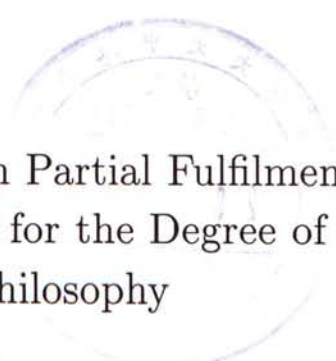


# An Asynchronous Forth Microprocessor

Ping-Ki TSANG

A Thesis Submitted in Partial Fulfilment  
of the Requirements for the Degree of  
Master of Philosophy  
in  
Computer Science & Engineering



Supervised by:  
Prof. Philip LEONG

© The Chinese University of Hong Kong  
Jan 2000

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or whole of the materials in the thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.



## 摘要

低功率、高代碼密度、有好的軟件開發工具、高性能及細小是嵌入系統（特別是以電池作電源）裏的微處理器的必要條件。因此，我們開發了一枚異步微處理器以探索異步邏輯在低電量應用的潛力及示範以異步邏輯來迎合嵌入市場的低成本與低電量的要求。

以堆疊機器作嵌入式微處理器是十分合適的。我們開發了一個堆疊架構的微處理器（MSL16）。MSL16 擁有一個很細小的指令集及面積，我們以現場可編程門陣列（FPGA）實施了一個同步的原型。其後，我們再以馬田合成方法（Martin's synthesis method）實施一個異步微處理器，MSL16A。

我們以高速超大規模集成電路硬體描述語言（VHDL）描述 MSL16，發現它的最高運行時頻為 33MHz，而此同步原型只佔據 175 Xilinx 4000 系列可重構邏輯塊（CLBs）。此設計的異步實施（MSL16A）擁有 66,500 個晶體管，以 1.2 微米互補金屬氧化物半導體（CMOS）技術來實施，我們預期它的平均性能為 33MIPS 而功率則為 95mW。

# An Asynchronous Forth Microprocessor

submitted by

**Ping-Ki TSANG**

for the degree of Master of Philosophy  
at the Chinese University of Hong Kong

## Abstract

Microprocessors for embedded battery powered applications require low power consumption, good code density, good software development tools, high performance, and small area. An asynchronous delay-insensitive implementation of an asynchronous processor, which directly addresses these issues, was developed to explore the potentials of asynchronous logic for low-power applications and to demonstrate the feasibility and practicability of using asynchronous circuits to meet the cost and power constraints of the embedded market.

Stack machines meet all of the requirements of embedded processors and a stack based architecture, called the **minimal instruction set**, small and low power **16** bit microprocessor (MSL16) was developed. A synchronous prototype implementation of the architecture was successfully tested on a field programmable gate array device. The design was then reimplemented using Martin's synthesis method to produce MSL16A, a 16-bit asynchronous microprocessor.

MSL16 was synthesized from a VHDL description and was found to be operational at 33MHz. This synchronous prototype occupies merely 175 Xilinx 4000 series configurable logic blocks (CLBs) which is particularly small for its performance. The asynchronous implementation, MSL16A, contains about 66,500 transistors and the expected performance is 33 MIPS, using 1.2 $\mu$ m CMOS technology,



for a power consumption of 95mW. Comparisons with previous asynchronous and commercial low power microprocessors, are included. When scaled to the same technology, MSL16A performs better in terms of size, power consumption and energy efficiency because of its high code density and simple architecture. This makes MSL16A very competitive for battery powered portable or embedded applications.

# Acknowledgments

I would firstly like to acknowledge the support of my advisor, Prof. Philip Leong and Prof. Tony Lee, for the useful weekly discussions and their inspired leadership of this project. Their insights gave me many ideas, inspiration, and guidance to this work. I warmly thank Prof. Philip Leong for proof reading and commenting on this thesis. Without him, this thesis would not have been completed.

I have received a great deal of support and encouragement from my colleagues in Ho Sing Hang Engineering Building Room 1026, especially, Philip, Ken, Thomas, Oldfield, Fei, Peter and Small Keung. I would like to thank everyone of them for the daily gathering, endless discussions and, above all these, the fun and precious moments we all shared.

Finally, I wholeheartedly thank my mother, Nancy Wang for her lifelong chore of bringing me up, her warmth and invaluable lifelong support.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgments</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Aims . . . . .	1
1.2 Contributions . . . . .	3
1.3 Overview of the Thesis . . . . .	4
<b>2 Asynchronous Logic</b>	<b>6</b>
2.1 Motivation . . . . .	6
2.2 Timing Models . . . . .	9
2.2.1 Fundamental-Mode Model . . . . .	9
2.2.2 Delay-Insensitive Model . . . . .	10
2.2.3 QDI and Speed-Independent Models . . . . .	11
2.3 Asynchronous Signalling Protocols . . . . .	12
2.3.1 2-phase Handshaking Protocol . . . . .	12

2.3.2	4-phase Handshaking Protocol . . . . .	13
2.4	Data Representations . . . . .	14
2.4.1	Dual Rail Coded Data . . . . .	15
2.4.2	Bundled Data . . . . .	15
2.5	Previous Asynchronous Processors . . . . .	16
2.6	Summary . . . . .	20
<b>3</b>	<b>The MSL16 Architecture</b>	<b>21</b>
3.1	RISC Machines . . . . .	21
3.2	Stack Machines . . . . .	23
3.3	Forth and its Applications . . . . .	24
3.4	MSL16 . . . . .	26
3.4.1	Architecture . . . . .	28
3.4.2	Instruction Set . . . . .	30
3.4.3	The Datapath . . . . .	32
3.4.4	Interrupts and Exceptions . . . . .	33
3.4.5	Implementing Forth primitives . . . . .	34
3.4.6	Code Density Estimation . . . . .	34
3.5	Summary . . . . .	36
<b>4</b>	<b>Design Methodology</b>	<b>37</b>
4.1	Basic Notation . . . . .	38

4.2	Specification of MSL16A . . . . .	39
4.3	Decomposition into Concurrent Processes . . . . .	41
4.4	Separation of Control and Datapath . . . . .	45
4.5	Handshaking Expansion . . . . .	45
4.5.1	4-Phase Handshaking Protocol . . . . .	46
4.6	Production-rule Expansion . . . . .	47
4.7	Summary . . . . .	48
<b>5</b>	<b>Implementation</b>	<b>49</b>
5.1	C-element . . . . .	49
5.2	Mutual Exclusion Elements . . . . .	51
5.3	Caltech Asynchronous Synthesis Tools . . . . .	53
5.4	Stack Design . . . . .	54
5.4.1	Eager Stack Control . . . . .	55
5.4.2	Lazy Stack Control . . . . .	56
5.4.3	Eager/Lazy Stack Datapath . . . . .	58
5.4.4	Pointer Stack Control . . . . .	61
5.4.5	Pointer Stack Datapath . . . . .	62
5.5	ALU Design . . . . .	62
5.5.1	The Addition Operation . . . . .	63
5.5.2	Zero-Checker . . . . .	64



5.6	Memory Interface and Tri-state Buffers . . . . .	64
5.7	MSL16A . . . . .	65
5.8	Summary . . . . .	66
<b>6</b>	<b>Results</b>	<b>67</b>
6.1	FPGA based implementation of MSL16 . . . . .	67
6.2	MSL16A . . . . .	69
6.2.1	A Comparison of 3 Stack Designs . . . . .	69
6.2.2	Evaluation of the ALU . . . . .	73
6.2.3	Evaluation of MSL16A . . . . .	74
6.3	Summary . . . . .	81
<b>7</b>	<b>Conclusions</b>	<b>83</b>
7.1	Future Work . . . . .	85
	<b>Bibliography</b>	<b>87</b>
	<b>Publications</b>	<b>95</b>

# List of Tables

2.1	Categorization of Asynchronous Circuits . . . . .	9
2.2	Four states of a channel . . . . .	15
2.3	Characteristics of eight previous asynchronous microprocessors . .	16
3.1	The MSL16 instruction set . . . . .	31
3.2	Translation of Forth primitives . . . . .	34
3.3	Code Density Calculation . . . . .	35
5.1	Truth table of a C-element . . . . .	51
5.2	Carry Output of a Full Adder . . . . .	63
6.1	Simulated cycle time for different Stack Design . . . . .	70
6.2	Power-Delay Product Comparison . . . . .	71
6.3	Size of a single 16-bit stack element(automatically generated) . . .	72
6.4	Performance of the ALU . . . . .	73
6.5	MSL16A chip summary . . . . .	77
6.6	Characteristics of MSL16A, ASPRO-216 and TITAC-2 . . . . .	79

6.7 Scaled performance of MSL16A, ASPRO-216 and TITAC-2 . . . 79

# List of Figures

2.1	Fundamental-mode circuit structure . . . . .	10
2.2	CMOS inverter . . . . .	11
2.3	2-phase handshaking protocol . . . . .	13
2.4	4-phase handshaking protocol . . . . .	14
2.5	Data transfer via (a)dual-rail encoding and (b)bundled data . . . . .	14
3.1	Instruction format for MSL16 . . . . .	30
3.2	The datapath of MSL16 . . . . .	32
4.1	Bare Communication on Channel $E$ . . . . .	46
5.1	The C-element . . . . .	50
5.2	A Simple Arbitration System . . . . .	51
5.3	A basic arbitration circuit . . . . .	52
5.4	The Stack . . . . .	55
5.5	Implementation of the Eager Stack's control process . . . . .	57
5.6	Implementation of the Lazy Stack's control process . . . . .	59

5.7 Implementation of communication . . . . . 60

5.8 Dual-rail to single-rail and single-rail to dual-rail converters . . . . 64

5.9 The datapath of MSL16A . . . . . 65

6.1 MSL16 prototype board . . . . . 68

6.2 IRSIM simulation of test program (Part 1) . . . . . 76

6.3 IRSIM simulation of test program (Part 2) . . . . . 76

6.4 MSL16A chip image . . . . . 78



# Chapter 1

## Introduction

### 1.1 Motivation and Aims

Driven by the growing market for portable battery operated computation devices, performance was no longer the single most important feature of a microprocessor. Today, many embedded applications employ a microprocessor which has requirements of being low power and small area with performance merely a secondary issue. Power efficiency is becoming increasingly important as portable systems are becoming physically smaller and battery weight is becoming more critical. Longer battery life can only be obtained by improving the capacity of the battery or by optimizing the power efficiency of a portable system. The advancement of battery technology is slow, digital designers must address this issue by lowering the power requirements of portable devices.

Asynchronous designs are believed to be ideal for low power applications as they only dissipate power when and where they are active. Previous work has shown that the clock power can be twice the logic power for static logic and about three times the logic power for dynamic logic [1]. Asynchronous circuits do not require global synchrony and thus eliminate the need for global clocks. Moreover, the handshaking protocol of asynchronous circuits removes spurious transitions

and each transition has its own meaning which saves power by nature. Recent research has demonstrated that asynchronous circuits techniques have matured and implementations of asynchronous processors have been reported [2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

Code density is crucial as it relates directly to the requirements of cache and memory sizes. The power consumption in memory and related parts of a system is inversely proportional to the code density of processor embedded in it. A processor with a higher code density means it requires a smaller cache and memory to run at equivalent performance than a lower code density processor. It has been shown that a processor does not dominate a system's total power consumption, about 50% is dissipated by the memory [12]. In addition, large caches can occupy 90% of the total chip area and dissipate about 43% of the total chip power [13]. As a result, the code density of a processor must also be optimized for low power applications.

The aim of this thesis was to address these issues in low power system design by developing a **minimal instruction set, small and low power 16 bit microprocessor (MSL16 [14])**, which was an architecture having desirable features like good performance, high code density and small area. High code density and small area were achieved by utilizing a stack architecture with a minimal instruction set, simple datapath and control. It was designed to directly execute Forth, which is a stack based portable integrated programming environment, operating system and programming language having code density typically higher than that of C or assembly language. Forth has been successfully used in many well known embedded applications [15, 16, 17, 18] and several commercial microprocessors have been designed to run Forth [19, 20]. A synchronous prototype implementation of the MSL16 architecture was successfully tested on a field programmable gate array device, which was found to be small, fast and power efficient.

An asynchronous re-implementation of the MSL16 microprocessor, called



MSL16A, has been developed to investigate the potential advantages that asynchronous designs may enjoy, namely average-case performance instead of worst-case performance and low power consumption. MSL16A was also developed to demonstrate the feasibility and practicability of using asynchronous circuits to meet the cost and power constraints in low power embedded applications.

## 1.2 Contributions

This thesis presents an architecture, called MSL16, which was developed to directly execute the Forth language. MSL16 (which stands for **m**inimal instruction set, **s**mall and **l**ow power **16** bit microprocessor) was designed to support the development of complex software and designed to be included as a coprocessor inside an embedded system. MSL16 has a minimal instruction set of 17 instructions, a 16 bit datapath and very simple control and datapath. This architecture has desirable features like high speed, small number of gates and high code density. Code density is critical in power efficiency as memory and large caches may consume up to half of the total system power [12, 13]. The above features lead to an FPGA implementation that was fast, small and power efficient which is presented in Chapter 6.

Stacks are a fundamental building block in microprocessors, microcontroller and DSPs. However, to the best of the author's knowledge, there have not been any reported quantitative comparisons between different delay-insensitive stack designs and implementations. Three different asynchronous stack designs and implementations, the Eager Stack, the Lazy Stack and the Pointer Stack, were developed in this thesis. The tradeoff among the three different designs were analyzed in terms of performance, power and silicon area to justify the use of the chosen stack design when re-implementing MSL16 asynchronously. The three designs are competitive in size but the Pointer Stack was finally employed as it

was found to be the most energy efficient.

An asynchronous re-implementation of MSL16, called MSL16A, was successfully developed based on Martin's synthesis method [21] to demonstrate the feasibility of using asynchronous circuits in low power embedded systems. The processor was realized both by manual layout and using the Caltech Asynchronous Synthesis Tools (CAST). The work described in this thesis includes the development of this asynchronous re-implementation and its performance estimation. MSL16A was compared with two other previous asynchronous microprocessors, ASPRO-216 [22] and TITAC-2 [9], as well as the commercially available synchronous StrongARM 110 [13]. When scaled to the same technology, MSL16A was found to be smaller, dissipates less power and was more energy efficient than the other designs.

### 1.3 Overview of the Thesis

The thesis is arranged as follows:

Chapter 2 considers asynchronous logic, the potential benefits asynchronous designs can bring, the signalling protocols and different data representation techniques. A brief description of different asynchronous design styles and previous asynchronous processors are also presented. Chapter 3 gives a brief introduction to stack machines and the programming language Forth which explains why they are particularly suitable to embedded applications, followed by a presentation of architecture of MSL16. The instruction set architecture and its datapath components are also examined, explaining how the architecture of MSL16 improves the code density of Forth programs which can greatly enhance the power efficiency of a low power system.

Chapter 4 describes Martin's synthesis method which was the design method-

---

ology used for MSL16A. Chapter 5 describes the implementation of MSL16A. The stack design is critical for the overall performance of the processor, and three different asynchronous stack designs were compared. Evaluations of the three stack designs and the ALU, together with other interesting implementation issues are discussed. In Chapter 6, results from simulation of the processor are presented. The synchronous FPGA based implementation is first presented, followed by the evaluation procedure and the performance comparison of MSL16A with other processors. In Chapter 7, concluding remarks about this work are presented.



# Chapter 2

## Asynchronous Logic

MSL16A uses asynchronous logic as this approach is believed to have benefits over standard synchronous designs in terms of energy efficiency, speed and robustness [23]. The first section of this chapter explains the motivation for using asynchronous logic in low power system. Different asynchronous design styles are presented in the subsequent section. The final section reviews eight recent asynchronous microprocessors.

