the output as soon as the first byte of the incoming cell is stored in buffer. There is no need to wait until the complete cell is stored in the buffer.

The control section of the switch, not shown in the figure, is also highly modular and distributed, which adds to the switch flexibility. Details of the datapath and the control path of the switch are given in the next section.

5.3 Switch Design

Figure 5.3 shows a block diagram of the switch. The switch has I inputs, O outputs and B buffers. The modular architecture of the switch allows each of these parameters to be chosen independently.

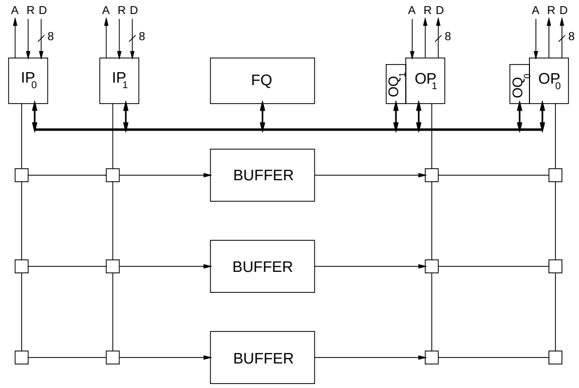


Figure 5.3: Packet Switch Block Diagram.

The lower portion of the figure corresponds to the datapath. The input distributors and output concentrators are implemented using switch matrices. Every node of the matrices is a cross-point switch element. Each column of the input matrix is formed by B elements so that each input can be distributed to every buffer. The input matrix contains a total of $I * B$ switch elements. Similarly, the output matrix contains a total of $O * B$ switch elements. Each buffer can store a complete cell. The buffers are implemented as independent FIFO storage elements, not as a common memory.

The top portion of the diagram shows the control path of the switch. The main control components are the Input Processors (IP), the Output Processors (OP) and their associated Output Queues (OQ), and the Free Buffer Queue (FQ). Each fabric input is controlled by its associated IP and there is an OP attached to every output. All IPs and OPs operate concurrently and independently of each other. The FQ is the only centralized control component and is used to keep track of free (available) buffers.

On the input side, the switch operates as follows. The switch is initially idle and, as is the case in many asynchronous systems, there is no activity until an input arrives. Packets arrive at the IPs one byte at a time. Data communication is synchronized using handshaking signals. Cell reception takes place in two phases: input path setup and cell buffering.

- **Input path setup.** The first byte of every packet is a self-routing byte (SRH) that identifies the destination output of the packet (it is not part of the cell that is transmitted). When the IP receives the SRH it requests a free buffer from the FQ. The FQ returns the address of a free buffer to the requesting IP and marks the buffer as used. The IP then activates the input switch element that corresponds to the assigned buffer and, concurrently, sends the address of the buffer to the OQ associated with the OP that controls the destination output port. At this point the path from the input to the buffer is properly setup and the reception of the cell can start.

  Each IPs must access the FQ and OQs during its setup phase. Direct channels from each IP to the FQ and every OQ could be used, but this would result in a very complex circuit that could be idle most of the time. Instead, a shared medium (BUS) is used to connect the IPs to the FQ and all OQs. It is critical to avoid a situation in which two or more IPs try to access the same resource concurrently. An arbitration process guarantees that only one

IP has control of the BUS at a time.

- **Cell buffering.** A complete cell will be received by the IP one byte at a time. During cell reception, the IP simply forwards each byte to the buffer through the input switch. The buffers operate as passive components that are waiting for data at their inputs. To achieve the high bandwidth required by modern ATM networks, the IP must introduce low overhead in this forwarding process. Cell buffering can be conducted concurrently by all IPs. Once the complete cell has been received, the IP deactivates the input switch element.

On the output side, the operation is as follows. The OPs are initially idle and will remain so until activated by the OQ. Each OQ is organized as a FIFO and stores the addresses of buffers that contain cells destined to the corresponding output port. An OQ activates its associated OP by sending it the first buffer address. Cell transmission takes three phases: output path setup, cell transmission, and buffer release.

- **Output path setup.** This phase is simpler than the input one. The OP simply receives the address of the buffer from the OQ and activates the output switch element that connects it to the required buffer.

- **Cell output.** Once a path is established, the cell can be transmitted to the output one byte at a time. The buffer operates as a passive component and will acknowledge transmission requests as soon as the first byte is ready. The OP forwards the bytes from the buffer to the fabric output. As was the case for the IPs, the forwarding process has to introduce low overhead to achieve high-speed transmission. All OPs can transmit cells concurrently.

  A positive characteristic of the switch, due to its asynchronous nature, is that cell output may start as soon as the first byte is in the buffer. There is no need to wait for the complete cell to be stored as is the case in many synchronous switches.

- **Buffer release.** Once the complete cell has been transmitted, the output processor deactivates the output switch element and, concurrently, reports to the FQ that the buffer is available again. The OP accesses the FQ through the shared BUS and only one OP can do it at a time. As was the case with the IPs, if several OPs contend for BUS access an arbitration process guarantees mutually exclusive operation.

  It is interesting to note that other possible strategies for buffer release are possible. Due to the possibility of inputs and outputs to operate at very different speeds, a decision was made to release the buffer only when it is completely empty. In situations in which the speeds are the same, a buffer can be released as soon as the first few bytes have been read, opening space for incoming bytes. This results in a more efficient utilization of the buffers, *i.e.,* less buffers may be needed to maintain the same performance of the switch.

The details of the operation of the main control components as well as their macromodular implementation will be reviewed in the following sections.

5.3.1 Asynchronous Bus (BUS)

A shared medium or bus can simplify the interconnection of different modules that must communicate with each other. While it is very common to use buses in synchronous systems, many asynchronous designs use point-to-point connections instead. The main problem to use buses in asynchronous designs is that buses cannot be considered isochronic [59, 54], and this compromises the validity of the common delay models[1]. In many cases, the cost of the point-to-point connections

[1] As pointed out by Molina *et al.* [54], in older technologies or in board level buses, in which the wire delay is considerably shorter than gate delays, the bus may safely be considered isochronic

is very expensive and restricts the flexibility and expandability of a system. This represents a disadvantage with respect to synchronous implementations.

Recent work on buses in asynchronous systems has shown the feasibility of their use. Bainbridge and Furber [5] introduced a bus design which is intended to provide system-level interconnection of asynchronous macrocells, like their AMULET processor core. In [54], Molina and Cheung propose a bus for quasi-delay-insensitive modular dual-rail systems.

Bus Structure

In our switch design, the need to interconnect all *IP*s and *OP*s to the *FQ* and the *OQ*s practically imposes the use of a bus, if the switch is to scale easily. Figure 5.4 shows the structure of the *BUS*. Actually, there are two buses that can operate concurrently: the input bus (*I-BUS*) and the output bus (*O-BUS*). The *I-BUS*, shown on the lower portion of Figure 5.4, is used for communication between the *IP*s, the *FQ*, and the *OQ*s during the input path setup phase. The *O-BUS*, in the top portion of the figure, is used for communication between the *OP*s and the *FQ* during the buffer release phase. The head of *FQ* connects to the *I-BUS* to send free buffer information to the *IP*s and the tail is hooked to the *O-BUS* to receive buffer release information from the *OP*s.

The terminology used to describe the operation of the *BUS* is borrowed from the MARBLE asynchronous bus [5]:

- Initiator: a module that can start a bus transaction by issuing a request on one of the *BUS* lines. In the switch, the *IP*s are the initiators in the *I-BUS* and the *OP*s are initiators in the *O-BUS*.

- Target: a module that can accept a request from an initiator through the bus. In our case, the *FQ* and the *OQ*s are the targets.

(see, for example, the TRIMOSBUS [84]).

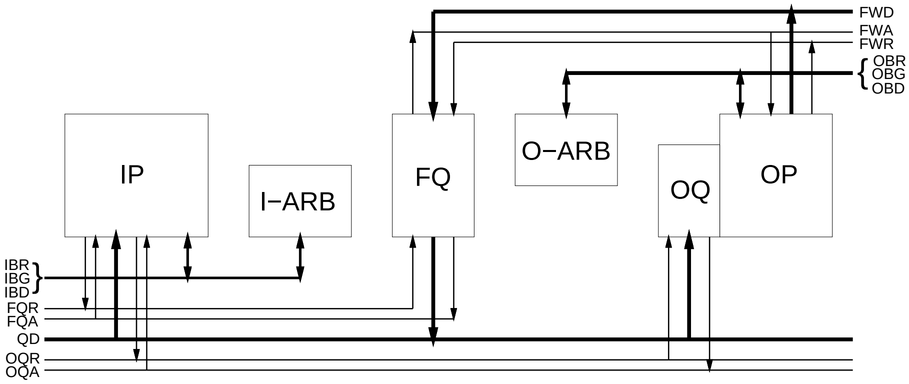


Figure 5.4: Asynchronous Internal Bus.

- Arbiter: a module that controls access to the bus. An initiator that wants to use the bus must first gain access to the bus by requesting it to the arbiter. Arbiters must guarantee mutual exclusion between initiators. As shown in the figure, there is an arbiter for the *I-BUS* (*I-ARB*) and another for the *O-BUS* (*O-ARB*).

The buses are organized in such a way that they operate almost like point-to-point channels between an initiator and a target, once the initiator has been granted use of the bus. The bus is composed of three types of signals:

- Data signals: used to transmit data from the sender to the receiver. As in point-to-point channels, in a push transaction, the initiator acts as the sender and the target is the receiver. In a pull transaction, the roles are interchanged. Data uses single-rail encoding.