As a complete treatment of asynchronous design styles is beyond the scope of this thesis, only a brief description of current asynchronous design styles is presented here. The design methodology adopted in building MSL16A, Martin's synthesis method, will be detailed in Chapter 4. More in-depth surveys on asynchronous design styles can be found in [24, 23]

### 2.1 Motivation

The MSL16A processor uses asynchronous circuit because of the following potential benefits.

**No clock skew** Clock skew is the difference in arrival times of the global clock

signal at different parts of the system. Synchronous circuits usually rely on an externally generated clock signal which is distributed to all of its circuit elements. The clock period is dependent on the maximum clock skew. With today's VLSI circuits exceeding 15mm per side, several nanosecond of clock skew is not unusual. With a fixed 5 nanosecond (200MHz) clock period, several nanoseconds of clock skew is disastrous.

Clock deskewing methods are available (like the balanced H-tree) but the costs are extremely high. The designer of the DEC<sup>1</sup> ALPHA took another approach [25] but the result is a clock driver chip that occupies about 10% of the chip area and consumes more than 40% of the total power generated by the chip. The price to pay for keeping the clock skew under control is very high. Asynchronous circuits, by definition, do not have a globally distributed clock, and the clock skew problem is eliminated automatically.

**Low power** Asynchronous circuits consumes power only in areas involved in computation. Idle components waste negligible power. The global clock in standard synchronous circuit keeps on toggling all the time and power is dissipated along the long clock lines. Previous work [1] has shown that clock power is about twice the logic power for static logic and about three times the logic power for dynamic logic. Some power management mechanisms can shut down the idle parts of advanced synchronous systems with extra circuitry but asynchronous system have this efficiency by nature.

In addition, in response to a clock edge, a number of signals in a synchronous system change several times before reaching a stable value. These spurious transitions do not cause the circuit to fail but every transition, useful or not, consumes power. On the other hand, every transition in an asynchronous circuit is meaningful. Any glitches will cause the circuit to malfunction. They generally make fewer transitions and hence consume

---

<sup>1</sup>Digital Equipment Corporation

less power.

**Average-case performance** A fixed clock period is chosen depending on the worst-case timing analysis of a synchronous circuit. However, worst-case situation rarely occurs but still it has to be clocked so that the rare worst-case condition is accommodated. Asynchronous circuits take advantage of the best- and average-case computation situations as they sense when a computation has completed. Substantial savings can be gained for circuits like ripple-carry adders where the worst-case delay is much worse than the average-case delay.

**Robustness** Asynchronous circuits operates over a wide range of temperature and supply voltage. They are more tolerant to variations in physical or electrical parameters. They adapt to those variations automatically as they sense computation completion, and will run as quickly as the current physical and electrical properties allow.

Circuitry which guarantee correct mutual exclusion of independent signals are subject to metastability [26], which a system can remain in an unstable equilibrium state for an unbounded amount of time [27]. There is a chance for such a mutual exclusion circuit to fail in synchronous systems as all elements have to exhibit bounded response time. Asynchronous systems can wait for an arbitrary long time, allowing robust mutual exclusion.

Along with the advantages described above, it is worth noting that asynchronous circuits have several problems as well. Asynchronous circuits are more difficult to design as hazards must not be introduced in the circuit to avoid incorrect results. Moreover, the signalling protocol and dual-rail data representation in asynchronous systems work against energy efficiency and silicon area. Asynchronous circuits require extra time for synchronization, thus increasing the average-case delay. As a result, further investigation is needed to see to what



extent the potential benefits of asynchronous circuits can be realized.

## 2.2 Timing Models

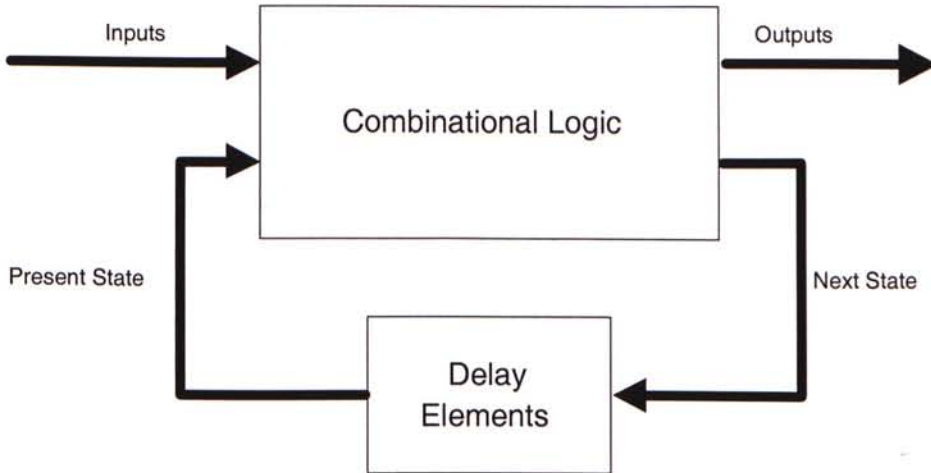
Asynchronous design styles can be categorized by the timing models they assume. As shown in Table 2.1, they can be divided into four groups according to the assumptions made on gate and wire delays. The timing models will be discussed in this section.

Model	Gate delay assumption	Wire delay assumption
Fundamental-mode	Bounded	Bounded
Delay-insensitive	Unbounded	Unbounded
Speed-independent	Unbounded	Negligible
Quasi-delay-insensitive (QDI)	Unbounded	Isochronic fork

**Table 2.1:** Categorization of Asynchronous Circuits

### 2.2.1 Fundamental-Mode Model

The fundamental-mode model, also called the Huffman model [28, 29], assumes that both gate and wire delays are bounded. In this model, the asynchronous circuit is decomposed into two parts, the combinational logic part and the feedback part. The fundamental-mode model asynchronous circuit structure is shown in Figure 2.1. The environment must be able to control the timings of inputs such that input transitions only occur when the circuit is in either the present state or the next state. This means that the next input transition cannot take place until the entire system settles into a stable state.



**Figure 2.1:** Fundamental-mode circuit structure

### 2.2.2 Delay-Insensitive Model

The delay-insensitive model assumes that both gate and wire delays are unbounded but finite. This model imposes the least restrictions on circuit delays and a delay-insensitive circuit works correctly as long as all gate and wire delays are finite. This is an attractive property for synthesis and testing but it has been proved that the class of delay insensitive circuit is very limited [30]. A simple example in [31] has shown that even the simplest CMOS inverter is delay sensitive. The inverter circuit shown in Figure 2.2 would fail if there is a large delay difference from the input to the pMOS and nMOS transistors as both may turn on at the same time. As a result, it is often assumed that the skew between the wire delays after the forking is less than one gate delay in practice.

Moreover, the delay-insensitive timing model has a great impact on the resulting circuit structure. It is assumed that given enough time a subcircuit will have settled in a bounded-delay model. On the other hand, in a delay-insensitive model, a subcircuit may not be settled even after a long time as delays are unbounded. The recipient must send a signal to inform the sender when it has



received the data. This function relies on the completion detection circuit in the receiver which requires a new way of passing data also. The protocol and data representation techniques will be discussed in Section 2.3.

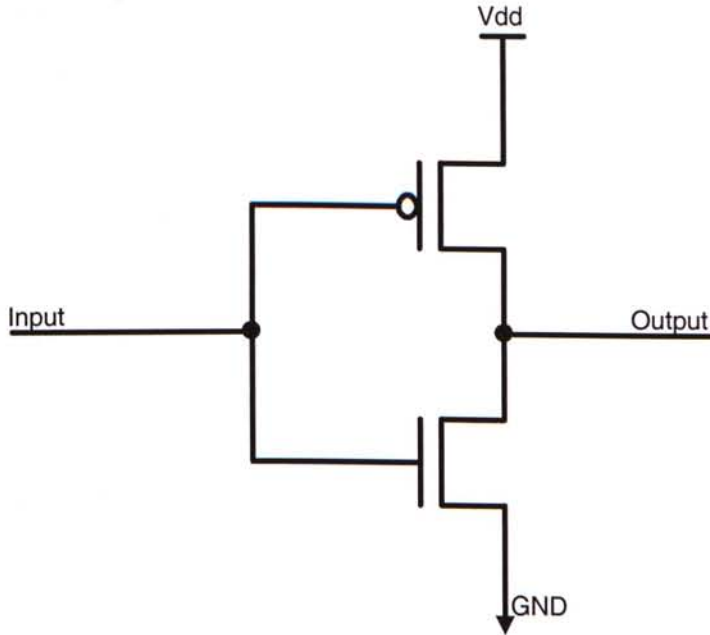


Figure 2.2: CMOS inverter

### 2.2.3 QDI and Speed-Independent Models

These two models differentiate themselves in their choice of compromise to delay insensitivity. The speed-independent model, also called the Muller model, makes the assumption that while gate delays are unbounded but finite, all wire delays are negligible. Quasi-delay-insensitive circuits adopt the delay-insensitive assumption (both wire and gate delays are unbounded but finite) but forks are isochronic. An isochronic fork [21] is a set of interconnecting wires where the difference in delays between destinations is negligible.

While quasi-delay-insensitive and speed-independent models allow more im-

plementation alternatives than the pure delay-insensitive model, the delay assumptions they require can sometimes be difficult to realize in practice. In custom designs delay elements can be added to balance the delay to different fork ends. However, in field-programmable gate arrays and deep submicron technologies, wire delays can often dominate logic delays and the automatic routing tools may not be able to handle the isochronic constraint.

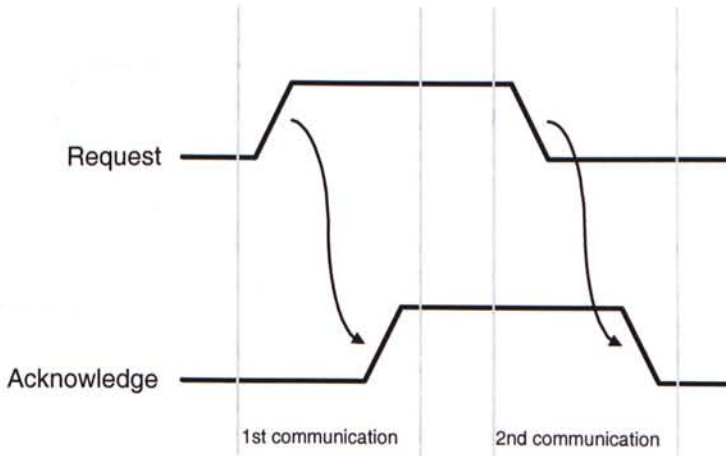
In addition, some parts of a fork may cross a chip boundary in large designs and matching the delays between on-chip and off-chip destinations is almost impossible. To avoid this problem, Martin's synthesis method [21] restricts isochronic forks to small localized areas.

## 2.3 Asynchronous Signalling Protocols

With an unbounded model, communications between blocks in an asynchronous system require some sort of handshaking mechanism. Pairs of wires called *request* and *acknowledge* are generally used to indicate when an action should be initiated and when an operation is completed. Two signalling protocols have been proposed for handshaking using the *request* and *acknowledge* signals. They are classified based on the number of transitions passing through the handshaking wires.

### 2.3.1 2-phase Handshaking Protocol

The 2-phase handshaking protocol is a non return to zero protocol. Figure 2.3 shows an example of 2-phase communication. In a 2-phase communication, the sender makes a single transition on the request wire to initiate the communication. The receiver senses the request, services the request and then responds by making a single transition on the acknowledge wire. Note that only a transition



**Figure 2.3:** 2-phase handshaking protocol

is important and the rising and falling edges are both significant and equivalent.

If the first communication starts with a transition from Low to High, the next communication will start with a transition from High to Low as there is no intermediate recovery stage.

### 2.3.2 4-phase Handshaking Protocol

The 4-phase handshaking protocol is a return to zero protocol and is illustrated in Figure 2.4. The first two phases are active phases while the other two phases are recovery phases. The rising edge of request initiates the communication and the receiver responds by changing the acknowledge wire to a High level also. The falling edges of request and acknowledge wires are inserted to return the *request* and *acknowledge* signals to a logical Low level and indicates a successful communication.

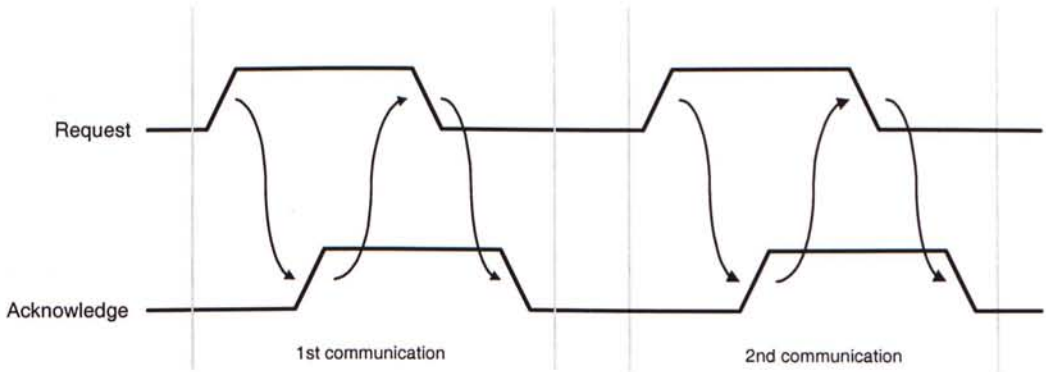


Figure 2.4: 4-phase handshaking protocol

## 2.4 Data Representations

In asynchronous designs, the transfer of data must be handled carefully as there is no global clock signal to indicate when a computation can start or when it is completed. Additional wires must be used for synchronization. The two commonly used approaches are described below.

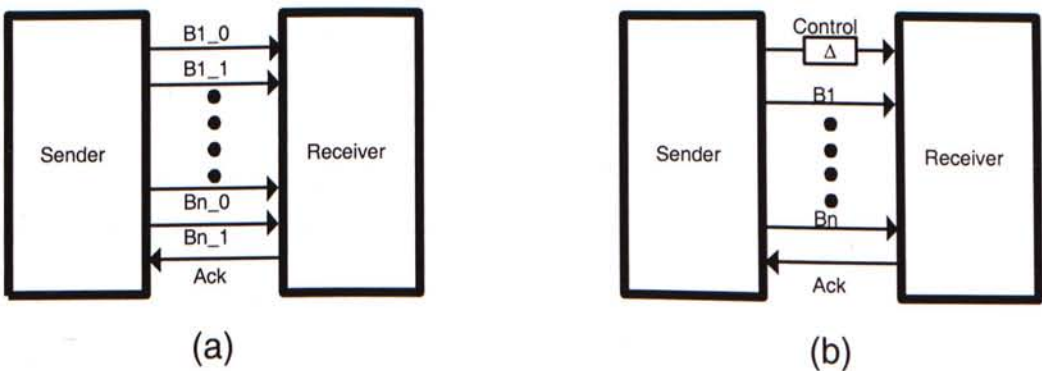


Figure 2.5: Data transfer via (a) dual-rail encoding and (b) bundled data



### 2.4.1 Dual Rail Coded Data

By using a single wire per bit to transfer data, the receiver will not be able to differentiate between a wire that will not change and one that has not yet changed. Thus, every bit of information needs to be encoded with two separate wires representing a logic '1' and a logic '0'. Initially, both of them are zero. To transmit a '1' (refer to Figure 2.5a), the wire representing logic '1' is raised and vice versa. It follows that a dual-rail encoded binary communication channel can have four states (see Table 2.2). Hence, it is possible to detect when a data bit is valid.

wire_1	wire_2	meaning
0	0	idle
1	0	a valid one
0	1	a valid zero
1	1	invalid state

**Table 2.2:** Four states of a channel

Unlike bundled data, the timing information is mixed with the data. The main advantage of this approach is that the resulting circuits are delay-insensitive but at the cost of increasing the number of wires and chip area.

### 2.4.2 Bundled Data

Bundled data allows a single wire for each data bit and associates one control wire to indicate the validity of all of the data (as illustrated in Figure 2.5b). The delay in the extra control wire must be guaranteed to be longer than that of each of the data wires. This bundling constraint requires that a transition on the control wire does not occur until after the data lines are stable at the receiver.