- Control signals: (request and acknowledge) used by initiators and targets to synchronize the flow of data across the bus. The request signal is issued by the initiator and the acknowledge is issued by the target. In cases where there is more than one possible target, several request signals are used to identify the desired target. The control signals follow the pulse-mode handshaking protocol.

- Arbitration signals: (bus request, bus grant, and bus done) used by the initiators to gain and release access to the bus. The arbiters follow the *RGD* (request, grant, done) protocol (see Section 4.4). This protocol is an unusual handshaking protocol in the sense that events in the last signal (bus done) is not acknowledged by any other signal. It can be thought of as a 3-phase handshake.

I-BUS Operation

The *I-BUS* is used during the input path setup phase. This phase requires two bus transactions, which take place as follows. The IP issues a bus requests (IBR) to the arbiter (*I-ARB*). The arbiter grants the request. At this point, the IP has control of the bus and requests a free buffer from FQ, using request signal FQR. This is a pull transaction since the initiator receives data from the target. When FQ is ready, it sends back the free buffer and issues an acknowledge (FQA). Note that all IPs will receive this information, since all of them are connected to the bus. Only the requesting one will actually use the information and all others will ignore it. This concludes the first transaction.

The IP still maintains control of the bus and starts the second transaction by sending the free buffer to the desired OQ and issuing a request (OQR). In this case, there are several OQs connected to the bus. There must be enough request signals to allow the IP to identify which OQ is the target of the transaction. This is a push transaction. When the OQ stores the data, it issues an acknowledge (OQA) that completes the transaction. At this point, the setup phase is finished and the IP releases the bus by issuing a bus done signal (IBD).

O-BUS Operation

The *O-BUS* is used during the buffer release phase. This phase is very simple and requires only one transaction, which takes place as follows. The OP requests the bus using OBR. *O-ARB* grants the bus using OBG. The OP then sends the available buffer to the FQ and issues a request (FWR). When FQ has stored the data, it issues an acknowledge (FWA) that completes the transaction. The OP releases the bus by issuing a bus done signal (OBD).

There is no priority in the arbitration processes. All IPs and all OPs are considered equal. In this case, IPs and OPs require access to the BUS only once for every cell so, given that the FQ access time is less than the time it takes to receive or transmit a complete cell, no processor can monopolize access to the BUS. If an access to the FQ by an IP takes T_I time, the worst case wait for any IP before it is granted access is $(I-1)*T_I$. The worst case wait for an OP is $(O-1)*T_O$, where T_O is the time taken by an OP to access the FQ.

5.3.2 Input Processor (IP)

The input processor (IP) is a critical component of the switch. Its operation has a large impact on both the latency of a cell traversing the switch and the maximum throughput that each input can handle. All incoming cells "traverse" the IP to reach the buffer where they are stored. Thus, our objective is to design the IP so that it introduces low overhead in this critical path. Figure 5.5 shows the IP module and its interface signals.

IP Operation

As explained earlier, cell reception consists of two phases: path setup and cell buffering. The operation of the IP is different in the two phases, as described in the specification in Figure 5.6.

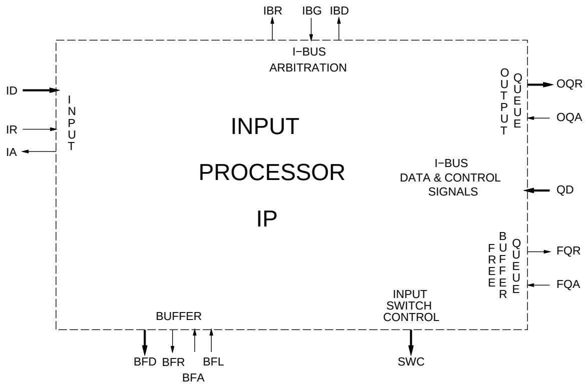


Figure 5.5: Input Processor Module Interface.

```
IF ($SRH$)
   FORK
      reset $SRH$
      {  get_bus;
         set $BUS$;
         get_free_buffer;
         FORK
            load $SWI$
            {  write_output_queue;
               reset $BUS$;
               release_bus
            }
         JOIN
      }
   JOIN
ELSE
   IF (write_buffer = $LAST$)
      FORK
         set $SRH$
         clear $SWI$
      JOIN
```

Figure 5.6: Input Processor Specification.

Boolean variable SRH (Self Routing Header) is used to identify the phase. If $SRH = 1$ then the IP is expecting the self routing header as the next byte. When the byte arrives, SRH is rest, the path to a free buffer is established and cell buffering starts. When the last byte of the cell arrives, SRH is set and the IP is ready to start the reception of the next cell.

A second boolean variable, BUS, is used to keep track of the status of the bus. $BUS = 1$ indicates that the bus has been granted to the IP. In this case, the processor has control over the bus and can proceed with the required transactions. When the IP completes the use of the bus, it resets BUS. The order of the processes is critical: BUS is set *after* the bus has been granted and reset *before* the bus is released. This guarantees that only one IP is controlling the bus at any time.

Register (SWI) is used to store the address of the free buffer that will be used to store the incoming cell. This information is received from FQ through the BUS. The outputs of the register are used to control which input is activated to communicate the IP with the buffer. Clearly, the size of the register depends on the number of buffers in the switch.

IP Macromodule Design

Using the macromodules presented in Chapter 4, the above specification can be directly translated into a macromodular diagram, as shown in Figure 5.7. Every command maps directly to a macromodule. Every ";" in the above specification maps into a 2-STEP module, which are used for sequencing.

Communication through the bus is different from that through point-to-point channels: any process that uses the bus must send and receive signals through a Bus Driver (B/D) macromodule. Boolean variable BUS is used to enable and disable these macromodules. The modules are enabled *after* the bus has been granted to

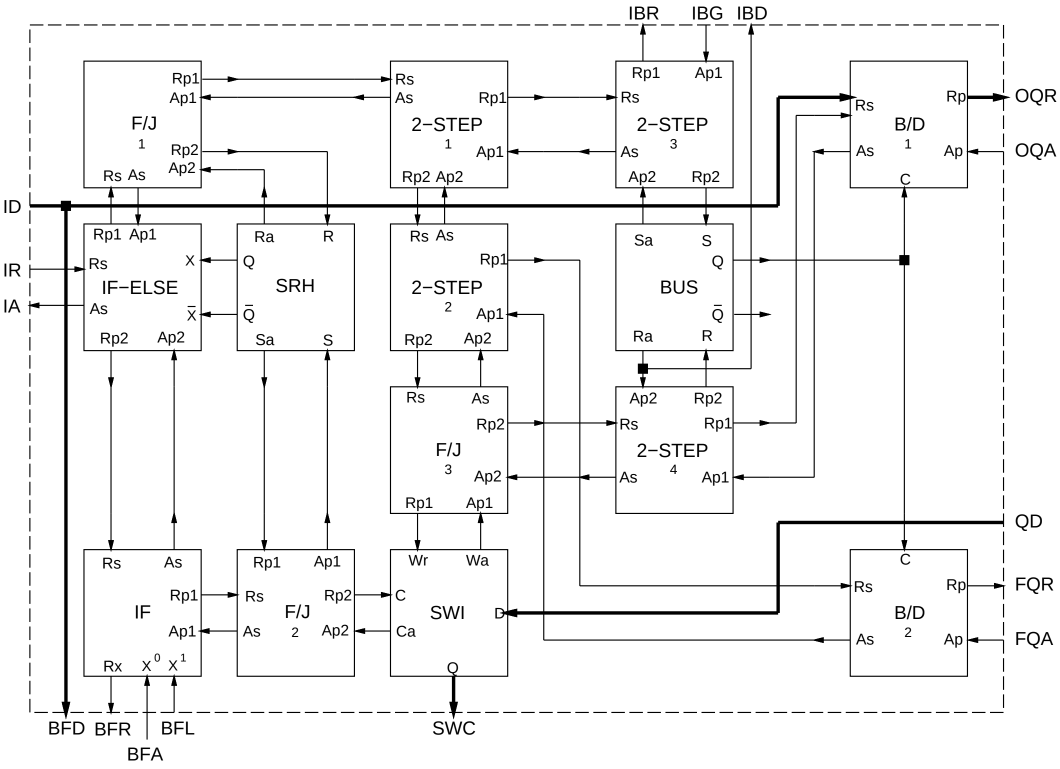


Figure 5.7: Input Processor Macromodule Diagram.

the IP and disabled *before* the bus is released. This guarantees that there are no conflicting B/D modules enabled concurrently.

A careful reader will note that there is a 2-STEP module missing in the diagram. This exception in the translation is related to the bus request/release process. As pointed out earlier, the arbitration signals use an RGD protocol, which consists of three phases: the *Done* is not acknowledged. An unacknowledged process means that the actual completion of the process is not important and may be considered completed immediately: the request signal may be used as its own acknowledge. In this case, the release_bus command has no completion signal, thus the bus done signal (IBD) is sent to the next process as its completion signal.

IP Optimization

The use of pre-designed modules may introduce inefficiencies in the design of a system. In some cases, it is possible to examine the macromodule diagram and optimize it in a number of ways. Two basic optimizations, introduced in Chapter 4, are used: JOIN elimination and module substitution.

JOIN elimination

Synchronizing two processes is an expensive operation, both in terms of performance and area. One of the most effective ways to improve the performance of a system is to eliminate synchronization modules. Figure 5.8 shows a segment of the IP specification. A few irrelevant details have been deleted to simplify the analysis.

The first branch of the fork contains a single process: reset SRH while the other branch is formed by a sequence of five processes: get_bus; set BUS; get_free_buffer; write_output_queue; reset BUS. The first branch corresponds to the reset of an SR flip-flop, requested and acknowledged locally inside the IP. On the other hand, the second branch involves the set and reset of SR flip-flops and

```
FORK
    reset $SRH$
    {  get_bus;
        set $BUS$;
        get_free_buffer;
            ....
                write_output_queue;
                reset $BUS$;
            ....
    }
JOIN
```

Figure 5.8: *IP* Specification Segment: Two Concurrent Processes.

several external actions: access to the arbiter, the free buffer queue and one of the output queues, all through the bus.

It seems that, under all reasonable scenarios, the first branch of the fork will *always* complete first. In this case, the JOIN element can be eliminated and the completion signal of the second branch of the fork (the "slow" branch) used in place of its output. Similar analyses lead to the elimination of the three JOIN elements in the *IP* module.

Module Substitution

A second optimization that can improve the performance of a system is the use of simplified version of a macromodule if reasonable timing assumption are met. Figure 5.9 shows a segment of the *IP* macromodule diagram. Variable SRH is the selection variable for the IF-ELSE module. SRH is reset by a pulse sent from the IF-ELSE module. In this situation, an IF-ELSE module that allows X and $\overline{X}$ to change while its output pulse is still active must be used.

A simpler version of the IF-ELSE module could be used if SRH is guaranteed

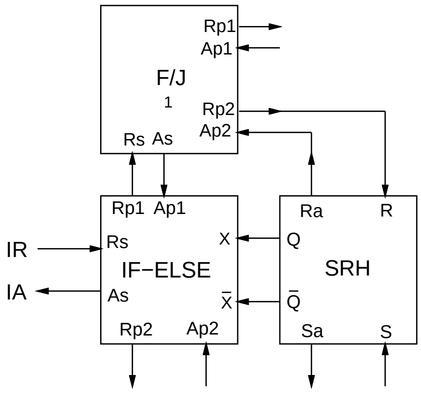


Figure 5.9: Segment of the *IP* Macromodular Diagram.

to remain stable while the IF-ELSE output pulse is active. To use this version of the module in our system, variable *SRH* cannot be reset by the IF-ELSE but by a different process. In our case, this is not a difficult change. As shown in Figure 5.8 above, the resetting of *SRH* is done concurrently with 5 other processes. Using, for example, the bus grant signal (*IBG*) to reset the variable will guarantee that its value remains stable and the simpler IF-ELSE module can be used safely.

The combination of this two optimizations usually results in a more efficient implementation of the system. Figure 5.10 shows the optimized macromodule diagram of the *IP*. To simplify the diagram, the 2-STEP modules have been substituted by their circuit implementation. As shown in Chapter 4, the pulse-mode 2-STEP module can be implemented using only three wires.

The optimized diagram contains only half of the macromodules of the original diagram, and all the FORK/JOIN modules have been eliminated. Clearly, this optimization step has a large impact on the performance of the module.

IP Critical Path

Every byte of a cell "traverses" the *IP* on its way to the assigned buffer. It is critical to keep the overhead of this path as low as possible in order to maintain

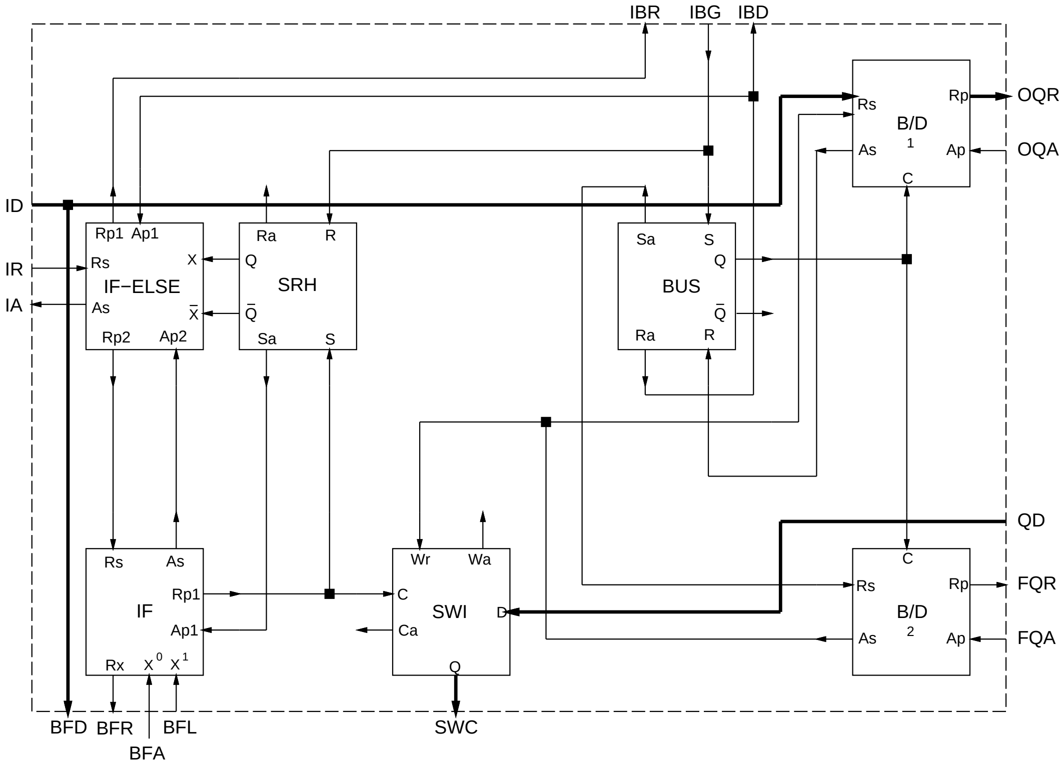


Figure 5.10: Optimized Input Processor Macromodule Diagram.

high-bandwidth switching. The macromodules can be substituted by their circuit implementations to obtain the complete IP circuit. Figure 5.11 shows the actual circuit path from the fabric inputs to the IP outputs to the buffers (connected through the input switch elements).

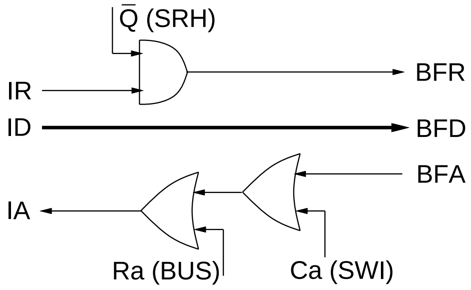


Figure 5.11: Fabric Input to Buffer Path (Data and Control).

The IP critical path is very efficient. The Input Data (ID) is not affected by the IP (except for the delay that may be introduced by wiring, which depends on the final layout of the circuit). The control signals –Input Request (IR) and the Buffer Acknowledge (BFA)– go through a single gate each[2]. Due to this "short" path inside the IP, the buffer itself will have the largest impact on the input throughput of the switch.

5.3.3 Output Processor (OP)

The Output Processor (OP) is also a critical component of the switch. The OP controls the outgoing flow of data. It can have a large impact on the performance of the switch because every byte of a cell passes through the OP in its way from the buffer to the fabric output. As was the case with the IP, the goal is to keep the overhead of the OP as low as possible. Figure 5.12 shows the OP module and

[2] The two 2-input OR gates will be mapped into a single 3-input OR gate during the technology mapping process.

its interface signals.

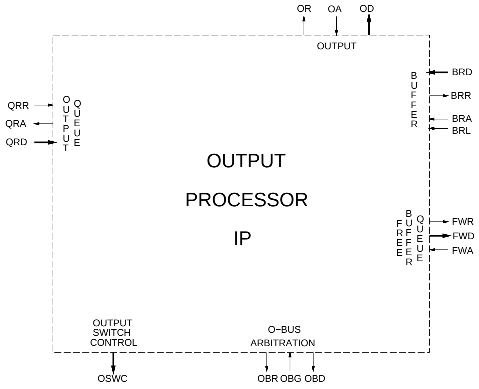


Figure 5.12: Output Processor Module Interface.

OP Operation

Cell transmission consists of three steps: Output path setup, cell output, and buffer release. The operation of the OP in all phases is described in the specification in Figure 5.13.

The OP is initially idle. It is activated by its Output Queue (OQ) whenever it receives the address of a buffer. The OP starts by setting up the output switch elements. A register (SWI) is used to store the address of the buffer where the outgoing cell is stored. This information is received from OQ. The size of the register depends on the number of buffers in the switch. Cell output takes place next. Boolean variable $DONE$ is used to signal when the complete cell has been transmitted.

When the complete cell has been transmitted, the buffer can be returned to the Free Queue (FQ). This phase of the operation requires access to the $O\text{-}BUS$. A second boolean variable, BUS, is used to keep track of the status of the bus. $BUS = 1$ indicates that the bus has been granted to the OP. In this case, the

```
set_switches;
reset $DONE$;
REPEAT
    IF (read_buffer = $LAST$)
        set $DONE$;
    write_output
UNTIL ($DONE$);
FORK
    reset_switches
    {  get_bus;
        set $BUS$;
        release_buffer;
        reset $BUS$;
        release_bus
    }
JOIN
```

Figure 5.13: Output Processor Specification.

processor has control over the bus and can proceed with the required transaction. When the *OP* completes the use of the bus, it resets *BUS*. The order of the processes is critical: *BUS* is set *after* the bus has been granted and reset *before* the bus is released. This guarantees that only one *OP* is controlling the bus at any time.