The main advantage of this technique is that standard datapath components or cell libraries can be used. Bundled data allows fewer wires to be used but



violates the delay-insensitive model as it is inherently delay-bounded. It can save considerable area at the cost of meeting the bundling constraints.

## 2.5 Previous Asynchronous Processors

Many asynchronous microprocessors have been previously implemented or proposed [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. This section briefly describes the design styles and architectures of eight asynchronous processors which are summarized in Table 2.3.

Processor	Design Style	Instruction Set	Organization
CAP	4-phase, dual rail, quasi delay insensitive	Own 16-bit RISC like	Fetch-execute pipeline
AMULET1	2-phase, bundled data	ARM	Pipelined, no forwarding
AMULET2e	4-phase, bundled data	ARM	Pipelined, forwarding
ECSTAC	Fundamental mode	Own, variable length	Pipelined, no forwarding
TITAC	4-phase,dual rail quasi delay insensitive	Own 8-bit	No pipelining
TITAC-2	4-phase dual rail scalable delay insensitive	MIPS R2000	pipelined, forwarding
FAM	4-phase, dual rail quasi delay insensitive	Own RISC like	Pipelined
ASPRO-216	4-phase, dual rail quasi delay insensitive	16-bit RISC like 64 custom instructions	pipelined no forwarding

**Table 2.3:** Characteristics of eight previous asynchronous microprocessors

### The Caltech Asynchronous Processor (CAP)

Professor Alain Martin at Caltech built the first asynchronous processor in 1989 [2]. The processor is delay insensitive and its datapath is dual rail en-

coded. It was built based on Martin's synthesis method [21]. The processor was described using a language that is based on Hoare's model of *Communicating Sequential Processes (CSP)* [32] and then compiled to a circuit by means of program transformations.

The instruction set is a conventional 16-bit-word of the RISC-like load/store type with 16 registers. It consists of several concurrent processes responsible for instruction fetch, manipulating the program counter, ALU operation, memory access...etc. The processor was implemented in a 1.6 $\mu$ m CMOS process. In room temperature, it operates at 18 millions of instructions per second(MIPS) at 5V. The circuit is functional at very low supply voltage with optimum power-delay product at around 2V. Its performance reaches 30MIPS in liquid nitrogen at 77K.

## AMULET1 and AMULET2e

The AMULET group led by Steve Furber at the University of Manchester built the first asynchronous implementation of the ARM instruction set, AMULET1 [33]. AMULET2e [8] is a similar processor which achieved full code-compatibility with the ARM6 processor.

The ARM instruction set was designed for synchronous processors [34] with some of its features only convenient in the synchronous system. AMULET1 and AMULET2 implements this instruction set completely including the difficult areas of interrupts and exceptions.

AMULET1 was designed using a 2-phase bundled data design style based on [35] It has a 5-stage pipeline without result forwarding. A locking mechanism is used to stall the register read stage until their operands have been written by previous instructions. It also permits out of order completion of load instruction relative to normal ALU instructions. AMULET1 was implemented in a 1.2 $\mu$ m



CMOS process with a peak performance of 5.3MIPS.

AMULET2e was designed using a 4-phase bundled design style as the CMOS implementation of 2-phase control elements in AMULET1 was inefficient. The pipeline of AMULET2e is slightly shorter and employs both branch prediction and result forwarding to increase the pipeline utilization. AMULET2e was implemented in a  $0.5\mu\text{m}$  3-layer metal CMOS process and its performance reaches 40MIPS at 3.3V.

## ECSTAC

ECSTAC is an asynchronous processor designed by researchers at the University of Adelaide [7]. It was implemented using fundamental mode circuits. It is deeply pipelined with 8-bit registers and ALU. It has a complex variable length instruction format with a total of 113 distinct instruction types including mode variants. There is no result forwarding scheme in the datapath and, like AMULET1, a register locking mechanism is employed to stall instructions until their operands are available. The anticipated throughput was 40MIPS.

The designers reported that the variable length instructions and the mismatch between the datapath width and address size made the processor more complex and slower.

## TITAC and TITAC-2

TITAC is a simple asynchronous microprocessor built by a group at the Tokyo Institute of Technology [5]. TITAC is an asynchronous version of an 8-bit von Neumann microprocessor based on the quasi delay insensitive timing model. It has a straightforward architecture without any pipelining and a simple accumulator-based instruction set. The datapath design uses a dual rail, multi-level AND-OR

scheme with a binary decision diagram for efficient signal generation.

TITAC-2's [9] instruction set is based on the MIPS R2000. It is a 32-bit asynchronous microprocessor with an architecture which is very similar to the MIPS R2000 processor. It has a five stage pipeline and was designed using 4-phase dual rail encoding scheme. It was fabricated in a  $0.5\mu\text{m}$  3-layer metal CMOS process and operates at 52.3MIPS at 3.3V at room temperature. TITAC-2 runs correctly from 1.5V to 6.0V.

## FAM

FAM [11] is a 32-bit dual rail asynchronous processor with RISC like load-store instruction set. The datapath contains 32 general purpose registers, 32-bit ALU, multiplier and 32-bit barrel shifter. It has a four stage pipeline but register read, ALU operation and register write fit in a single stage to eliminate the need for result forwarding. FAM was implemented in  $0.5\mu\text{m}$  CMOS process and its performance is about 300MIPS.

## ASPRO-216

ASPRO-216 [22] is a 16-bit quasi delay insensitive asynchronous microprocessor. It was built based on Martin's synthesis method. It is a scalar processor which issues instructions in-order while completing their instructions out-of-order. The processor includes sixteen 16-bit general purpose registers together with two two distinct on-chip memories, one for data and the other for program. In addition, there are 64 possible custom instruction slots and the processor is heavily pipelined. ASPRO-216 was implemented in a  $0.25\mu\text{m}$  five metal-layer CMOS process and a peak processing rate of 200MIPS was expected.

## **2.6 Summary**

In this chapter, an introduction to asynchronous logic and asynchronous design styles was presented. MSL16A uses asynchronous circuits as this approach is believed to have benefits in terms of energy efficiency, speed and robustness. In addition, asynchronous circuits eliminate the clock skew problem found on synchronous circuits, exhibit average-case performance and allow robust mutual exclusion. MSL16A adopts the quasi-delay-insensitive timing model which assumes that wire and gate delays are unbounded but finite while forks are isochronic. Handshaking protocols and data representation techniques generally used in asynchronous systems were discussed. Dual-rail coded data are required to satisfy the unbounded delay timing assumption while bundled data are simpler and smaller but violates the delay-insensitive model. Finally, a review of earlier work on asynchronous microprocessors was also presented.



# Chapter 3

## The MSL16 Architecture

This chapter presents the architecture of the MSL16 processor. MSL16A shares the same architecture with different design methodology and implementation techniques. The first section describes RISC machines. The next two sections give a brief introduction to stack machines and the Forth programming language, explaining why they are ideal for low power systems. The pipelining of MSL16, its instruction format, its instruction set and the datapath will be discussed in the last section. This section also presents how MSL16 was designed to meet the tightly constrained power and cost requirements of low power portable applications. The code density of MSL16 are also estimated as code density relates directly to power consumption of a complete system.

### 3.1 RISC Machines

The basic principles behind the original RISC (Reduced Instruction Set Computer) processors are reviewed below:

- A simple instruction set is faster than a complicated one
- Complicated addressing mode for instructions are unnecessary

- A large register file facilitates software
- Let the compiler handles complicated functions
- Simple processors are easy to design

Some of these principles are still valid today but RISC processors tend to be very complicated nowadays, some are even more complicated than their CISC (Complex Instruction Set Computer) counterparts. Pipelining was introduced to increase the throughput but the pipeline still has to be flushed and refilled whenever a branch is taken. Modern RISC architectures follow the principles of making the common case fast, reducing the instruction set to simplify hardware implementation, having a uniform instruction encoding so that it is easily decoded, supporting a small number of addressing modes, using large register files and relying on caches to provide high memory bandwidth [36]. These design criteria were chosen to maximize the performance of the machine with power consumption and chip area being secondary concerns.

Large amount of cache memory is needed to buffer instructions if the core speed of a RISC processor is much faster than the main memory because of its lower code density. The associated cache control circuitry will increase the system complexity and large caches may take up to 90% of the total chip area [13]. Moreover, RISC processors does not utilize cache memory efficiently as program size for RISC machine is usually larger. A large register file is windowed to facilitate subroutine call and return but the large register file will slow down the processor for context switches.

In portable and battery operated applications, the design criteria are different as the system cost and power consumption are highly constrained. There is normally no cache, the amount of memory used should be minimized, and some performance can be sacrificed for code density, which relates directly to the total

power dissipation of a system. These issues have influenced the architecture of MSL16.

## 3.2 Stack Machines

Stack machines are simpler than other CISC or RISC machines since their instruction sets do not need to encode operands. For example, in a RISC machine a 3-operand addition requires the two source and the destination registers to be encoded in the instruction. In a stack machine, however, an addition always operates on the top two elements of the stack and leaves the result on the stack so no operands are required in the instruction encoding. A stack architecture in general achieves a much higher code density than that of a RISC machine with a lower system complexity and smaller program memory requirements [19].

A stack machine can achieve high computational power at a low cost because of its low processor complexity [19]. The cost of a chip relates directly to the number of transistors, a lower complexity processor will lower the total system cost, which is tightly constrained in small embedded systems. An especially favorable application area for stack machines is in real time embedded control applications, where small size, high performance and excellent support for interrupt handling are required [19].

Besides, it is easier to write compilers for stack machines since they have very few exceptional cases and, most of the time, the operands are just the top element of the stack and the **T** (top of stack) register. Writing a compiler can take up a significant amount of development time, and an efficient and error-free compiler is essential for testing the system.



### 3.3 Forth and its Applications

Although the C language appears to be the ubiquitous high level programming language for RISC machines, the development effort for a programming environment including assembler, compiler and operating system is rather large, and the code density is not particularly good. Forth is an obvious language to consider using on a stack machine. This is because Forth forms both an assembly and high level language for a stack processor with two stacks, one for expression evaluation and parameter passing, the other for storing subroutine return addresses. The Forth language basically defines a stack based computer architecture which will be emulated by the stack processors while executing Forth programs [19].

A Forth system is an interactive development environment which usually combines an assembler, stand-alone operating system, interpreter and compiler. A Forth system is typically built upon a small number of primitives, and the higher level routines call the lower level primitives to implement the rest of the system. The system (which bundles the operating system and compiler) is very simple and can be ported in a matter of several weeks, compared to man years of development effort for a reasonable C compiler.

Forth encourages reuse of code, all primitive functions being compiled into subroutines and all high level functions being compiled to lists of addresses (which point either to functions or primitive functions). It has been estimated that Forth machines typically have 2.5 to 8 times better code density than CISC designs and another 1.5 to 2.5 over RISC architectures [37].

Forth is interactive in nature and is widely used in embedded applications. Forth programs work by extending the language to include the functions needed to implement a given application. Unlike languages such as C or Fortran, control instructions (such as conditionals and loops) can also be extended by the user [38]. Enhanced functionality (e.g. object oriented features) can be added



to the language by the users. Forth's functionality is achieved with binary sizes usually measured in kilobytes. As an example, the public domain eForth 1.0 system [39] for the Intel 8086 microprocessor is less than 5K in size and the entire system can be ported to another microprocessor by rewriting the 31 primitive instructions upon which the system is based.

Forth programs are characterized by a high frequency of subroutine calls and returns, these instructions dominating all other operations. For a set of benchmark Forth programs, Koopman found a static frequency of approximately 33% and a dynamic frequency of 22% for call/return instructions [19]. This makes it very important to make the CALL and RETURN instructions as fast and compact as possible.

ANS Forth is an American National Standards Institute (ANSI) standard language [40]. Originally developed for the interactive control of telescopes in observatories using small computers [18], Forth has been successfully used in many well known embedded applications. Some notable examples are that Forth is used in the boot firmware for all Sun (and many other) workstations (IEEE 1275-1994 standard [16]), the Federal Express SuperTracker scanner/terminal [15], and in many space applications [17] such as the Galileo probe, space shuttle and Hopkins Ultraviolet Telescope. Several commercial microprocessors have been designed to run Forth such as Novix NC4016 [19], Harris RTX 2000 [19], Silicon Composers SC32 [19] and the Computer Cowboys MuP21 [20]. These chips achieved high performance, low power consumption and small area using very simple hardware designs.

## 3.4 MSL16

Microprocessors are being used in an extremely diverse range of applications from low cost simple applications to extremely high performance real-time ones. Incorporating a microprocessor enables embedded systems to perform more complex tasks without requiring an external microprocessor. On the other hand, the market for portable computing is growing rapidly as new generations of personal digital assistants (PDAs), intelligent cellular phones and other handheld devices are now available to consumers. These applications are characterized by their high performance computation power requirements within an extremely constrained cost and power budget. Energy efficiency is critical as most portable devices are battery powered. A more energy efficient system will give a longer battery life for the same capacity. A larger battery is highly undesirable since it increases the size of a device, and a smaller device are naturally more favorable.

Traditional embedded processors which match the power and cost limitation cannot deliver the performance required by new applications. Incredible performance improvements have been made at the other end of the processor spectrum, with increased power dissipation and system cost that are not compatible with portable systems. This called for a new class of microprocessor which gives high computation power with small area, low power and energy efficient.

In [14], a microprocessor design and implementation called MSL16 (which stands for **m**inimal instruction set, **s**mall and **l**ow power **16** bit microprocessor) which was developed to directly address these issues was presented. MSL16 has the following features:

- high speed
- small number of gates
- high code density



- a high level language programming language and operating system
- highly customizable for different applications
- portable to different FPGA devices and vendors (for FPGA based implementation)

MSL16 is a MISC (Minimum Instruction Set Computer) architecture. The basic idea behind MISC is simplicity and the principle of simplicity in RISC is strictly enforced. MSL16 has a minimal instruction set of 17 instructions, a 16-bit datapath and very simple control and datapath. Most instructions are 4 bits in length, contributing to a high instruction density and low memory bandwidth requirements. Unlike RISC machines, MSL16 does not require cache memory as the use of a slower memory will not severely degrade the system performance. It has been shown that large caches may take up to 90% of the total chip area and dissipate about 43% of the total chip power [13].

MSL16 is a stack architecture which in general achieves a higher code density than RISC machines. Additionally, MSL16 was designed to execute the programming language “Forth”. Forth machines typically have a higher code density than CISC and RISC designs [37]. Code density relates directly to system power consumption and with battery operated devices such as mobile phones, PDAs and laptop computers becoming increasingly popular, power consumption is becoming increasingly important. In a portable application such as the Berkeley Infopad project [12], the total system’s power consumption (1.2 W) is dominated by the power consumption of a static RAM memory (600 mW) rather than that of the ARM60 microprocessor (120 mW). A processor with higher code density requires less memory to operate without lowering its performance. This implies a processor with higher code density is more power efficient.

Without incorporating any cache inside the processor, having a high code density and simple architecture, MSL16 can be fast and yet very small and low

power. These characteristics tightly match the requirements of the portable market. The MSL16 architecture is presented below, and how it improves the code density of Forth program is detailed in Section 3.4.6. MSL16A is the asynchronous re-implementation of MSL16 which shares the same architecture and it is a small, fast, low power and high code density microprocessor. The design methodology and implementation detail of MSL16A are presented in Chapter 4 and Chapter 5 respectively.

### 3.4.1 Architecture

The architecture of MSL16 is similar to that of MuP21 [20]. The MuP21 is a 20-bit CPU which has 25 5-bit instructions and implemented in 1.2 $\mu$ m CMOS process, uses 7000 CMOS transistors and has a peak execution rate of 100 MIPS. The synchronous prototype of the MSL16 architecture was synthesized from a VHDL description. It was highly portable and was designed to be easily customized for particular applications and/or retargeted for different FPGA devices and vendors. Compared with the MuP21, the MSL16 architecture has 16 4-bit instructions, and when implemented using a Xilinx Inc, 4000 series FPGA, occupies 175 configurable logic blocks (CLBs) at a peak execution rate of 33 MIPS on a 4006E-1 device. The results of this prototype is presented in Chapter 6.