OP Macromodule Design

Figure 5.14 shows the *OP* macromodule diagram. The diagram was obtained from the specification above. There are many similarities with the *IP* diagram, such as the use of an *RGD* protocol to communicate with the arbiter, variable *BUS* to keep track of access to the bus and to control the *B/D* module use to interface to the bus.

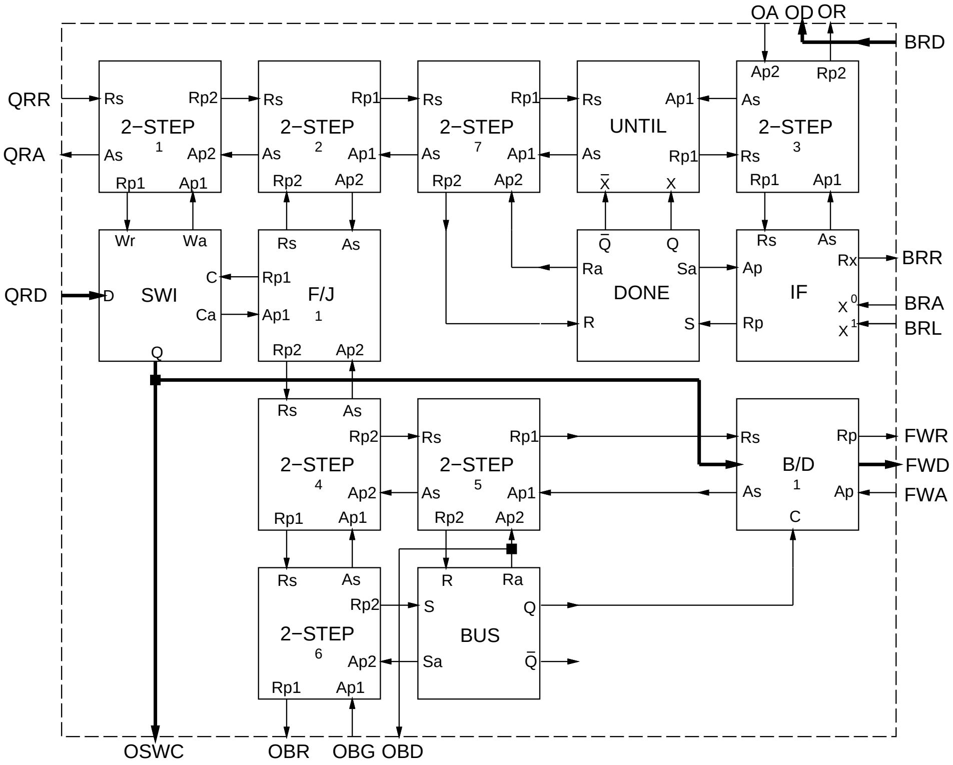

OA OD OR
BRD
QRR
QRA
Rs Rp2 2−STEP 1 As Ap2 Rp1 Ap1
Rs Rp1 2−STEP 2 As Ap1 Rp2 Ap2
Rs Rp1 2−STEP 7 As Ap1 Rp2 Ap2
Rs Ap1 UNTIL As Rp1 X̄ X
Ap2 Rp2 As 2−STEP 3 Rs Rp1 Ap1
Wr Wa C SWI Ca D Q
QRD
Rs As Rp1 F/J 1 Ap1 Rp2 Ap2
Q̄ Q Ra Sa DONE R S
Rs As Rx Ap IF X 0 X 1 Rp
BRR
BRA
BRL
Rs As Rp2 2−STEP 4 Ap2 Rp1 Ap1
Rs Rp1 2−STEP 5 As Ap1 Rp2 Ap2
Rs Rp B/D 1 As Ap C
FWR
FWD
FWA
Rs As Rp2 2−STEP 6 Ap2 Rp1 Ap1
R Ra S Q BUS Sa Q̄
OSWC
OBR
OBG OBD

Figure 5.14: Output Processor Macromodule Diagram.

OP Optimization

As pointed out previously, the initial macromodular diagram can be optimized using different techniques and making reasonable timing assumptions. The optimized version of *OP* is shown in Figure 5.15. In this figure, the 2-STEP modules have also been substituted by their all-wires implementation to simplify the diagram.

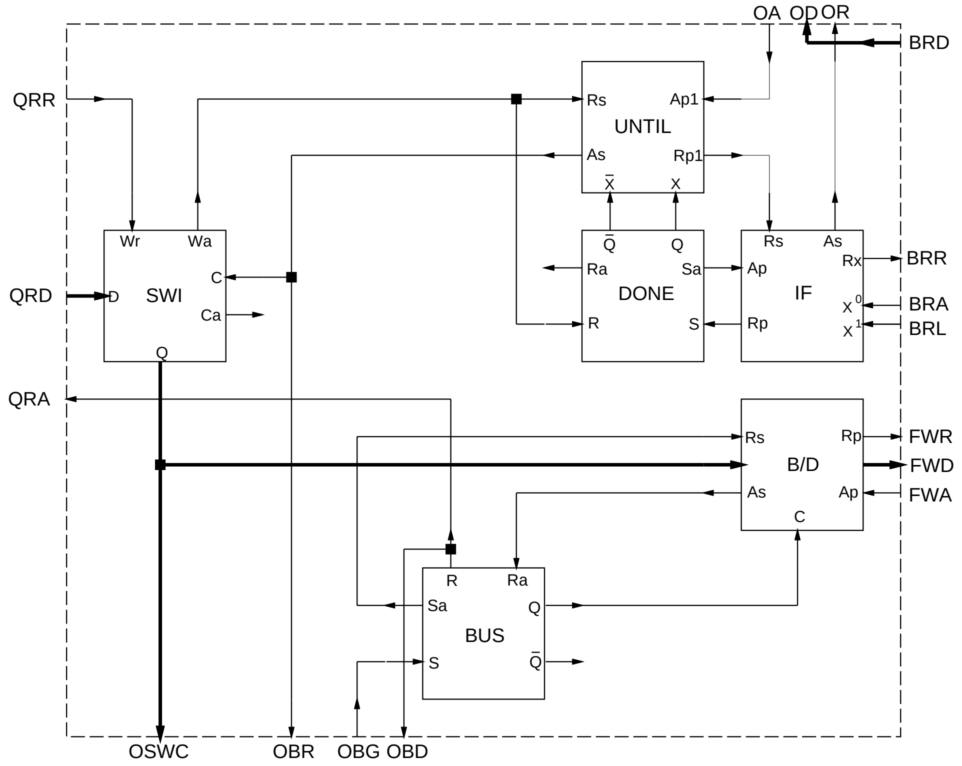


Figure 5.15: Optimized Output Processor Macromodule Diagram.

OP Critical Path

Finally, we examine the critical path of *OP*. Every cell traverses the processor on its way from the buffer to the fabric output. This is the path that has the largest impact on the output throughput of the switch. The section of the buffer-output path that corresponds to OP is shown in Figure 5.16.

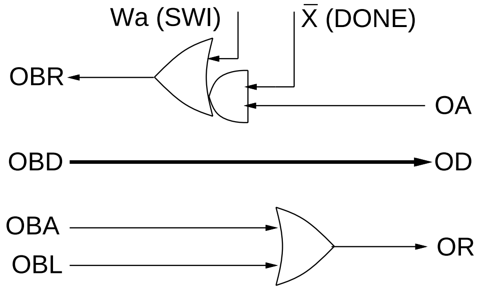


Figure 5.16: Buffer to Fabric Output Path (Data and Control).

The use of pulse-mode macromodules and reasonable timing assumptions to optimize OP results in a very efficient path, which introduces low overhead in the cell transmission process.

5.3.4 FIFO Buffers

The Buffers ($BUFF$) are, in a sense, the main components of the switch. Every cell must be stored in one of them. The buffer has two ports. An input port that receives cells (a byte at a time) from the IP and an output port that transmits cells to OP, also a byte at a time. As shown in Figure 5.17, both ports of the buffer are passive, activated by IP and OP respectively.

The buffer communicates with IP and OP using pulse-mode handshaking. The processors request data transfers using the request signals: IP uses R_W for writing and OP uses R_R for reading data from the buffer. The buffer acknowledges through A_W and A_R, respectively. An important characteristic of the buffer interface is that the buffer reports to IP when the last byte of a cell has been stored, using L_W instead of AW. Similarly, the buffer indicates to OP that the last byte has been transmitted by acknowledging through L_R instead of A_R.

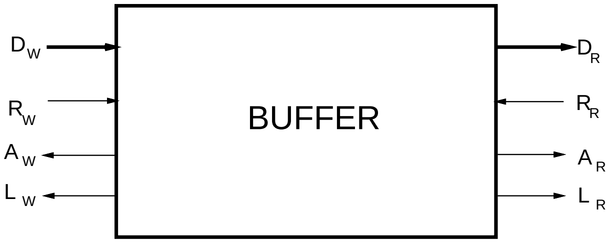


Figure 5.17: Buffer Module Interface.

Asynchronous FIFO storage can be organized in different ways. Two possible organizations are discussed below, and the problems and benefits of each are also presented.