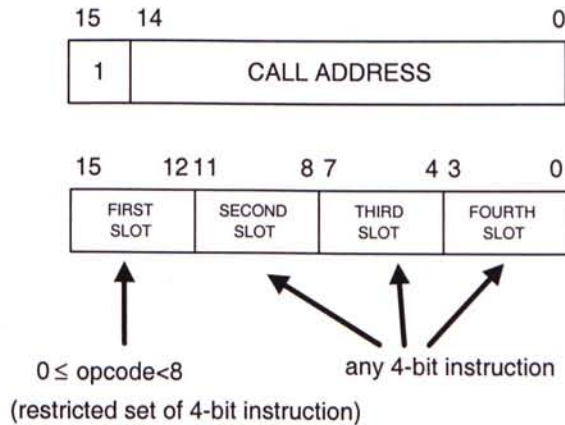
MSL16 is a dual stack machine with 16-bit data and memory buses. The data stack is used for temporary variable storage and subroutine parameter passing, and the return stack is used mainly to hold subroutine return addresses. The data and return stack are implemented internally which allows them to be accessed in parallel with instruction fetches on the memory bus. A two stage FETCH/EXECUTE pipeline is employed so that the following two steps operate in parallel



```
FOREVER
{
    1. Fetch a 16-bit word from memory
       and latch in the instruction
       register (IR)

    2. If the most significant bit of IR set
       {
           CALL instruction - other 15 bits
           form the subroutine address
       }
       else
       {
           IR contains four instructions -
           execute in sequence (first slot
           can only use the lower 3 bits so
           must have  $0 \leq \text{opcode} < 8$ ).
       }
}
```

With the exception of the CALL instruction, MSL16 instructions are encoded with 4 bits, allowing four instructions to be packed in one 16 bit word. A 16-bit instruction fetch generally obtains 4 instructions, excluding those with CALL or LIT instructions, reducing the required memory bandwidth and overall system power consumption. As a result, the effect of starvation of the pipeline on performance is reduced and hence using a memory 4 times slower than the average cycle time would not significantly reduce the performance of the processor. This could also reduce the cost of building an embedded system with MSL16A. A pipeline stage is said to be starved if it is forced to wait for the previous stage



**Figure 3.1:** Instruction format for MSL16

to complete.

The execution speed of MSL16 is high because of its simple instruction set, and a short critical delay path. The two stage pipeline has a low latency so the effect of stalling the pipeline during memory operations and branches is reduced. The top of stack is implemented as a separate register, the **T** register. Operands to the ALU are normally the two top elements of the stack and the result is usually stored in the **T** register. This scheme virtually eliminates the instruction decoding and register fetch process normally required in a RISC machine.

### 3.4.2 Instruction Set

The instruction format used for MSL16 is shown in Figure 3.1. All of the instructions with the exception of CALL and LIT expect their operands to be on the stack. If the most significant bit of the instruction register is set, it is a CALL instruction and the remaining 15 bits form the subroutine address. In other words, the first slot can only contain a limited set of instructions with  $\text{opcode} < 8$ .

If LIT appears in the first or second slot of the IR, its operand will be loaded from the least significant byte of the IR. However, if LIT appears in the final slot, the processor status word containing system flags will be loaded into the T register. To load a full 16-bit literal into the T register, 2 successive LITs, one placed in the first slot and the other in the second slot, followed by an XOR instruction are required. The operand of the first LIT instruction contains the most significant byte of the literal while the operand of the second LIT instruction contains the least significant byte of the literal. The XOR instruction will merge them correctly into a single 16-bit value stored in T.

Opcode	Instruction	Action
0	NOP	no operation
1	AND	$T \leftarrow T \text{ AND } DS$ , pop DS
2	XOR	$T \leftarrow T \text{ XOR } DS$ , pop DS
3	+	$T \leftarrow T + DS$ , pop DS
4	0=	$T \leftarrow -1$ if (T=0) else $T \leftarrow 0$
5	LIT	push T to DS, if LPC = 0, $T \leftarrow \text{LSB}(\text{IR}) \& "00000000"$ if LPC = 1, $T \leftarrow "00000000" \& \text{LSB}(\text{IR})$ if LPC = 3, $T \leftarrow \text{processor status word}$
6	2/	$T \leftarrow T / 2$
7	-	$T \leftarrow DS - T$ , pop DS
8	DUP	push T to DS
9	DROP	pop DS to T
10	GOTO	Jump to T if $T \neq 0$ , pop DS to T
11	R>	push T to DS, pop RS to T
12	>R	push T to RS, pop DS to T
13	@	LOAD mem[T] to T
14	!	STORE T to mem[ds]
15	SWAP	Swap T with DS
MSB=1	CALL	PUSH PC to RS, jump to IR

**Table 3.1:** The MSL16 instruction set

The instruction set of MSL16 is given in Table 3.1. For the synchronous version, instructions involving a memory reference (@ and !), change the flow of execution, (CALL and GOTO) and SWAP take two cycles and the remaining

instructions are single cycle. However, for MSL16A, all instructions are completed in a single cycle with different cycle time and the pipeline will adjust automatically. Some instructions just take longer to complete.

In MSL16A, the fetch and execute stages do not have fixed cycle times. The fetch cycle depends on the speed of the external memory while the execute cycle depends on the instructions executed and their operands as well. For example,  $1 - 1$  (implemented as  $1 + (-1)$ ) will take longer to complete than  $1 + 1$  in the ALU.

### 3.4.3 The Datapath

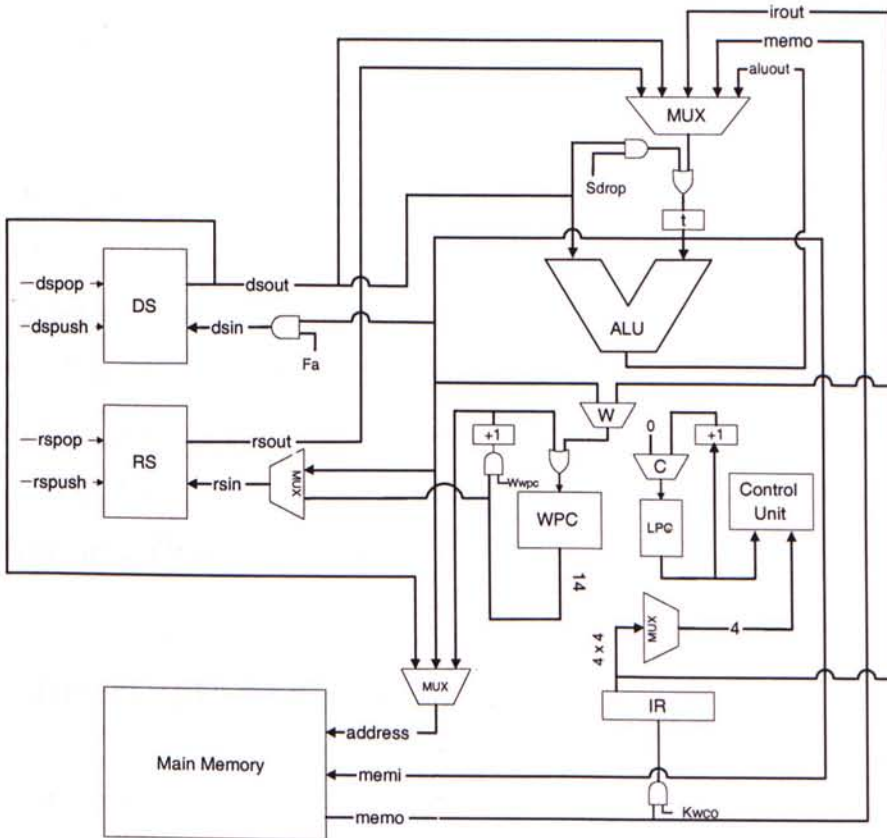


Figure 3.2: The datapath of MSL16



The datapath of MSL16 is shown in Figure 3.2. The main components in the datapath of MSL16 are

- A data stack (DS) for temporary variables and subroutine parameters. The very top element of the stack is stored in a separate register T so that the top two stack elements are available to the ALU.
- A return stack (RS) to store subroutine return addresses
- An instruction register(IR) which holds the four 4-bit instructions to be executed
- A PC (Program Counter) which stores the address of the next instruction
- An IR (Instruction Register) which stores the instruction being executed
- An ALU which takes operands from T and the top element of DS or RS the result will be returned to T.

The data stack and return stack are both  $32 \times 16$  bits in size. A study has shown that a stack depth of 32 is sufficient for most reasonable programs [37]. The PC is actually composed of a 2-bit nibble PC(LPC) and a 14-bit word PC(WPC). The WPC stores the address of the next instruction while the LPC is used to keep track of which of the four instructions in the IR is currently being executed. The LPC will be reset to zero automatically when a LIT, CALL or GOTO instruction is encountered.

### 3.4.4 Interrupts and Exceptions

MSL16 does not currently support interrupts or exceptions. Stack machines generally have good interrupt performance since there are no registers to save. The cause of the interrupt is easy to determine since the simple pipelining means there is only a single instruction being executed at any time.

### 3.4.5 Implementing Forth primitives

All Forth primitives must be represented by the 17 instructions of MSL16 if the stack architecture defined by Forth are to be emulated. **AND**, **XOR**, **+**, **-**, **0=**, **DUP**, **DROP**, **SWAP**, **R>**, **>R**, **2/**, **@** and **!** are directly supported. The most frequently used Forth primitives which are not directly implemented are translated in Table 3.2. The first column lists the most frequently used but not included Forth primitives and the corresponding translation in MSL16's instructions are shown in the second column. The equivalent instruction sizes in bits are included for the estimation of code density of MSL16 to compare with a 16-bit Forth reference machine [19].

Forth Primitive	Translated Instructions	Instruction Size (bits)
0BRANCH	0=, LIT addr_msb, LIT addr_lsb XOR, AND, GOTO	$4 \times 4 + 16 \times 2 = 48$
2*	DUP, +	$4 + 4 = 8$
BRANCH	LIT addr_msb, LIT addr_lsb XOR, GOTO	$16 \times 2 + 4 \times 2 = 40$
DDROP	DROP, DROP	$4 \times 2 = 8$
OVER	>R, DUP, R>, SWAP	$4 \times 4 = 16$
EXIT	R>, GOTO	$4 \times 2 = 8$
LIT	LIT const_msb, LIT const_lsb, XOR	$16 \times 2 + 4 = 36$

**Table 3.2:** Translation of Forth primitives

### 3.4.6 Code Density Estimation

By translating the most frequently used Forth primitives, the code density of Forth programs implemented with MSL16A's own instruction set could be estimated. In [19], the static instruction frequencies of all Forth primitives for a 16-bit Forth system are listed. The list was compiled with four benchmark programs. The average static frequencies (of the four benchmark programs) are used to calculate the average instruction size for MSL16.

Forth Primitive	Static Frequency (X%)	Instruction size (Y bits)	X*Y	average instruction size
!	2.32	4	9.28	798.07 ÷ 76.17 =10.48 bits
+	2.90	4	11.60	
-	1.52	4	6.08	
0BRANCH	3.10	48	148.80	
2*	1.49	8	11.92	
>R	1.36	4	5.44	
@	5.59	4	22.36	
BRANCH	1.73	40	69.2	
CALL	25.87	1	25.87	
DDROP	1.42	8	11.36	
DROP	1.86	4	7.44	
DUP	3.28	4	13.12	
EXIT	7.47	8	59.76	
LIT	9.41	36	338.76	
OVER	2.49	16	39.84	
R>	1.50	4	6.00	
SWAP	2.81	4	11.24	
TOTAL	76.17	TOTAL	798.07	

**Table 3.3:** Code Density Calculation

The first column in Table 3.3 contains the most frequently used Forth primitives and the average static frequencies listed in [19] are shown in the second column. The third column lists the number of bits required to translate the primitive into MSL16's own instructions. The average instructions size of MSL16 was found to be about 10.48 bits long. This means that the same Forth program compiled for MSL16 can be 34.5% smaller than that for the 16-bit reference system. Because of its high code density, a system built with MSL16 can utilize a smaller external memory to store program codes. This also implies that MSL16 required smaller caches to improve system performance although no cache is incorporated in MSL16 currently.

As illustrated in the Berkeley InfoPad project [12], a processor does not dominate a system's total power dissipation, the memory itself can consume 50% of the total power. Similarly, the two large caches in the StrongARM 110 [13]



consume 43% of the total chip power and fill up 90% of the total chip area. As a result, for extremely low power systems, it is crucial to optimize the whole system instead of just making a low power processor. A 34.5% smaller main memory and a smaller cache can lower the total system power consumption and size remarkably. This will be particularly beneficial to battery operated embedded systems.

### **3.5 Summary**

A brief introduction to RISC machines, stack machines and the Forth language was presented in this chapter. Stack machines are simple, small and fast which are critical in real time systems and many stack machines were designed to run Forth. The Forth language is based on a set of primitives that execute on a stack machine architecture. It has been estimated that Forth machines have 1.5 to 2.5 times better code density than RISC designs [37]. Code density relates directly to power efficiency as it affects the memory and cache sizes, which can consume up to 50% of the total system power, in a low power system.

Also developed in this chapter was the architecture of MSL16. MSL16 is a dual stack machine with 16-bit data and memory buses designed to run Forth. A two stage pipeline is employed and the instruction set is minimal. MSL16 is fast and power efficient because of its simple architecture, doesn't require fast memory or caches and its high code density. The code density of MSL16 was estimated to be 34.5% higher than a 16-bit Forth reference system which makes it highly desirable for low power applications like battery operated portable systems.



# Chapter 4

## Design Methodology

MSL16A was implemented based on Martin's synthesis method [21]. An advantage of such compilation-based method over other methods is that complex concurrent systems can be described concisely in high level constructs without low-level timing concerns, which makes modification and verification of the system behavior easier. The asynchronous circuits used are called *quasi-delay-insensitive*(QDI) circuits which do not use any assumption on delays in operators and wires [21]. The asynchronous control logic was designed using the Caltech Asynchronous Synthesis Tools (CAST), which is discussed in Chapter 5, while the datapath components were realized by manual layout with the layout editor Magic [41].

This chapter details how MSL16A was implemented based on Martin's synthesis method. The first section gives the basic notations used in the higher level description of the processor while subsequent sections describe the compilation methods. For a more complete explanation of the algorithm, readers are suggested to read [21].

## 4.1 Basic Notation

The notation used in describing the processor at a high level, is based on C.A.R. Hoare's Communicating Sequential Processes (CSP) [32]. A full description of the notation used in this paper can be found in [21].

An assignment of an expression  $e$  to a variable  $b$  is " $b := e$ ". If  $x$  is a Boolean variable,  $b \uparrow$  and  $b \downarrow$  represents  $b := \mathbf{true}$  and  $b := \mathbf{false}$  respectively.

A selection statement is of the form  $[G_1 \rightarrow S_1 \square \dots \square G_n \rightarrow S_n]$ .  $G_i \rightarrow S_i$  a "guarded command" where  $G_1$  through  $G_n$  are Boolean expressions (the *guards* of the commands) and  $S_1$  through  $S_n$  are program parts. The operational semantics of the selection statement is: "Wait until one of the  $G_i$ 's is **true**, then choose a guarded command with a **true** guard non-deterministically and execute the corresponding program part."  $[G]$  is equivalent to  $[G \rightarrow \mathbf{skip}]$ , which means "wait until  $G$  is **true**".