FIFO Buffers: Micropipeline Implementation

In [83], Sutherland introduced micropipelines as a simple and elegant way to implement asynchronous pipelines. A micropipeline without processing between stages is a simple FIFO buffer. Figure 5.18 shows how a buffer can be implemented using the basic micropipeline structure and pulse-mode macromodules.

As shown in Chapter 4, pulse-mode handshaking allows the use of simple, level-based latches, as opposed to expensive event-driven ("capture-pass") latches, suggested by Sutherland. Two modifications to the usual micropipeline are required to implement the desired buffer interface, as shown in Figure 5.18.

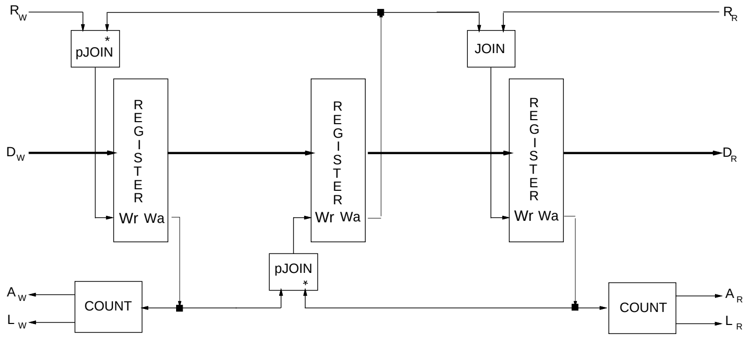


Figure 5.18: FIFO Buffer: Micropipeline Implementation.

First, the output port of traditional micropipelines is an active port. It sends out a request as soon as there is valid data stored in the last stage. On the contrary, the output port of the buffer must be a passive port. The buffer must wait for a request from OP before sending any information to the output. The use of a JOIN element instead of a primed one (pJOIN) in the last stage will solve this problem. Unfortunately, this means that data will not be stored in the last register until the request arrives, thus the buffer must have $N + 1$ stages in order to store N bytes.

Second, the designed interface with IP requires that the buffer reports when the last byte of the cell is stored. Due to the elastic nature of the micropipeline, there is no easy way to identify this situation directly. Instead, a counters must be added to the buffer, to keep track of the number of acknowledge pulses, as shown in Figure 5.18. The counter receives the pulses in its input and acknowledge the first $N - 1$ of them through output A_W and the Nth pulse through L_W, indicating that the last byte of the cell has been stored.

The buffer must also report to OP when the last byte of the cell has been transmitted. A second counter is added on the output side. This counter keeps track of the number of output acknowledge pulses and indicates, through output L_R, when the last byte of the cell has been transmitted.

Due to the pipeline structure of the FIFO buffer, every byte of data must propagate through all stages before it is available at the output. This has a large impact on the latency of a cell through the switch. Clearly, the latency of every byte through the buffer is proportional to the total number of bytes in the cell. On the other hand, the input throughput of the switch depends on the response time of the buffer, *i.e.,* the time it takes to store a byte in it, independently of the time it takes to appear at the output. The response time of this implementation of the buffer depends mostly on the response time of the latches, due to the low overhead introduced by the control components.

Although throughput is the key parameter in the operation of the switch, in some cases latency is also important. The long latency associated with the pipeline implementation of the buffer may be too high. In such cases, a different implementation of the buffer is needed. In any case, a low-latency buffer must also have a fast response time to maintain high throughput. Such an implementation is discussed in the following section.

FIFO Buffers: Low-Latency Implementation

The need for low-latency FIFO buffers in packet switches has been recognized by several researchers (see, for example, [49, 103, 80]). In [102], Yantchev *et al.* discuss the implementation of a transition signaling FIFO buffer that uses a structure similar to the one presented here.

The basic idea of the low-latency buffer is to allow an incoming byte to be stored directly in any register and, conversely, the contents of every register to be read directly to the output. Thus data never moves inside the buffer. This represents the minimum possible latency through the buffer. Figure 5.19 shows the macromodule diagram of the buffer.

The structure of the low-latency buffer is very similar to the datapath of the complete switch. The registers are connected in “parallel”, thus allowing access to

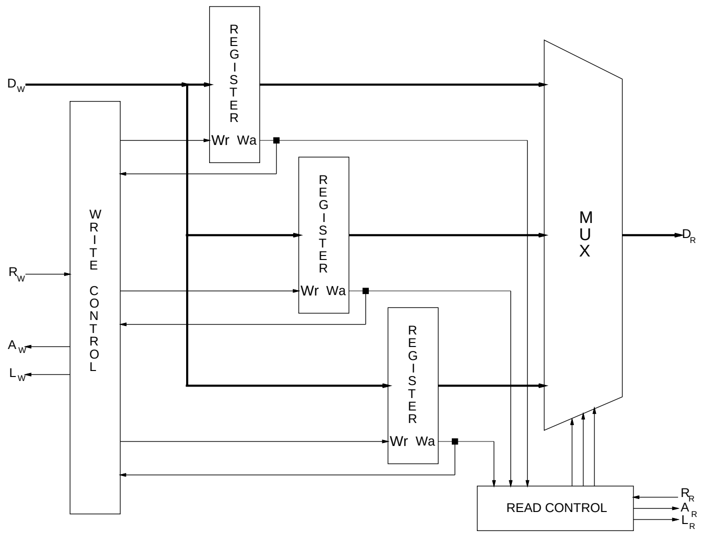


Figure 5.19: Low-Latency FIFO Buffer.

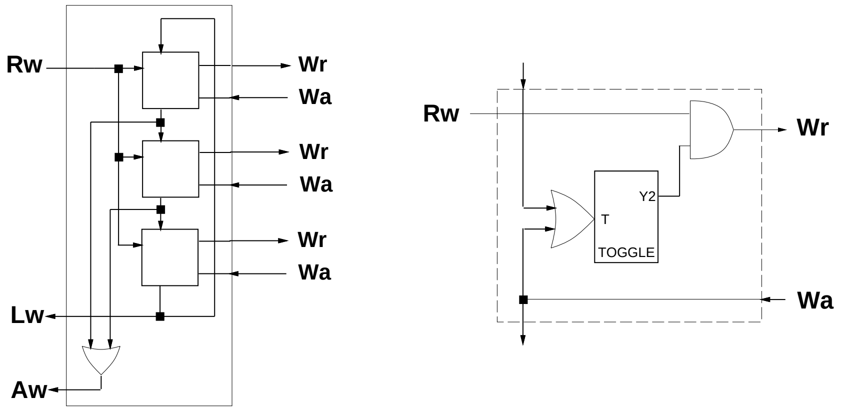
any one of them directly. Input data is forked to the input of all registers. The Write Control module selects the register that is accessed in a write operation. On the output side, a multiplexer is used to select the register that will output data in a read operation. The Read Control module keeps track of the register to be read and generates appropriate control signals for the multiplexer.

A write operation in this buffer involves two actions: (*i*) Storing a byte of data in a register, and (*ii*) "updating" the information in the Write Control to activate the following register on the next write operation. The two operations must be executed in sequence, thus the response time of the buffer is likely to be larger than that of the pipelined implementation, which only needs to store the byte in the input register. "Updating" is carried out after the buffer has acknowledged the write operation. This is a typical trade-off between throughput and latency. In any case, the response time of the low-latency buffer must be fast enough to sustain the input throughput required by the switch. The same considerations apply to read operations.

It is important to note that reading and writing can be done concurrently.

Reading can start as soon as the first byte has been stored in the buffer. The Read Control Module is responsible for not allowing a read operation from a register that has not been written yet.

Figure 5.20 shows the implementation of the Write Control module. This module is essentially a pulse-mode ring counter. A ring counter is a module that counts events and provides a "ONE-HOT" output, *i.e.,* an N-counter has N outputs and only one of them is active at any given time. Ashkinazy *et al.* [2] and Yantchev *et al.* [102] introduced designs for asynchronous ring counters. Both designs have transition signaling interfaces thus are not applicable to pulse-mode systems.



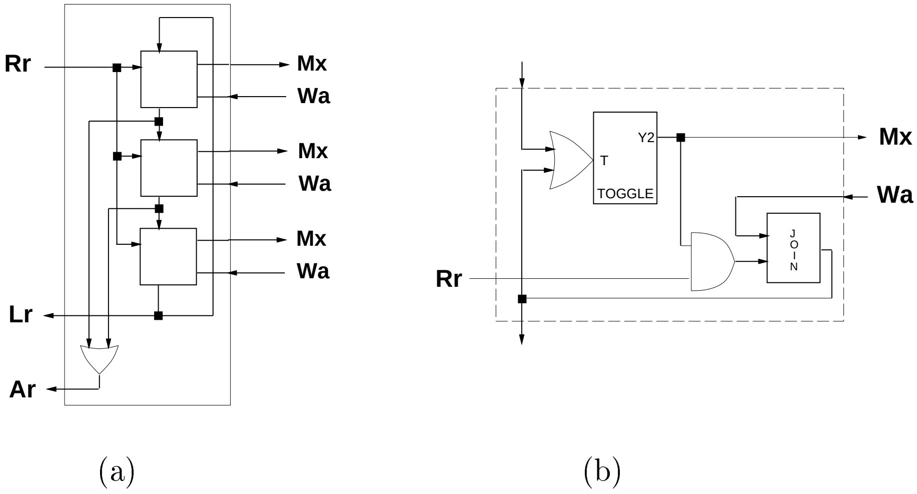
(a) (b)

Figure 5.20: Low-Latency FIFO Buffer: Write Control.