A loop statement is of the form  $*[G_1 \rightarrow S_1 \square \dots \square G_n \rightarrow S_n]$ . The operational semantics is: "Choose a guarded command with a **true** guard, execute the corresponding program part and then repeat until all  $G_i$ 's are **false**".  $*[S]$  stands for  $*[\mathbf{true} \rightarrow S]$  and means "repeat  $S$  forever".

For a communication channel  $X$ ,  $X!u$  means the communication action of sending the value of expression  $u$  through channel  $X$ . Similarly,  $X?v$  denotes the communication action of receiving the value from channel  $X$  and storing it in variable  $v$ . The combined effect of the two statements is to assign the value  $u$  from a process to variable  $v$  of another process.  $X!u$  cannot complete and its native process suspends unless  $X?v$  is executed in the corresponding process. Thus, dataless channels can be used to enforce synchronization between processes. A communication action on such a channel is expressed by naming the action with the channel's name. The boolean expression  $\overline{X}$ , the *probe* of channel  $X$ , is **true**

if a communication action over channel  $X$  is pending, i.e. communication action  $X$  can be completed without suspension.

Moreover, sequential composition is represented by a “;” and parallel composition, which is weakly fair, is denoted by “||”. Finally, “ $X \bullet Y$ ”, *coincident execution*, means communication actions over channels  $X$  and  $Y$  are to complete at the same time.

## 4.2 Specification of MSL16A

At the top-most level, MSL16A was described by a high level sequential program which is a non-terminating loop containing the FETCH and EXECUTE stages of the pipeline. The complete sequential program for the microprocessor is shown below:

```
*[ FETCH :  $i, pc := mem[pc], pc + 1;$ 
  EXECUTE : [  $alu(i.op) \rightarrow tos := aluf(tos, ds[k], i.op)$ 
    [  $nop(i.op) \rightarrow skip$ 
      [  $lit(i.op) \rightarrow [ lpc = 0 \rightarrow tos := lsb(ir) \& "00000000"$ 
        [  $lpc = 1 \rightarrow tos := "00000000" \& lsb(ir)$ 
        [  $lpc = 3 \rightarrow tos := psw$ 
      ]
    ]
    [  $dup(i.op) \rightarrow ds[k] := tos$ 
    [  $drop(i.op) \rightarrow tos := ds[k]$ 
    [  $goto(i.op) \rightarrow [ tos! = 0 \rightarrow pc := tos$ 
      [  $tos = 0 \rightarrow skip$ 
    ]
    [  $rto(i.op) \rightarrow ds[k] := tos; tos := rs[k]$ 
    [  $tor(i.op) \rightarrow rs[k] := tos; tos := ds[k]$ 
    [  $at(i.op) \rightarrow tos := mem[tos]$ 
  ]
```



```

    [ store(i.op) → mem[ds[k]] := tos
    [ swap(i.op) → tos ⇔ ds[k]
    [ call(i.op) → rs[k] := pc; pc := ir
    ]
]

```

The sequential program is basically a loop statement which denotes the **FETCH** and **EXECUTE** stages of MSL16A are repeated forever. The *FETCH* and *EXECUTE* inside the program are labels inserted simply for illustration purpose. Variable *i* contains the instruction which is currently being executed. It is described in the **PASCAL** record notation as a structured variable consisting of several fields. All instructions contain an *op* field for *opcode* while the parameter fields depend on the types of the instructions, which were described in Chapter 3.

The array *mem* describes the external memory and the indexes *pc* to it describe the program counter. The data stack and return stack are described by the arrays *ds*[*k*] and *rs*[*k*] while variable *k* points to the currently active element in the stacks. The design and implementation of the stacks are discussed in Chapter 5.

Variable *tos* represents a separate top-of-stack register which allows concurrent access to *tos* and *ds*[*k*] (top element of data stack) for the ALU. The instruction register is described as *ir* and, similarly, the least-significant-byte of the instructions, which is used by **LIT**, is represented as *lsb*(*ir*). For example, *dup*(*i.op*) → *ds*[*k*] := *tos* means if the instruction currently being executed is **DUP**, the value of the top-of-stack register will be pushed into the data stack. In **SWAP**, *tos* ⇔ *ds*[*k*] indicates the value of the top-of-stack register and the value of the currently active data stack elements is swapped.



### 4.3 Decomposition into Concurrent Processes

The previous sequential program was then decomposed into a set of concurrent processes, which operate in parallel, communicate and synchronize with each other through *channels*, based on Hoare's CSP (Communicating Sequential Processes) model [32]. A channel connects two processes and the two ends of a channel are called *ports*. Attempts were made to convert each process into smaller sub-processes until it was simple enough such that no further decomposition is necessary. The final program of MSL16A is discussed below in two parts.

```
/* FETCH : continuously fetches instructions */
```

```
FETCH ≡ *[JR1; PCI1; JR2; PCI2]
```

```
/* JR : temporary instruction holder
```

```
will discard instructions when necessary
```

```
pre-decode for CALL, ISCALL stalls the pipeline */
```

```
JR ≡ *[[ $\overline{JR1}$  → JR1; DOUT?i; JR2; [ $\overline{INST1}$  & !call → INST1!i; INST2
```

```
    [[ $\overline{INST1}$  & call → INST1!i; INST2; ISCALL
```

```
    [[ $\overline{NEXTa}$  → NEXTa]
```

```
    ]
```

```
]]
```

```
/* IR : instruction register
```

```
fetch instruction from JR and pass to EX 4-bit at a time */
```

```
IR ≡ *[INST1?i; INST2;
```

```
    E1!i • LPI1; [ $\overline{E2}$  → LPI2; E2; E1!i • LPI1; [ $\overline{E2}$  → LPI2; E2; E1!i • LPI1;
```

```
        [ $\overline{E2}$  → LPI2; E2; E1!i • LPI1; [ $\overline{E2}$  → LPI2; E2;
```

```
            [[ $\overline{NEXTb}$  → E2; LPI2]
```

```

                                        ]]NEXTb → E2; LPI2]
                                ]]NEXTb → E2; LPI2]
                        ]]NEXTb → E2; LPI2]
]

/* LPC : nibble pc
   controls the mux to EX also */
LPC ≡(*[[LPI1 → LPI1; x := lpc + 1; [LPI2 → lpc := x • LPI2
                                        ]LZ → lpc := 0 • LZ; LPI2
                                        ]
    ]]
    ]]*[[LPI1 & lpc = 0 → LPI1 • Na1; LPI2 • Na2
        ]LPI1 & lpc = 1 → LPI1 • Nb1; LPI2 • Nb2
        ]LPI1 & lpc = 2 → LPI1 • Nc1; LPI2 • Nc2
        ]LPI1 & lpc = 3 → LPI1 • Nd1; LPI2 • Nd2
    ]]
)

```

Process *FETCH* fetches a 16-bit word, generally containing four 4-bit instructions, from the memory and stores it in *JR*. Process *JR* is a temporary instructions holder which also pre-decodes the first instruction to check if the 16-bit word is a *CALL* instruction. Moreover, if the instruction currently being executed in process *EXEC* is *goto*, *JR* will discard the pre-fetched instructions in it through the **NEXTa** communication with *EXEC*. Process *IR* receives the 16-bit instruction batch from *JR* and then sends the corresponding instruction to process *EXEC* according to *LPC*. The process *LPC* updates the nibble pc of the next instruction concurrently with the instruction execution. The tempo-

rary variable  $x$  is inserted to keep the value of  $lpc$  stable during the execution of instructions.

```
/* WPC : word pc
    sends instruction address to memory */
WPC ≡ *[[ $\overline{PCI1}$  →  $PCI1$ ;  $y := wpc + 1$ ;  $PCI2$ ;  $wpc := y$ 
    [ $\overline{PCW}$  →  $W?wpc \bullet PCW$ 
    [ $\overline{PCR}$  →  $WPC!wpc \bullet PCR$ 
    ]]
```

```
/* ALU : the ALU,  $i$  is the opcode */
ALU ≡ *[[ $\overline{AC}$  →  $AC?i$ ;  $z := aluf(a, b, i)$ ;  $ALU!z$ ]]
```

```
/* the stacks,  $k$  is the pointer */
DS[k] ≡ (*[[ $\overline{DPUSH}$  &  $k$  →  $DPUSH \bullet DSOUT!x$ ]]
    || * [[ $\overline{DPOP}$  &  $k$  →  $DPOP \bullet DSIN?x$ ]]
    )
```

```
RS[k] ≡ (*[[ $\overline{RPUSH}$  &  $k$  →  $RPUSH \bullet RSOUT!x$ ]]
    || * [[ $\overline{RPOP}$  &  $k$  →  $RPOP \bullet RSIN?x$ ]]
    )
```

```
TOS ≡ *[[ $T!x$ ;  $U?x$ ]]
```

```
/* EXEC : execution unit
    instruction ends with NEXTa before NEXTb
    GOTO: PCW inside NEXTa to make sure pre-fetch is done
    CALL: ISCALL will resume the pipeline */
```



```

alu0 { AND, XOR, +, - }
alu1 { 2/, 0= } */
EXEC ≡ *[ E1?i;
    [alu0(i) → AC!i • DPOP • T; E2
    [alu1(i) → AC!i • T; E2
    [nop → E2
    [lit&lpc = 0 → DPUSH • T; NEXTb1; E2; NEXTb2; LZ
    [lit&lpc = 1 → DPUSH • T; NEXTb1; E2; NEXTb2; LZ
    [lit&lpc = 3 → DPUSH • T; E2
    [dup → DPUSH • TR; E2
    [drop → TW • DPOP; E2
    [goto → [T! = 0 → NEXTa1; TR • PCW; NEXTa2 • NEXTb1; E2;
                NEXTb2; LZ
                [T = 0 → E2]
    [rto → RPOP1 • DPUSH1 • T1; DPUSH2 • T2 • RPOP2; E2
    [tor → RPUSH1 • T1 • DPOP1; RPUSH2 • T2 • DPOP2; E2
    [at → T1; T2; E2
    [store → T • DPOP; E2
    [swap → DPOP; T • DPUSH; E2
    [call → RPUSH • PCR; PCW; ISCALL; NEXTb1; E2; NEXTb2; LZ
    ]
]

```

After receiving the instruction from *IR*, process *EXEC* decodes and executes it. Process *WPC* implements the word *pc* of MSL16A which updates the address of the next instruction fetch. The execution of ALU instruction by process *ALU* can overlap with the fetch operation and the update of *wpc*. There are two different classes of ALU instructions and they differentiate themselves by the

need of a **DPOP** communication during execution. The array of processes  $DS[k]$  and  $RS[k]$  describe the data stack and return stack respectively. Variable  $k$  is the pointer in the stack which means only the stack element holding the pointer will involve in the *PUSH/POP* communication.

## 4.4 Separation of Control and Datapath

Each concurrent process must be decomposed into two parts, the control part and its associated datapath. As the datapath of a process can be implemented in a fairly standard way, the datapath of a process is separated from the control part. To obtain the control part of a process, each communication command involving data passing is replaced with a “bare” communication on the channel. e.g.  $E!i$  and  $E?i$  would be replaced with  $E$ . Then, all data assignments are delegated to subprocesses and replaced with a communication command on a new channel between the control part and the datapath. After these transformations, the control part of each concurrent process consists merely boolean expressions in conditionals and communication commands. For example, after the removal of the datapath, the control for the *TOS* becomes  $*[T;U]$ . Finally, each communication command is implemented with 4-phase handshaking protocol.

## 4.5 Handshaking Expansion

Even though process decomposition can simplify complex control structures, commands such as *probe* and *selection*, *send* and *receive* remain. The next step in the synthesis method represents each communication action with operation on Boolean variables. The two ends of a channel must obey the 4-phase (return-to-zero) handshaking protocol in order to maintain correctness.

### 4.5.1 4-Phase Handshaking Protocol

Consider the matching pair of actions  $E1!i$  and  $E1?i$  in processes  $IR$  and  $EXEC$  respectively. The bare communication on channel  $E$  can be implemented by two handshake wires as shown in Figure 4.1.

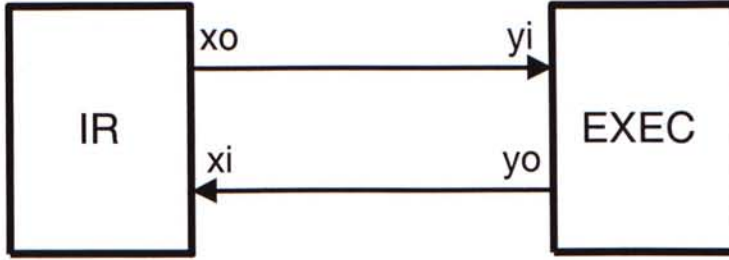


Figure 4.1: Bare Communication on Channel  $E$

To convert them into sets of transitions, they are expanded into 4-phase handshaking protocol. The protocol is not symmetrical: all communications in one end are implemented as *active* and all communications in the other end are implemented as *passive*. The standard active and passive implementations are<sup>1</sup>:

$$\begin{array}{cccc}
 [yi]; & yo \uparrow; & [\neg yi]; & yo \downarrow & (passive) \\
 \uparrow & \downarrow & \uparrow & \downarrow \\
 xo \uparrow; & [xi]; & xo \downarrow; & [\neg xi] & (active)
 \end{array}$$

It has been shown [42, 21] that it is easier to implement active input ports than a passive one. Therefore, all input actions in MSL16A are implemented as active and all output actions as passive. For example, the handshaking expansion for the control of the  $TOS$  is

$$TOS = *[[t_i]; t_o \uparrow; [\neg t_i]; t_o \downarrow; u_o \uparrow; [u_i]; u_o \downarrow; [\neg u_i]].$$

<sup>1</sup>By convention, input and output variables are subscripted with  $i$  and  $o$  respectively



Reshuffling of transitions and insertion of state variables are often performed to distinguish ambiguous states to maintain the correctness of the transition sequence.

## 4.6 Production-rule Expansion

Production-rule expansion is the transformation from a handshaking expansion to a set of production rules, which lead to a physical circuit realization. After the handshaking expansion step with all states distinguishable is done, the explicit sequential operators (the semi-colons, ;) are removed by transforming it into a set of *production rules*. Each of the concurrent processes is represented with a set of production rules for realization of actual circuit.

A production rule (PR) is of the form " $G \rightarrow S$ ".  $G$  is called the *guard* which is a Boolean expression and  $S$  is an assignment of **TRUE** or **FALSE** to a Boolean variable. For any variable  $z$ , a PR for  $z \uparrow$  and a PR for  $z \downarrow$  must be *complementary* and *non-interfering* (never both enabled). In addition, *self-invalidating* PR's are not allowed since they never occur in actual physical circuits. An example of a self-invalidating PR is  $\neg z \rightarrow z \uparrow$  which the assignment of  $z$  falsifies its own guard. *Reset* signals are added to the PR sets in order to put the microprocessor in the proper state upon power-up.

It is important to realize that the circuits which result from this synthesis process require complex and custom gates, which cannot be broken down into simpler components [23].

## 4.7 Summary

The design flow and circuit style used for MSL16A were an original application of Martin's synthesis method which relies on the time-honored "divide-and-conquer" principle. This methodology decomposes a high-level sequential program describing the microprocessor into a set of concurrent processes written in a language similar to Hoare's *Communicating Sequential Processes* and then translates them into asynchronous circuits. Communication commands are converted into sets of transitions and expanded based on the 4-phase handshaking protocols. To eliminate indistinguishable states, transitions are reshuffled and state variables are inserted when necessary. Finally, production rules are generated, which lead to a physical circuit realization.

In MSL16A, all datapath elements are accompanied by a small control circuit (*separation of control and datapath*, see Section 4.4), obeying the 4-phase handshaking protocol, for synchronization of *request* and *acknowledge* signals. Data are all dual-rail encoded within the processor core. Only one out of two rails is raised at each active phase of the four-phase protocol.

Control circuits were generated and verified with CAST, a set of tools developed at CALTECH for the synthesis of asynchronous circuits (described in Section 5.3), while all other elements were created by manual layout with Magic [41]. MSL16A was functionally verified with IRSIM [43] and the simulation results are presented in Chapter 6.