The block diagram of the control module is shown in Figure 5.20(a). W_r and W_a are pulse signals that correspond to the write request and acknowledge of every register in the buffer. Figure 5.20(b) shows the implementation of the counter blocks, which is similar to the stages of the counters discussed in Chapter 4. Only one block will has $Y_2 = 1$ at any given time. This guarantees that only one write request is generated. The corresponding acknowledge clears that block and sets the following one, in preparation for the next write operation. The “ring” nature of the counter is due to the connection of the last block to the first one.

The write acknowledge of the last register is passed directly as L_w, indicating that the last byte of a cell has been stored. All other acknowledge signals are merged together to generate A_w, the acknowledge signal for other write operations.

The Read Control Module is also based on a ring counter, as shown in Figure 5.21(a). The main difference with respect to the Write Control is that a read operation can only be executed if the corresponding register has been written. It can happen that the output is operating at a higher speed than the input and the Read Control can reach a register that has yet to be written. In this case, the read operation must wait for the write operation to complete. Figure 5.21(b) shows that a JOIN element is used to synchronize the pulse from the counter with the write acknowledge of each register to guarantee that the read operation is executed only after the write to the register is complete.



(a) (b)

Figure 5.21: Low-Latency FIFO Buffer: Read Control.

A design decision to depart from pure pulse-mode operation has been made in the Read Control Module. The multiplexer control signals (Mx) are level signals. The use of level signals greatly simplifies the design of the multiplexer, which justifies the departure from the pulse-mode discipline. The level signals are available inside the counter thus there is no additional cost related to their generation. Also, these are one-hot signals also, which makes the multiplexer design even simpler. Clearly, this decision does not compromise the correctness and robustness of the module.

5.4 Switch Simulation

The operation of the macromodular packet switch was simulated using SIMIC [2]. Figure 5.22 shows details of the operation of two input channels and two output channels of the switch.

5.4.1 Operation of the Macromodular Switch

The figure shows the request (R) and acknowledge (A) signals, and 4 data bits (D) of each of the four channels[3]. The reception of two cells through each of the input channels and their output transmission was simulated. Cell $1A$ is received by $Input_1$ first, addressed to $Output_1$. Concurrently, after a short delay, cell $2A$ is received by $Input_2$ and is also addressed to $Output_1$. Cell $1B$, addressed to $Output_2$, is received by $Input_1$ immediately after cell $1A$ arrives. Finally, cell $2B$ is received by $Input_2$ after cell $2A$. Cell $2B$ is the only cell in this simulation run addressed to $Output_2$. Each cell consists of a one-byte self-routing header, which is used for internal processing and is not forwarded to the output, and 53 data bytes, which are stored in a buffer and transmitted to the destination output.

[3] Only 4 data bits are presented in the figure to limit the total number of displayed signals. The actual datapaths of the simulated system are all 8 bits wide.

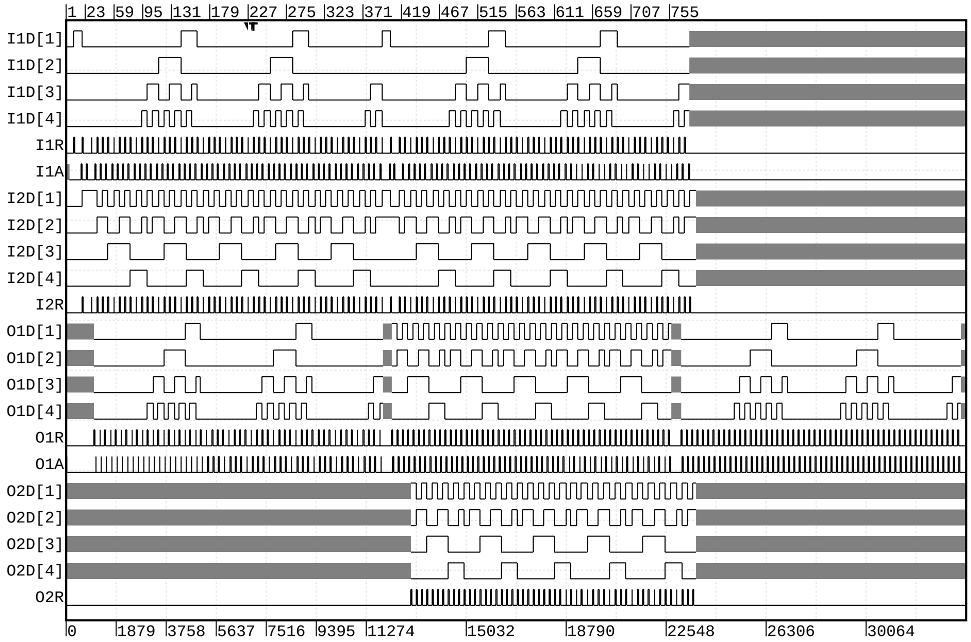
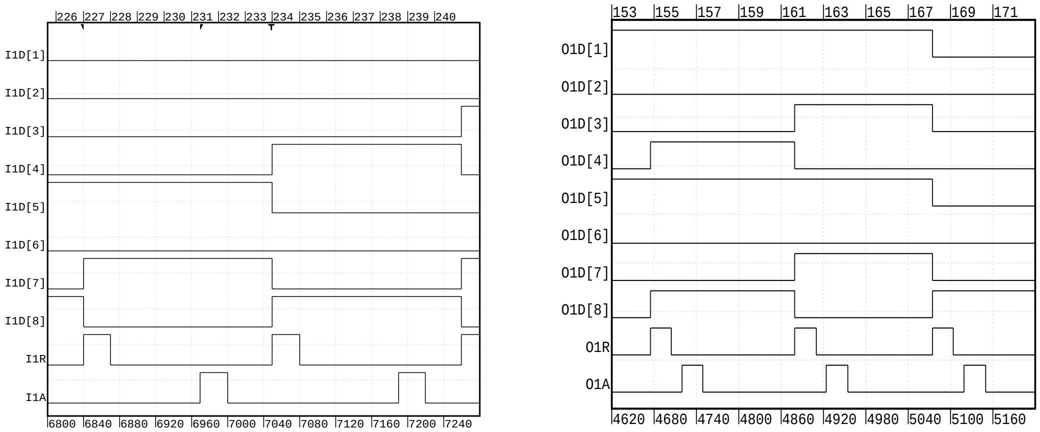


Figure 5.22: Functional Simulation: Packet Switching.

Several aspects of the operation of the switch are depicted in Figure 5.22. All Input and output channels operate concurrently, at their own speed, and without the need for internal synchronization. This results in a short cell latency traversing the switch. Cells $1A$ and $2A$, addressed to the same output, are received concurrently. The first one is transmitted almost immediately to the output while the second is buffered for latter transmission. Cells are transmitted to the outputs in the same order that they were received (cell $2A$ is sent before cell $1B$, which arrives later).

Figure 5.23(a) shows the details of the reception of two incoming cell bytes. The response time of the switch is slightly different for the two bytes, due to the asynchronous internal operation of the switch. Similar characteristics are shown in Figure 5.23(b) for the output transmission of two consecutive bytes.

Simulated response times, channel throughput rates and other key parame-



(a) (b)

Figure 5.23: Simulation: (a) Input, and (b) Output Response Time.

ters of the switch are given in the following subsection.

5.4.2 Characteristics of the Macromodular Switch

For simulation purposes, the input environment feeds cell data bytes to the switch at a fixed rate of 380 Mbits/Sec, close to the maximum input throughput of the switch. The output environment accepts cell data bytes from the switch as soon as they are available and takes a fixed time (5 nSec) to “process” each one.

The basic pulse-mode elements were simulated using SPICE [58]. Typical parameters for a MOSIS 1.2 μm CMOS technology and a 5V power supply were used in the simulation. These results were fed to SIMIC, which was used for the complete system simulation. Figure 5.1 summarizes the results of the simulation. The key response times and the equivalent throughput rates are listed. It is important to note that listed throughput results are always net values, *i.e.,* only actual cell bits are taken into account. Added overhead information, such as the self-routing

	TIME	**THROUGHPUT**
PARAMETER	nSec	Mbits/Sec
Input Ports		
Cell Reception (53 bytes)	1105.5	383
Cell Reception (including SRH processing)	1133.0	374
SRH Processing (no contention for bus)	24.75	
Data Byte Processing time (average)	13.50	
Output Ports		
Cell Transmission (53 bytes)	1061.5	399
Inter-Cell Dead time (second cell from Buffer)	42.25	384
Inter-Cell Dead time (second cell from Input)	82.50	370
Data Byte Processing time (average)	17.00	
Switch Latency		
Input–Output Latency (Output Port Free)	79.95	

Table 5.1: Timing Parameters of Simulated Packet Switch.

header, are not counted as transmitted bits. This shows more realistic throughput rates but results in lower values.

The table indicates that the switch can sustain an input throughput of 374 Mbits/Sec, and an output throughput of 370 Mbits/Sec. This represents rates 2.4 times higher than the target ATM value of 155 Mbits/Sec. An interesting parameter is the inter-cell dead time in output ports. This is a result of internal activity in the buffers, output processors (OPs) and output queues (OQs). Dead time is only 3.8% of the cell transmission time for the common case, *i.e.,* the cells are already in the buffers, and 7.2% when the cells are being received concurrently. The difference is due mainly to the response time of the output queue (OQ) of the port. A desirable feature of the switch is the low input to output latency of cells traversing the switch (~80 nSec). This is clearly a result of the asynchronous implementation. The following subsection presents a comparison with a similar synchronous switch.