# Chapter 5

## Implementation

This chapter describes the design of MSL16A in detail in a bottom-up fashion. The first section discusses the implementation of the C-element, a widely used asynchronous circuit primitive logic element, in the processor. The asynchronous arbitration circuit used is presented in the second section. Three different stack designs were proposed in search for a low power stack implementation and these designs are described in the next section. A description of the ALU design follows and the chapter concludes with an overall description of the entire MSL16A processor.

### 5.1 C-element

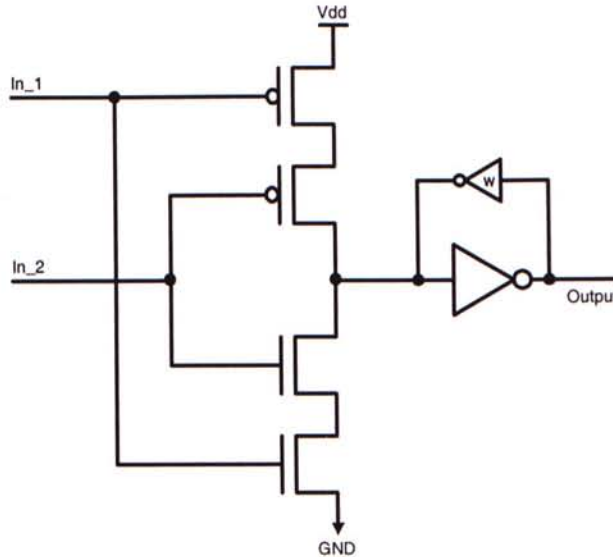
The C-element, introduced by D.E. Muller [44] in 1959, is also called the **Muller C-element**. A C-element has two inputs  $x$  and  $y$ , and one output  $z$ . Its logical behavior is described in Table 5.1. If both  $x$  and  $y$  are the same, the output will become the same as the inputs; otherwise the output remains the same. The behavior of the output can also be expressed in terms of  $x$  and  $y$ , and the



previous state of the output,  $z'$ , by the following Boolean expression

$$z = (x + y) \cdot z' + x \cdot y$$

C-elements are widely used as primitive logic elements in asynchronous VLSI circuits to sense the completion of communication actions. Figure 5.7 in page 60 shows how C-elements can be utilized to sense the completion of a data transfer operation in an asynchronous system.



**Figure 5.1:** The C-element

The implementation of the C-element in MSL16A, introduced by Alain Martin and used in the Caltech asynchronous microprocessor, is shown in Figure 5.1. This circuit utilizes an inverter latch to maintain the state of the output when the values of the input are different. The feed-back inverter is a weak one to allow the changes in the state of the inverter latch.

Previous work on C-element and various designs by other researchers can be found in [45, 46, 47].

$x$	$y$	$z$
0	0	0
0	1	unchanged
1	0	unchanged
1	1	1

Table 5.1: Truth table of a C-element

## 5.2 Mutual Exclusion Elements

MSL16A has a single memory port and a single program counter which is shared by the *FETCH* and *EXEC* processes. As both processes were concurrent, an arbitration circuit was required to resolve simultaneous requests of a shared resource.

An asynchronous arbiter is a circuit that dynamically allocates a single shared resource to the concurrent processes in an asynchronous system. Each process issues a *request* when it requires the shared resource and waits until the arbiter produce a *grant*. The process then uses the resource and releases the *request* when finished. After servicing a *request*, the arbiter releases the *grant*. If the arbiter receives several requests from different processes, a *grant* will be generated to exactly one of them and leaves other requests pending until that process, the one who receives the *grant*, has released the *request*. The arbiter then services another request and all *requests* are serviced on a mutually exclusive basis.

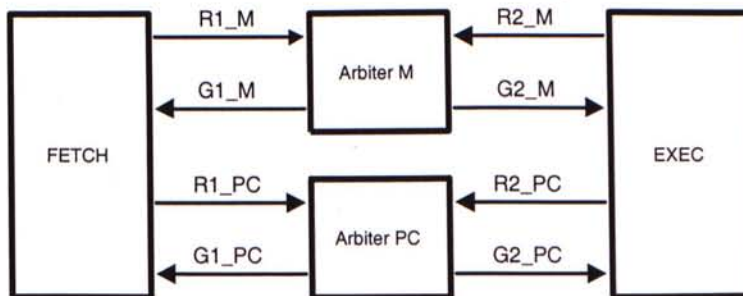
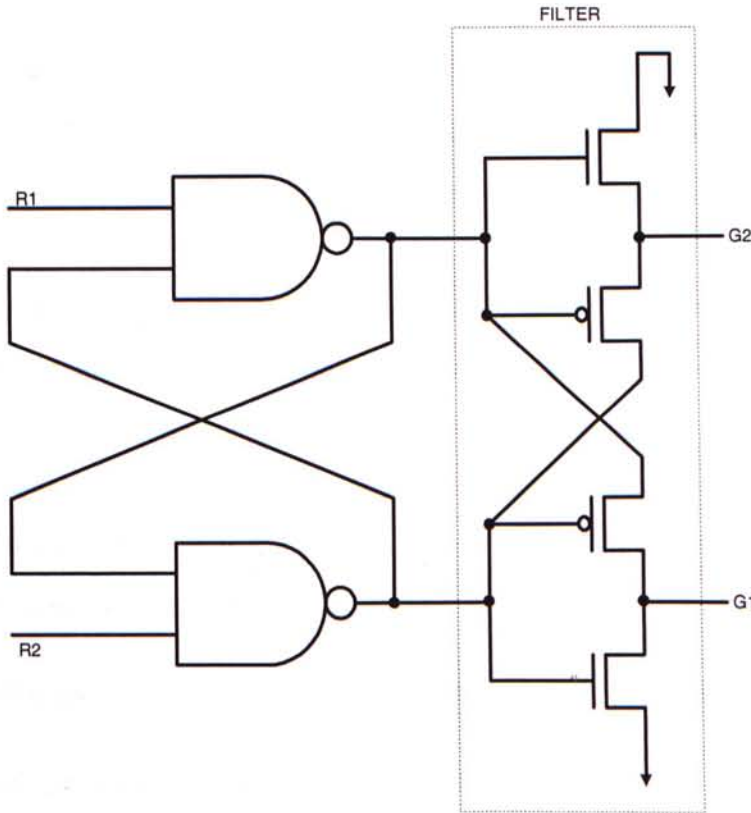


Figure 5.2: A Simple Arbitration System

In MSL16A, fairness are ensured not by the design of the arbitration circuit but the design of the concurrent processes at the CSP (Communicating Sequential Process) level. An arbiter is *strongly fair* when a request is granted after a bounded number of other requests are granted. Arbiters in MSL16A do not have to address this issue as the execution of *EXEC* depends on *FETCH*. The request from *EXEC* will always be granted as *FETCH* can only pre-fetch a 16-bit instruction once before *EXEC* has finished the execution of previous instructions. The arbiter is thus *strongly fair*. The arbiter system of MSL16A is shown in Figure 5.2.



**Figure 5.3:** A basic arbitration circuit

A number of complex asynchronous arbitration circuit designs have been reported in the literature [48, 49, 50, 51, 52]. However, in quasi-delay-insensitive designs, the correct functioning of a circuit containing an arbiter does not depend



on the duration of the metastable state and relatively simple implementations of arbiters can be used. Figure 5.3 shows the implementation of the arbiter in MSL16A. There are totally two arbiters in the microprocessor, one for the access of the memory port and the other for the program counter. Deadlock situations will never occur as none of the execution of any instructions in process *EXEC* requires the *grants* from both arbiters. They will never have to wait for a *grant* signal while holding the other one.

### 5.3 Caltech Asynchronous Synthesis Tools

Part of the processor's layout was generated automatically with the Caltech Asynchronous Synthesis Tools (CAST). A brief description of the CAST tools used is given below. For a full description, please refer to [53].

**prs2prs** The main purpose of prs2prs is to help organize a production rule set in a hierarchical manner. Prs2prs can read a hierarchical production rule set, apply some operations to it, and print the result. By default, no operations are applied, but flags can be used to force, e.g., flattening of the input file (all PRs will be in Disjunctive Normal Form). The input and output languages are identical.

**bubble** Performs *bubble-shuffling* on a production rule set (PRS).

**prs2tau** prs2tau inputs a production rule set and, using one of four possible timing models, converts it into a set of timed production rules described as a .tau file. This output file can then be used by prsim for timing simulations.

**prsim** prsim is an interactive event-driven simulator for a set of timed production rules. A timed production rule has the same format and logical interpretation as a standard production rule except that the former has an

associated timing value. This value specifies the delay between the guard becoming true and the subsequent firing of the corresponding transition.

**cellgen** cellgen takes a PRS as input and produces a set of CMOS magic cells, each one implements an element in the PRS. In all cases, it generates a definition file, which relates the variables in the PRS to nodes in the cells, and an output file, which specifies how the cells should be interconnected. The two files are used by Vgladys to generate the final layout of the PRS.

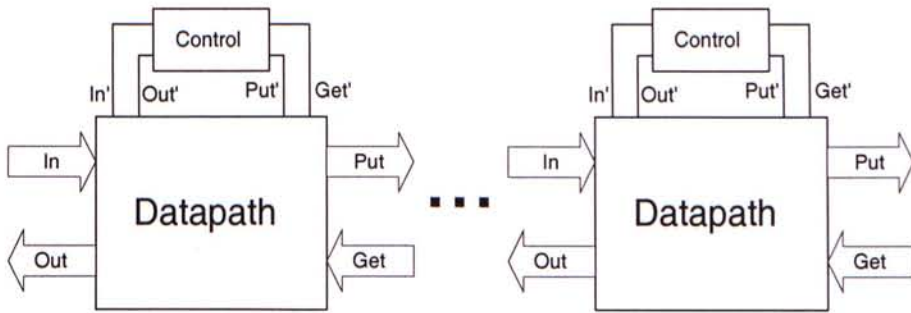
**Vgladys** By using magic cells generated by cellgen , Vgladys generates the final layout of the PRS.

As described in Section 4.6, each of the concurrent processes describing MSL16A is represented by a set of production rules. CAST simulates asynchronous circuits at the production rule level. By using this set of tools, different circuit elements of MSL16A were allowed to be first verified at a higher level, before creating circuit layouts, and thus reduced the development effort.

## 5.4 Stack Design

Stacks are a fundamental building block in microprocessors, microcontroller and DSPs. Stack design is crucial to the performance of MSL16A as the data and return stacks consume almost half of the total chip area. As almost all instructions involve in one or more stack push or pop operations, a poorly designed delay-insensitive stack will greatly reduce the overall power efficiency of the microprocessor. However, to our knowledge, no one has made any comparison on different delay-insensitive stack implementations.

A stack element composes of a control process and 16 datapath processes. A stack can be implemented as an array of these stack elements by connecting the



**Figure 5.4:** The Stack

correct channels together (see Figure 5.4). For push operations, the *In* channel allows the stack element to receive a new data word while the *Put* channel is used to send out the stored data word. The *Get* and *Out* channels are used similarly when popped. The design of the Pointer Stack is a bit different and it will be explained in subsequent sections.

Three different stacks (the Eager Stack, the Lazy Stack and the Pointer Stack) were implemented and performance analysis was made to find out the which asynchronous stack implementation was most suitable for MSL16A.

### 5.4.1 Eager Stack Control

The Eager Stack is a simple design. The control processes of all stack elements within an Eager Stack are the same. Each stack element will reshuffle after a push or a pop operation. Regardless of whether the value stored in a single stack element is meaningful or not, it will send out its data word stored to the next element when pushed or get new data word from the next element when popped. i.e. all stack elements will do the same job no matter the stack is full, half-full or even empty. The control process does not have to maintain the current state of that particular stack element. This leads to a much simpler control process.



It will always behave like program P.

$$P \equiv *[[ \overline{in} \rightarrow put!x; in?x \\ \quad \square \overline{out} \rightarrow out!x; get?x \\ \quad ]]$$

Program P means that when there is data to be received through the channel *in*, the data word stored in register *x* will be sent to the next element through the channel *put*. Similarly, for pop operations, the stack will pass out the data word stored in *x* through the *out* channel and then pop the data from the next element through the *get* channel.

The stack is initially empty and state variable *z* is introduced to ensure mutual exclusion between the push and pop communication sequences. After handshaking expansion, we get

$$P \equiv *[[ z \wedge ini \rightarrow puto\uparrow; [puti]; t\uparrow; [t]; puto\downarrow; [\neg puti]; ino\uparrow; [\neg ini]; t\downarrow; [\neg t]; ino\downarrow \\ \quad \square z \wedge outi \rightarrow outo\uparrow; s\uparrow; [s]; z\downarrow; [\neg z \wedge outi]; outo\downarrow; get\uparrow; [geti]; s\downarrow; [\neg s]; geto\downarrow; \\ \quad \quad [\neg geti]; z\uparrow \\ \quad ]]$$

The compilation of program P gives the resulting circuit for the Eager Stack's control process. The circuit is shown in Figure 5.5.

### 5.4.2 Lazy Stack Control

The epithet “lazy” means that after a pop operation, all stack elements do not reshuffle unlike the Eager Stack. A popped stack element will change its states from *full* to *empty* and the control processes have to maintain the state information for each stack element. Each stack element is either empty or full. When it



element and pass the data out. If it is full, it will acknowledge the request with the stored data word and go into the *empty* state without asking for a new data word from the next element. This design will perform best if a pop operation is followed by a push operation or similarly a push operation followed by a pop operation.

The stack is initially empty. A channel  $(t, t')$  is introduced so that F can be called within E and  $z$  is introduced to ensure mutual exclusion between the two guarded commands of E. After handshaking expansion [21], we get

$$\begin{aligned}
 E &\equiv *[[ z \wedge ini \rightarrow ino \uparrow; s \uparrow; [s]; z \downarrow; [\neg z]; [\neg ini]; ino \downarrow; to \uparrow; [ti]; s \downarrow; [\neg s]; to \downarrow; [\neg ti]; z \uparrow \\
 &\quad \square z \wedge outi \rightarrow geto \uparrow; [\neg outi]; geto \downarrow \\
 &\quad ] ] \\
 F &\equiv *[[ ti' \wedge ini \rightarrow puto \uparrow; [puti]; u \uparrow; [u]; puto \downarrow; [\neg puti]; ino \uparrow; [\neg ini]; u \downarrow; [\neg u]; ino \downarrow \\
 &\quad \square ti' \wedge outi \rightarrow outo \uparrow; ti' \uparrow; [\neg ti' \wedge \neg outi]; outo \downarrow; to' \downarrow \\
 &\quad ] ]
 \end{aligned}$$

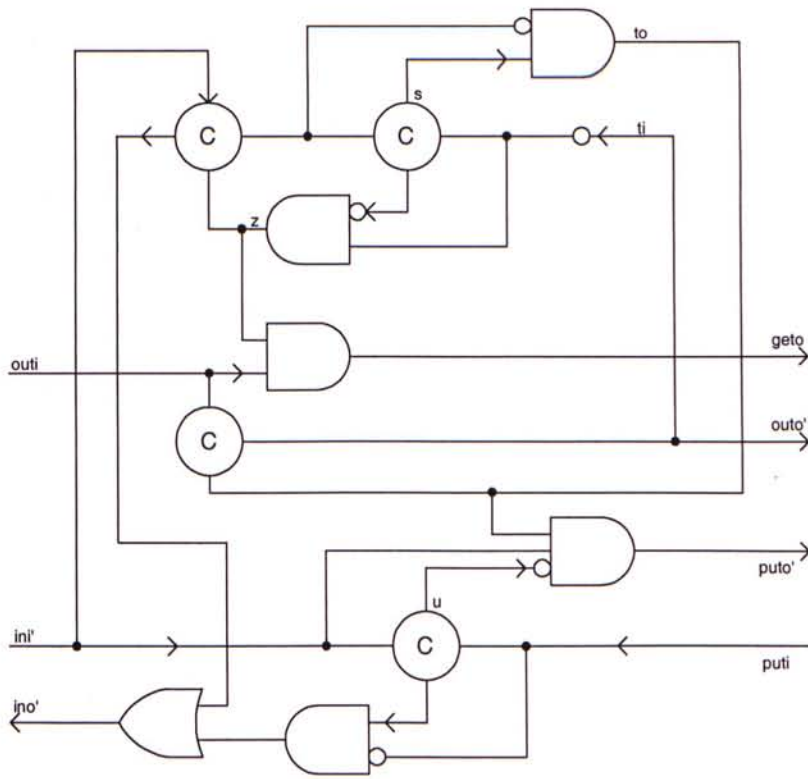
The compilation [21] of processes E and F gives the resulting circuit for the control part of the Lazy stack as show in Figure 5.6.

### 5.4.3 Eager/Lazy Stack Datapath

After the control processes had been created, the corresponding datapath processes must also be designed to construct the stack. The channels *in*, *out*, *put*, *get* of the control process were renamed *in'*, *out'*, *put'*, *get'* for the datapath process to communicate with it. The datapath communicates with the environment via the four channels *in*, *out*, *put*, *get* (see Figure 5.4). All data-passing actions are dual rail encoded [21] which requires two wires to transfer one bit, one for the binary value '0' and the other for binary value '1'.

When a register cell has been set to the correct value of the data wire, it will generate a completion signal which is gathered together by a C-element. Since





**Figure 5.6:** Implementation of the Lazy Stack's control process

a 16-input C-element is too slow to implement [2], a tree of smaller 4-input C-elements, which is called a *completion tree*, are used to generate a completion signal from a 16-bit datapath to the control process.

The datapath process has to implement the following channel interfaces:

- \*[in' • in?x]
- \*[out' • out!x]
- \*[put' • put!x]
- \*[get' • get?x]

After handshaking expansion, the four channel interfaces for the Eager Stack are expanded as

\* $[[ini1 \vee ini2]; ini' \uparrow; [ino' \wedge ini1 \rightarrow x \uparrow; [x] \square ino' \wedge ini2 \rightarrow x \downarrow; [\neg x]]; ino \uparrow;$   
 $[\neg ini1 \wedge \neg ini2]; ini' \downarrow; [\neg ino']; ino \downarrow]$

\* $[[x \wedge outh' \rightarrow outh2 \uparrow; [\neg outh']; outh2 \downarrow$   
 $\square \neg x \wedge outh' \rightarrow outh1 \uparrow; [\neg outh']; outh1 \downarrow]]]$

\* $[[x \wedge puth' \rightarrow puth1 \uparrow; [\neg puth']; puth1 \downarrow$   
 $\square \neg x \wedge puth' \rightarrow puth2 \uparrow; [\neg puth']; puth2 \downarrow]]]$

\* $[[geti1 \rightarrow x; [x]; geti' \uparrow \square geti2 \rightarrow \neg x; [\neg x]; geti' \uparrow]; [\neg geti1 \wedge geti2]; geti' \downarrow]$

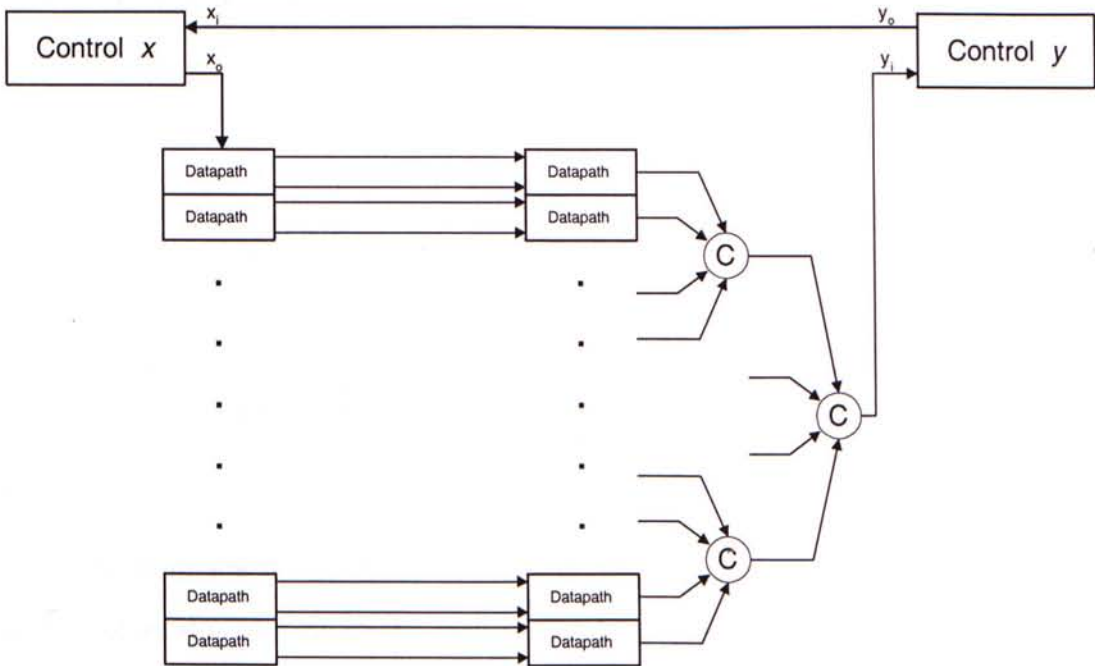


Figure 5.7: Implementation of communication

Figure 5.7 shows how a datapath is generally combined with a control to implement a communication command. The datapath processes for the Lazy stack are similar except that an optimization [21] has been made to improve its

performance. In program E, the values of  $x$  involved in both *get* and *out* actions are the same. Therefore, the received value does not have to be written into the register( $x$ ) before sending it out and can be sent directly on the *Out* channel instead.

#### 5.4.4 Pointer Stack Control

The Pointer Stack is the simplest design and delivers the best performance among the three stacks implemented. The key idea was that only one element will be active for each operation while others just idly waiting for their neighbors to pass the pointer to them. The pointer stack control was completely different from the two previous designs. The pointer was implemented as an internal bit variable within the control circuit. When it is active, it behaves like program A. Otherwise, it behaves like program I(Idle).

$$\begin{aligned}
 I &\equiv *[[ \overline{in} \rightarrow A \\
 &\quad \boxed{\overline{get} \rightarrow A} \\
 &\quad ]] \\
 A &\equiv *[[ \overline{push} \rightarrow push?x; put; I \\
 &\quad \boxed{\overline{pop} \rightarrow pop!x; out; I} \\
 &\quad ]]
 \end{aligned}$$

Stack elements do not have to reshuffle after any operation. The active element will send a request to the next element, asking it to become active, after a stack operation. As usual, the pointer will be passed with the four-phase handshaking protocol. All other stack elements will not involve in any communication and stay completely idle. The stack will perform this way no matter it is full, half-full or empty.



### 5.4.5 Pointer Stack Datapath

The datapath process is different from the one designed for the Lazy/Eager Stack as no channel interfaces have to be implemented. The *in*, *out*, *put*, and *get* channels are now reserved for stack pointer passing within the stack and two new channels, *push* and *pop*, are introduced for the stack to communicate with the environment. The registers within the datapath are implemented the same way as before. Comparisons with tables describing the performance of the three stacks, including speed, area and power consumption, are presented in Section 6.1.

## 5.5 ALU Design

A delay-insensitive ALU was developed as a part of an asynchronous implementation of the MSL16 microprocessor. The result was a small, simple ALU which delivers comparable performance with more sophisticated synchronous counterparts. The asynchronous nature of the unit takes advantage of best- and average-case performance while allowing rare worst-case operations to take longer to complete, thus giving a high average throughput.

The MSL16 architecture defines a stack-based processor in which arithmetic and logic operations normally require one or two operands to be read from the data stack and the result is returned to the stack as well. The set of functions provided by the ALU is standard which consists of basic logic operations, arithmetic shift, addition and subtraction. Time consumed in processing each operation is not constrained to a fixed cycle time but depend on both the operation and the data itself. Addition, or subtraction, is the most time consuming function as all logical operations are performed in a bitwise fashion while worst case addition operation may require communications across the entire 16-bit word length.

### 5.5.1 The Addition Operation

A single bit full adder requires 3 inputs, the 2 operand bits and a carry in. The speed of the addition is limited by the carry signal propagation speed across the whole word. The carry output from a single bit addition does not always depend on the carry input of the previous stage, e.g. both operand bits are '1's or '0's (see Table 5.2). It is highly unlikely that a carry signal will have to propagate across all bits before giving a correct result. However, in synchronous ALUs, this rare case must be handled and a considerable effort has been made in schemes like carry-look-ahead and carry-select so as to speed up the addition process. These approaches require more circuitry to accommodate the rare worst-case condition.

A	B	Cout
0	0	0
0	1	Cin
1	0	Cin
1	1	1

**Table 5.2:** Carry Output of a Full Adder

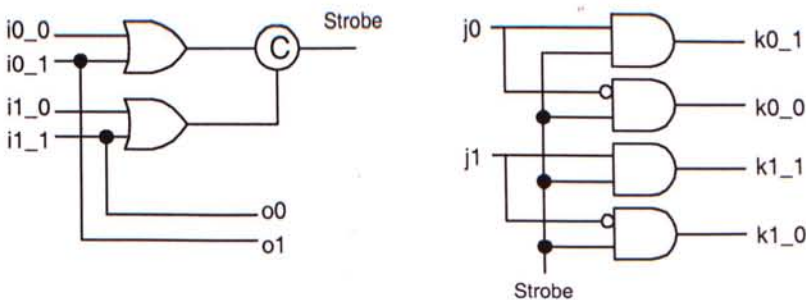
In an asynchronous ALU, addition may take different times to complete depending on the input data as operation completion can be sensed. As long carry chains are relatively rare, a simple ripple adder may be used to deliver "typical" performance at a smaller size. As the carry signals will propagate along different carry chains in parallel within the same operation, only the longest chain is significant in addition cycle time calculation. A study at Manchester University [54] suggests that the mean carry propagation length is about 4.4 bits for 32-bit operands, assuming operands to be random. Thus, the average longest carry chain can be safely assumed to be over four times less than the 16-bit worst-case of MSL16. Our adder has no special fast logic and performs addition with a chain of 16 full adders. More in-depth discussions on asynchronous adders can be found in [55, 56, 57, 58, 59].

### 5.5.2 Zero-Checker

Logic operations are not significantly data dependent and logic units are implemented in a standard fashion. One interesting unit is the Quick-Decision Zero-Checker based on [60]. An ordinary delay-insensitive zero-checker will wait for all inputs to be valid before issuing any output. However, a Quick-Decision Zero-Checker will raise the output signal whenever one of the input bits is non-zero without waiting for all other input bits to be valid. The validity of all inputs still have to be checked to satisfy the delay-insensitive protocol but this can be done concurrently with other operations that would otherwise be postponed. This is particularly useful for the condition test of a conditional branch.

## 5.6 Memory Interface and Tri-state Buffers

As data were dual-rail encoded within the processor core, NAND gates (as shown in Figure 3.2) could be used, instead of using tri-state buffers, to save area along all 16-bit data buses. Moreover, Dual-rail to single-rail and single-rail to dual-rail converters [5] were used to interface with the outside world with bundled data encoding to allocate extra pins for testing. Figure 5.8 shows the gate-level descriptions of the converters.



**Figure 5.8:** Dual-rail to single-rail and single-rail to dual-rail converters



## 5.7 MSL16A

The C-elements, memory interface, ALU and stacks were combined with multiplexors, incrementers and adders to form the complete datapath of MSL16A (shown in Figure 5.9). The control logic was derived using the Matin's synthesis method described in Chapter 4. These units together form the MSL16A design.

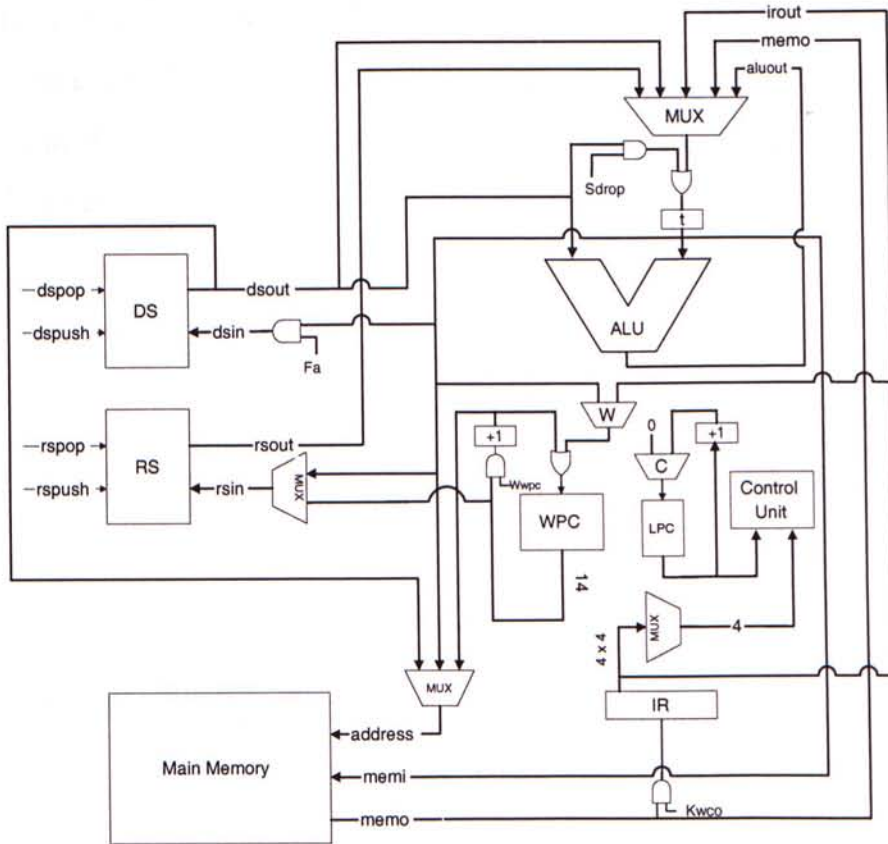


Figure 5.9: The datapath of MSL16A

---

## 5.8 Summary

Implementations details of important subcircuits of MSL16A were presented in this chapter. C-element is one of the most important circuit elements in asynchronous design and its implementation was presented. A simple arbitration circuit was employed in MSL16A to resolve the simultaneous request of shared resources. A description of three different asynchronous stack designs, the Eager Stack, Lazy Stack and Pointer Stack, was presented. Of the three designs the Pointer Stack was found to be the most energy efficient and the results will be presented in the next chapter. Also described in this chapter was the ALU and MSL16A processor. An evaluation of this design will be presented in the next chapter.

# Chapter 6

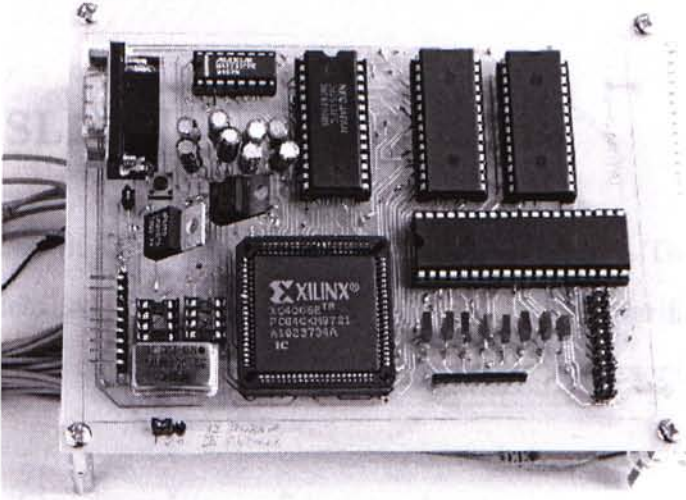
## Results

In this chapter, the performance of MSL16A is evaluated. The first section presents the results obtained by implementing MSL16 on FPGA (Field Programmable Gate Arrays). The performance of the synchronous version was encouraging which motivated the creation of an asynchronous re-implementation of MSL16. The second section evaluates MSL16A in which three stack designs are analyzed in terms of performance, power consumption and silicon area. This analysis helped in choosing the lowest power stack design for MSL16A. An evaluation of the ALU based on the same three criteria is presented in the same section. A description of the test program used and how the processor was functionally verified is also included. Finally, the performance of MSL16A is compared with two other asynchronous microprocessors, the ASPRO-216 and TITAC-2, and a low power synchronous microprocessor, StrongARM-110.