5.4.3 Comparison with a Synchronous System

The Switch–on–a–Chip [20, 42, 21] is an integrated cell-switching chip developed by the IBM Research Laboratory in Switzerland. It has been used in commercial ATM packet switches by IBM. Dutton and Lenhard [21] report the following characteristics of the chip: It has 16 input ports and 16 output ports. Each port may operate at speeds up to 400 Mbits/Sec simultaneously with all other ports. The 400 Mbits/Sec data rate is obtained when an internal clock rate of 50 MHz is used. The input and output datapaths are 8-bit wide. It contains 128 buffers accessible by every input and every output port. It has a "back-pressure" mechanism to allow the chip to refuse input when buffers are full. The chip is built on a 14.7 mm^2 die using a 0.7 μm CMOS technology. It contains around 2.4 million transistors.

The Switch–on–a–Chip (SoaC) was chosen as a reference because it is a

PARAMETER	Units	**Macromodular Switch** (Simulated)	**Switch–on--a–Chip** (Fabricated)
Channel Throughput (sustainable)	Mbits/Sec	370	400
Input–Output Latency (no queuing)	nSec	79.75	∼1000

Table 5.2: Comparison with Fabricated Synchronous Switch.

commercial, synchronous system with a similar architecture to our Macromodular design. Although the macromodular design may be regarded as a partial design, the results are very encouraging. Table 5.2 compares the main performance characteristics of the Macromodular switch and the SoaC. The table indicates that the asynchronous design can reach almost the same sustained per-channel throughput as the SoaC while keeping the latency over 10 times shorter. It is very important to keep in mind that the Macromodular switch is not a complete design and the results correspond to a simulation, thus they are only indicative of possible performance ratings.

5.5 Conclusions

The packet switch design presented in this chapter demonstrates the viability of pulse-mode macromodules to implement complex, high performance systems. The switch organization, asynchronous operation, and low control overhead introduced by pulse-mode macromodules result in a design that can handle 2.4 times the target ATM throughput of 155 Mbits/Sec. Also, the switch is characterized by very low input-to-output latency (over 10 times shorter than a comparable synchronous implementation).

The design of the switch may be regarded as a partial one. Several aspects of ATM switches have not been implemented, such as support for multicasting and cell priority processing. In any case, the results are very encouraging. Competitive, high-performance asynchronous systems are the result of low control overhead and careful attention to timing. The underlying self-timed strategy used in macromodular systems (detailed control of module internal timing and delays combined with delay-insensitive external interfaces) seems to be a right step in that direction.

Chapter 6

Conclusions

The main goal of the work described in this thesis was to contribute to the design of robust, performance-competitive asynchronous systems. The realization, through the analysis of many different experiences, that high-performance systems are the result of low control overhead and careful attention to timing focused the work on macromodular systems and on the underlying self-timed strategy used in the design of these systems.

If asynchronous designs are to be widely used, it is critical to identify application domains where asynchronous techniques are of practical interest and to demonstrate their potential advantages using real designs. A second goal of this work was to demonstrate the feasibility of the approach by applying it in the design of a large, relevant example.

The following results are the contributions of this work to achieve those goals:

- The introduction of an architectural optimization to "eliminate" one of the largest sources of control overhead in 4-phase macromodular systems: the return-to-zero phase. Actually, the optimization allows the overlap of the redundant phase with the execution of productive work that otherwise would be delayed. The circuitry guarantees that the system can operate correctly at the increased performance.

In particular, three new asynchronous sequencers were designed. Each sequencer increases the throughput of the entire system. Existing asynchronous datapaths do not operate correctly at this increased level of concurrency: *data hazards* may result. Modifications to the datapath were introduced to insure correct operation. Specifically, we introduce interlock mechanisms that safely handle concurrent operation in both dual-rail and single-rail datapaths.

The architectural optimization can also be regarded as an optimization for low power. The increased throughput obtained through more concurrent operation can be traded for lower energy consumption by a reduction of the power supply voltage, as discussed in Section 2.6.

SPICE simulation results show that, for a dual-rail datapath, the new sequencers allow roughly twice the throughput of non-concurrent sequencers. After voltage scaling, energy dissipation of the system is reduced by a factor of 2.5. Similar results are obtained for a single-rail system.

- The introduction of the use of pulses for efficient inter-module synchronization. The idea is complemented with the definition of a pulse-mode handshake protocol and the characterization of Pulse-Burst Operation (PBO), an important extension to traditional pulse-mode operation.

- The design of a basic set of macromodules, that efficiently implement control operations such as sequencing, selection, iteration, concurrency control, resource sharing, and arbitration. Modules for interfacing pulse-mode circuits with traditional 2-phase and 4-phase circuits are also included in the set.

  The modules are of an order of complexity that is very well suited to design by flow table based techniques. Timing problems corresponding to critical races, combinational hazards and essential hazards are constrained within individual modules and can be dealt with very effectively.

- The design of a large, relevant system, a packet switch, in which control overhead has a large impact on performance, stressing the need for an efficient design. The Macromodular packet switch demonstrates the viability of pulse-mode macromodules to implement complex, high-performance systems. The switch organization, its asynchronous operation, and low control overhead introduced by pulse-mode macromodules result in a design that can handle 2.4 times the target throughput of 155 Mbits/Sec. Also, the switch is characterized by very low input-to-output latency (over 10 times shorter than a comparable synchronous implementation).

Although further work is needed, our results suggest that pulse-mode macromodules can keep control overhead low without introducing complex, unsafe timing considerations, two necessary conditions to achieve robust, high-performance systems.

Bibliography

[1] M. Afghahi and J. Yuan. Double edge-triggered D-flip-flops for high-speed CMOS circuits. *IEEE J. of Solid-State Circuits*, 26(8):1168–1170, August 1991.

[2] Aaron Ashkinazy, Doug Edwards, Craig Farnsworth, Gary Gendel, and Shiv Sikand. Tools for validating asynchronous digital circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 12–21. IEEE Computer Society Press, November 1994.

[3] A. Bailey. 2N sequencers: Reset and performance. *Personal communication*, 1995.

[4] A. Bailey and M. Josephs. Sequencer circuits for VLSI programming. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 82–90. IEEE Computer Society Press, May 1995.

[5] W. J. Bainbridge and S. B. Furber. MARBLE a proposed asynchronous system level macrocell bus. In S. B. Furber and A. V. Yakovlev, editors, *Proceedings of the Second UK Asynchronous Forum*, pages 1–5. University of Newcastle, Newcastle upon Tyne, England, July 1997.

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

[7] Janusz A. Brzozowski and Jo C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[8] Wolfgang O. Budde, Hans-Georg Keller, Hans-Jürgen Reumerman, and Paul Van de Weil. An asynchronous, high-speed packet switching component. *IEEE Design & Test of Computers*, 11(2):33–42, Summer 1994.

[9] J. Calvo, J. I. Acha, and M. Valencia. Asynchronous modular arbiter. *IEEE Transactions on Computers*, C-35(1):67–70, January 1986.

[10] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. Optimizing power using transformations. *IEEE Trans. on Computer-Aided Design*, 14(1):12–31, January 1995.

[11] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE J. of Solid-State Circuits*, 27(4):473–484, April 1992.

[12] 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.

[13] T. I. Chappell, B. A. Chappell, S. E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, and R. L. Franch. A 2-ns cycle, 3.8-ns access 512-Kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE J. of Solid-State Circuits*, 26(11):1577–1585, November 1991.

[14] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[15] William J. Dally and Charles L. Seitz. The Torus routing chip. *Distributed Computing*, 1(14):187–196, April 1986.

[16] Paul Day and J. Viv Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[17] Martin de Prycker. *Asynchronous Transfer Mode.* Ellis Horwood, Chichester, England, second edition, 1993.

[18] Mark Dean, Ted Williams, and David Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 55–70. MIT Press, 1991.

[19] Mark E. Dean. *STRiP: A Self-Timed RISC Processor Architecture.* PhD thesis, Stanford University, 1992.

[20] W. E. Denzel, A. P. J. Engbersen, I. Iliadis, and G. Karlsson. A highly modular packet switch for Gb/s rates. In *Proc. International Switching Symposium*, volume 2, pages 236–240, Yokohama, October 1992.

[21] Harry J.R. Dutton and Peter Lenhard. *Asynchronous Transfer Mode (ATM).* Prentice-Hall, Upper Saddle River, NJ, second edition, 1995.

[22] 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, *Proc. Working Conf. on Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 137–147. Elsevier Science Publishers, 1993.

[23] Jo C. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Science of Computer Programming*, 18(3):223–245, June 1992.

[24] 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.

[25] C. Farnsworth, D. A. Edwards, J. Liu, and S. S. Sikand. A hybrid asynchronous system design environment. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 91–98. IEEE Computer Society Press, May 1995.

[26] C. Farnsworth, D. A. Edwards, and S. S. 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. IEEE Computer Society Press, November 1994.

[27] M. Favalli and L. Benini. Analysis of glitch power dissipation in CMOS ICs. In *Proc. International Symposium on Low Power Design*, pages 123–128, 1995.

[28] Charles M. Flaig. VLSI Mesh routing system. Technical Report 5241:TR:87, Computer Science Department, California Institute of Technology, 1987.

[29] S. Furber. Computing without clocks: Micropipelining the ARM processor. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 211–262. Springer-Verlag, 1995.

[30] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[31] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 11–16. IEEE Computer Society Press, March 1996.

[32] J. D. Garside, S. Temple, and R. Mehra. The AMULET2e cache system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–217. IEEE Computer Society Press, March 1996.

[33] G. Gopalakrishnan, P. Kudva, and E. Brunvand. Peephole optimization of asynchronous macromodule networks. In *Proc. International Conf. Computer Design (ICCD)*, pages 442–446. IEEE Computer Society Press, October 1994.

[34] Ruud A. Haring, Mark S. Milshtein, Terry I. Chappell, Sang H. Dhong, and Barbara A. Chappell. Self resetting logic register and incrementer. In *Proc. 1996 Symposium on VLSI Circuits*, pages 18–19. IEEE Computer Society Press, 1996.

[35] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach (2nd edition)*. Morgan Kaufmann, 1996.

[36] I. Iliadis and W. E. Denzel. Analysis of packet switches with input and output queuing. *IEEE Transactions on Communications*, 41(5):731–740, May 1993.

[37] Mark B. Josephs, Rudolf H. Mak, Jan Tijmen Udding, Tom Verhoeff, and Jelio T. Yantchev. High-level design of an asynchronous packet-routing chip. In Jørgen Staunstrup and Robin Sharp, editors, *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 261–274. Elsevier Science Publishers, 1992.

[38] Mark B. Josephs and Jelio T. Yantchev. CMOS design of the tree arbiter element. *IEEE Transactions on VLSI Systems*, 4(4):472–476, December 1996.

[39] K. Kagotani and T. Nanya. Performance enhancement of two-phase quasi-delay-insensitive circuits. *Systems and Computers in Japan*, 27(5):39–46, May 1996.

[40] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.

[41] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 233–243. IEEE Computer Society Press, March 1996.

[42] J.-Y. Le Boudec, E. Port, and H. L. Truong. Flight of the Falcon (ATM). *IEEE Communications Magazine*, 31(2):50–56, February 1993.

[43] Masaaki Maezawa, Itaru Kurosawa, Yoshio Kameda, and Takashi Nanya. Pulse-driven dual-rail logic gate family based on rapid single-flux-quantum (RSFQ) devices for asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 134–142. IEEE Computer Society Press, March 1996.

[44] Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[45] A. 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, Reading MA, 1990.

[46] A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[47] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.

[48] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[49] David May and Peter Thompson. Transputers and routers. In T. L. Kuniis and D. May, editors, *Transputers/OCCAM*. IOS Press, 1990.

[50] Edward J. McCluskey. *Logic design principles: with emphasis on testable semicustom circuits*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[51] Nick McKeown, Martin Izzard, Adisak Mekkittikul, William Ellersick, and Mark Horowitz. Tiny Tera: A packet switch core. *IEEE Micro*, pages 26–33, January/February 1997.

[52] R. E. Miller. *Sequential Circuits and Machines*, volume 2 of *Switching Theory*. John Wiley & Sons, New York, 1965.

[53] Nader Mirfakhraei. Design of a CMOS buffered switch for gigabit ATM switching network. *IEEE J. of Solid-State Circuits*, 30(1):11–18, January 1995.

[54] Pedro A. Molina and Peter Y. K. Cheung. A quasi delay-insensitive bus proposal for asynchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–139. IEEE Computer Society Press, April 1997.

[55] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[56] Charles E. Molnar, Ian W. Jones, William S. Coates, and Jon K. Lexau. A FIFO ring performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, April 1997.

[57] E. Mussol and J. Cortadella. Low-power array multipliers with transition-retaining barriers. In *Proc. International Workshop on Power, Timing Modeling Optimization and Simulation*, pages 227–238. OFFIS Press, October 1995.

[58] L. W. Nagel and D. O. Pederson. Simulation program with integrated circuit emphasis (SPICE). Technical Report ERL-M383, Electronics Research Lab., University of California, Berkeley, April 1983.

[59] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, Summer 1994.

[60] Vinod Narayanan, Barbara A. Chappel, and Bruce M. Fleischer. Static timing analysis for self resetting circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 119–126. IEEE Computer Society Press, 1996.

[61] L. S. Nielsen, C. Niessen, J. Sparsø, and K. van Berkel. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[62] L. S. Nielsen and J. Sparsø. A low-power asynchronous data-path for a FIR filter bank. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 197–207. IEEE Computer Society Press, March 1996.

[63] S. M. Nowick and B. Coates. UCLOCK: Automated design of high-performance asychronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 434–441. IEEE Computer Society Press, October 1994.

[64] S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[65] S. M. Nowick, K. Y. Yun, P. A. Beerel, and A. E. Dooply. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210–223. IEEE Computer Society Press, April 1997.

[66] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[67] Steven M. Nowick, Mark E. Dean, David L. Dill, and Mark Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *Integration, the VLSI journal*, 15(3):241–262, October 1993.

[68] Severo M. Ornstein, Mishell J. Stucki, and Wesley A. Clark. A functional description of macromodules. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 337–355, Atlantic City, NJ, 1967. Academic Press.

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

[70] R. C. Pearce, J. A. Field, and W. D. Little. Asynchronous arbiter module. *IEEE Transactions on Computers*, 24:931–932, September 1975.

[71] A. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[72] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 53–62, May 1995.

[73] L. A. Plana and S. M. Nowick. Concurrency-oriented optimization for low-power asynchronous systems. In *Proc. International Symposium on Low Power Electronics and Design*, pages 151–156, August 1996.

[74] L. A. Plana and S. M. Nowick. Architectural optimization for low-power non-pipelined asynchronous systems. *IEEE Transactions on VLSI Systems*, 6(1):56–65, March 1998.

[75] L. A. Plana and S. H. Unger. Pulse-mode macromodular systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 348–353. IEEE Computer Society Press, October 1998.

[76] Pierre Plaza, Luis A. Merayo, Juan Carlos Díaz, and José Luis Conesa. A 2.5 Gb/s ATM switch chip set. *IEEE Transactions on VLSI Systems*, 4(3):405–416, September 1996.

[77] W. W. Plummer. Asynchronous arbiters. *IEEE Transactions on Computers*, 21(1):37–42, January 1972.

[78] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[79] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, MA, 1980.

[80] C. L. Seitz and W. K. Su. A family of routing and communication chips based on the Mosaic. In G. Borriello and C. Ebeling, editors, *Proc. Research on Integrated Systems*, pages 320–337, Cambrigde, MA, 1993. MIT Press.

[81] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980.

[82] Mishell J. Stucki, Severo M. Ornstein, and Wesley A. Clark. Logical design of macromodules. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 357–364, Atlantic City, NJ, 1967. Academic Press.

[83] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[84] I. E. Sutherland, C. E. Molnar, R. F. Sproull, and J. C. Mudge. The TRIMOSBUS. In C. L. Seitz, editor, *Proceedings of the First Caltech Conference on Very Large Scale Integration*, pages 395–427, January 1979.

[85] Fouad A. Tobagi. Fast packet switch architectures for broadband integrated services digital networks. *Proceedings of the IEEE*, 78(1):133–167, January 1990.

[86] S. H. Unger. Computers without clocks. Technical Report CUCS-011-92, Columbia University, June 1992.

[87] S. H. Unger. A building block approach to unclocked systems. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 339–348. IEEE Computer Society Press, January 1993.

[88] Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, 1969.

[89] Stephen H. Unger. Double-edge-triggered flip-flops. *IEEE Transactions on Computers*, C-30(6):447–451, June 1981.

[90] Stephen H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.

[91] Stephen H. Unger. *The Essence of Logic Circuits.* IEEE Press, New York, second edition, 1997.

[92] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[93] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 72–79, 1995.

[94] K. van Berkel and M. Rem. VLSI programming of asynchronous circuits for low power. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 152–210. Springer-Verlag, 1995.

[95] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[96] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming.* Cambridge University Press, 1993.

[97] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[98] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[99] Daniel Weil, Alain Botta, Alain Chemarin, Gallay Philippe, Jacques Majos, and Michel Servel. A 16 x 6222 mb/s ATM switch: PRELUDE switch architecture integrated into a 6-million transistor monochip. *IEEE J. of Solid-State Circuits*, 32(7):1108–1114, July 1997.

[100] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE J. of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[101] Alexandre Yakovlev, Alexei Petrov, and Luciano Lavagno. A low-latency asynchronous arbitration circuit. *IEEE Transactions on VLSI Systems*, 2(3):372–377, September 1994.

[102] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev. Low-latency asynchronous FIFO buffers. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 24–31. IEEE Computer Society Press, May 1995.

[103] J. T. Yantchev and I. Nedelchev. Implementation of packet switching devices as delay-insensitive circuits. In G. Borriello and C. Ebeling, editors, *Proc.*

Research on Integrated Systems, pages 276–290, Cambrigde, MA, 1993. MIT Press.

[104] Yu-Shuan Yeh, Michael G. Hluchyj, and Anthony S. Acampora. The Knockout switch: A simple, modular architecture for high-performance packet switching. *IEEE J. on Selected Areas in Communications*, SAC-5(8):1274–1283, October 1987.

[105] 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*, pages 17–28. IEEE Computer Society Press, March 1996.

[106] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.

[107] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 140–153. IEEE Computer Society Press, April 1997.