### 6.1 FPGA based implementation of MSL16

MSL16 was first coded in VHDL (VHSIC Hardware Description Language) and functional correctness tested by making VHDL memory images of hand coded assembly language programs and then testing them on the Synopsys VHDL





**Figure 6.1:** MSL16 prototype board

simulator *vhdlbx*. The small size of the instruction set and the simplicity of the design made design and testing of the machine very simple. The test program uses all of the instructions of the MSL16 instruction set and also tests worst case carry condition of the ALU ( $1-1=0$ ).

The design was then targeted for Xilinx Inc, 4000 series FPGAs and synthesized using the Synopsys *FPGA compiler*. The resulting design occupied 175 configurable logic blocks (CLBs). The VHDL description is generic except that the Xilinx RAM feature was used to implement the two stacks and the program memory (a  $16 \times 16$  RAM which is initialized with our startup program). The Xilinx RAM feature achieves a  $16\times$  improvement in density over a normal CLB.

Finally, a prototype printed circuit board (shown in Figure 6.1) containing the MSL16 processor on a Xilinx 4006E-1 FPGA (which includes ROM), an RS232 port,  $32K \times 16$  bit static RAM, an Intel 8255 programmable peripheral interface chip and display LEDs was developed. Using this board, the design

was found to be operational up to 33 MHz, therefore the processor has a peak execution rate of 33 MIPS.

## 6.2 MSL16A

The evaluation of MSL16A is divided into three parts. The stack and ALU designs are detailed separately, followed by the functional verification of MSL16A and performance comparison with other low power processors.

### 6.2.1 A Comparison of 3 Stack Designs

To evaluate the three stack designs described in Section 5.4, three different 16-bit stack elements were designed. A 16-bit stack element was created by connecting one control process and sixteen 1-bit datapath processes together. The same input to the stack was used in all three cases for a fair comparison. In all cases, these results came from stack elements running at 5V power supplies. All PFETs are  $8\lambda \times 2\lambda$  and NFETs are  $4\lambda \times 2\lambda$  except some of them were sized for better performance. The stacks were implemented with CAST and simulated with HSPICE. All measurements were based on HSPICE(98.2) on a AMI  $1.2\mu$  CMOS double layer metal process, using MOSIS parametric test results of run N81Y.

#### 6.2.1.1 Performance

The required cycle times for push or pop operations are listed in Table 6.1. The best-case push cycle (first element is *empty*) of the Lazy Stack was found to be 13.3% shorter than that of the Eager Stack. Moreover, the best-case pop cycle (first element is *full*) of the Lazy Stack is 50.9% shorter. In fact, for the Eager Stack, the worst-case is the same as the best/average-case as each stack element will do the same communication actions when pushed/popped even if it is empty.

Stack	Operation	Cycle Time(ns)
Eager	push	17.280
	pop	15.936
Lazy	push ( <i>best case</i> )	14.980
	pop ( <i>best case</i> )	7.824
	push ( <i>general case</i> )	19.510
	pop ( <i>general case</i> )	$2.956 \times N + 7.824$
Pointer	push	11.838
	pop	6.205

**Table 6.1:** Simulated cycle time for different Stack Design

However, if the first element is *full*, the Lazy Stack will be 12.9% slower than the Eager Stack when performing a push operation. The Lazy Stack performs best if and only if the first stack element is *empty*. For pop operations, the Lazy Stack will only lose if  $N \geq 3$ , where  $N$  equals to the number of *empty* elements before the first *full* element. It can be easily seen that the Lazy Stack will perform best when a push operation is followed by a pop operation ( $N = 0$ ) but this will not always be the case. As delay-insensitive circuits take advantage of best and average cases, the Lazy Stack should perform better than the Eager Stack.

In terms of performance, the Pointer Stack is a clear winner. It is about 20% faster than the best-case Lazy Stack with a constant cycle time. i.e. its performance will not degrade no matter how much data is pushed into the stack. This performance gain is achieved by eliminating the data reshuffle operation required in the other two designs. Only one pointer passing step instead of a series of data reshuffling.

#### 6.2.1.2 Power Consumption

In Table 6.2, the cycle time, power dissipation and power-delay product of a single Lazy/Eager stack element were compared. Power-delay product figures are included because they represent the energy consumed per operation which is a better metric for energy efficiency than power as power could be reduced



by simply lowering the clock speed of a processor. The much higher energy efficiency of the Lazy Stack can be easily overlooked. For an Eager Stack with 32 elements deep, the final power-delay product is actually 32 times higher as each stack element consumes power for each push/pop operation.

	Eager Stack Push	Lazy Stack Push(best)	Pointer Stack Push
Delay(ns)	17.280	14.980	11.838
Power(mW)	18.013	17.456	19.240
Power-Delay Product(pJ)	311.265	261.491	227.763
	Eager Stack Pop	Lazy Stack Pop(best)	Pointer Stack Pop
Delay(ns)	15.936	7.824	6.205
Power(mW)	20.978	11.451	12.380
Power-Delay Product(pJ)	334.305	89.593	76.818

**Table 6.2:** Power-Delay Product Comparison

The Lazy Stack takes advantage of the fact that idle components in an asynchronous circuit waste negligible power. As a result, only the first element and the *full* elements in front of it consume power when pushed and only the first *full* element and elements before it consume power. The Lazy Stack is thus a more energy efficient asynchronous stack implementation than the Eager Stack in the best case.

However, the Pointer Stack is the clear winner in terms of energy efficiency. Although the power consumption of the Pointer Stack is higher than the other two designs, it is more energy efficient as it actually runs faster and only the active element consume power, resulting in a lower power-delay product. i.e. less energy wasted for each push/pop operation for all situations. It consumes 12.9%/14.3% less energy even compared to Lazy Stack's best case push/pop situations.

### 6.2.1.3 Silicon Area

The circuit layouts were generated with automatic layout tools which produced stacks that are not practical (too large in area) to use in reality but the comparison is still meaningful. The sizes of a single 16-bit stack element of the three different stack implementations are shown in Table 6.3. The stack actually implemented in MSL16A is much smaller as its layout was created by hand. The final stacks can be composed by linking up these 16-bit stack elements, depending on how deep the stacks are. Thus, the size of the final stacks are directly proportional to the size of its 16-bit stack elements.

Although the control process of the Eager Stack is much simpler, the size of a Lazy Stack element is 5% smaller than that of an Eager Stack element. As mentioned earlier, each 1-bit datapath process of the Lazy Stack is optimized as it does not have to store the value of the next element when popped. This saves a considerable amount of space as a single stack element contains 16 1-bit datapath processes. A larger control process is thus insignificant. That is the reason why the three stacks' sizes are quite close to each other as the datapath processes are similar for the three designs.

The size of the pointer stack is slightly bigger, 6% larger than the Eager Stack, but still it is chosen for its energy efficiency at the cost of a slight decrease of silicon area efficiency. However, its control process is more complicated as it involves more communication actions which required more time to develop and verify.

Stack	Area
Lazy	$2364 \times 350 = 827,400\lambda^2$
Eager	$1980 \times 441 = 873,180\lambda^2$
Pointer	$1372 \times 675 = 926,100\lambda^2$

**Table 6.3:** Size of a single 16-bit stack element(automatically generated)

### 6.2.1.4 Hand Layout vs. Auto Layout

The layout generated by the automatic layout tools (CAST) is not area optimized which is only suitable for control circuit generation. Work done by other colleagues suggests that the same design layout by hand can be seven times smaller, with comparable performance and power consumption, but requires a much longer development cycle. With CAST, the stack layout can be generated within several days while hand layout may take more than 2 months to complete. The final decision was that all datapath components layouts were developed by hand while the layout of the control circuit of MSL16A was generated by CAST automatically.

## 6.2.2 Evaluation of the ALU

Performance estimates for the asynchronous ALU were made using HSPICE. In all cases, these results came from an ALU running at 5V power supply. All PFETs are  $6\lambda \times 2\lambda$  and NFETs were  $3\lambda \times 2\lambda$  except some of them were sized for better performance. The layout of the ALU was created by hand and simulated with HSPICE. Similarly, all measurements are based on HSPICE(98.2) on a AMI  $1.2\mu$  CMOS double layer metal process, using MOSIS parametric test results of run N81Y.

operation	Delay(ns)	Power(mW)	Power-Delay-Product(pJ)
XOR	2.596	45.741	118.741
AND	2.605	43.341	112.903
2/	3.130	50.358	157.621
0=	7.019	48.078	337.459
addition(typ.)	17.746	24.543	435.540
addition(worst)	42.343	14.995	634.888

**Table 6.4:** Performance of the ALU

Simulation results are shown in Table 6.4. The timings for addition closely



resemble the findings in [54], where a 32-bit asynchronous ARM<sup>1</sup> ALU targeted at 1.2 $\mu$  CMOS process was implemented. The longest carry chain in a "typical" addition is assumed to be 4 bits and the worst case is a 16-bit carry propagation. The size of the ALU is  $4362 \times 428 \lambda^2$ .

### 6.2.3 Evaluation of MSL16A

The final layout of MSL16A was functionally simulated using IRSIM(9.4.1) [43] with a custom designed test program and it was found to be functionally correct. The value stored in the T register, the output of the data and return stacks, the value output at the memory port and the decoded instructions were monitored in the simulation process to make sure the processor ran correctly.

The following is the program which was used to test the MSL16A micro-processor. It uses all of the instructions of MSL16A, and the listing below is organized so that each line corresponds to a 16-bit word containing 4 $\times$ 4-bit instructions. The first sequence computes 1-1 which is implemented as 1+(-1) in the ALU. This tests the worst case carry path of the processor. The program runs forever as it loops back to the start with a clean stack.

```
a:  NOP LIT 1          (test +, -,SWAP,LIT and NOP)
    NOP LIT 1
    - LIT 1
    NOP SWAP + LIT

    LIT FF00
    NOP LIT FF        (use FFFF to complement T)
    XOR XOR DUP DUP  (test XOR and DUP)
```

---

<sup>1</sup>a 32-bit RISC architecture developed by Advanced RISC Machines Limited

```

XOR >R R> NOP      (set T to 0 to test >R R>)
LIT 0
0= LIT b            (test 0=)
AND GOTO NOP NOP   (jump to b)

b: LIT FF00         (store 10 to mem[FF00])
NOP LIT 10          (and then read from mem[FF00])
NOP ! @ DROP       (DROP 10)
NOP DROP NOP NOP   (make the stack empty)
CALL c              (subroutine call)
NOP LIT a           (loop back to a)
NOP GOTO NOP NOP

c: NOP R> GOTO NOP  (subroutine return)

```

Figure 6.2 and 6.3 show how MSL16A was functionally verified with IRSIM. The number of monitored signals was reduced for demonstration purpose. Bus *addr* was the address bus, *mem* was the emulated external memory, **T** monitored the separate top-of-stack register and the rest monitored the correctness of the decoded instruction. As shown in the figures, MSL16A was executing the 8th line of code in *a* (XOR >R R> NOP) and fetching the instruction stored in memory location 9. The memory returned 5000h which represented **LIT 0**. The first **XOR** set **T** to 0 as the value stored on the top of the data stack is the same as **T** (FFFEh). The value stored in **T** was correctly restored to 0 after the execution of >**R** and **R**>.

Line 9–11 implemented an unconditional jump. **T** became FFFFh(-1) after **0=** as **T** was 0. The address of *b* - 1(0012h) was then stored in **T** as the branch target address. **AND** was used to test whether the branch condition was met

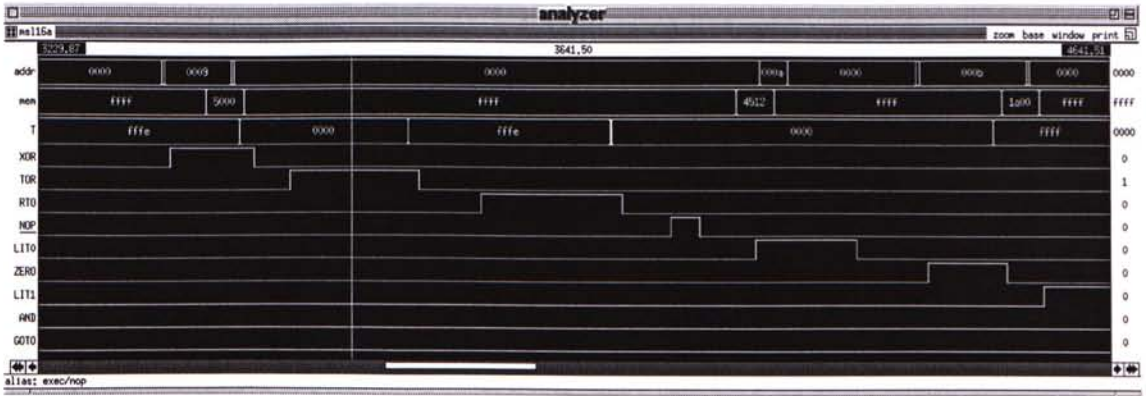


Figure 6.2: IRSIM simulation of test program (Part 1)

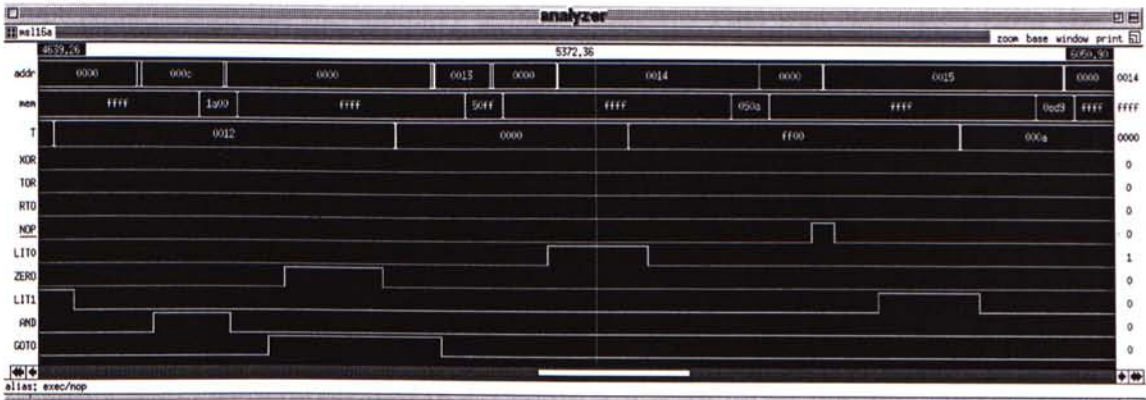


Figure 6.3: IRSIM simulation of test program (Part 2)

(set **T** to the target address if the condition was met, otherwise set **T** to 0) as **GOTO** would not alter the PC if **T** was 0 during execution. This could be seen in the execution of **GOTO** as **ZERO** was also raised to see if **T** is 0. After the execution of **GOTO**, MSL16A dropped the pre-fetched instruction at 000Ch and tried to fetch the instruction at 0013h(*b*) instead. The pipeline was stalled and waited for the memory to respond. After the reception of 50FFh from memory, **LIT FF00**, the processor started execution right away and begun to pre-fetch the next instruction from 0014h again.

A summary of the major chip features of MSL16A is given in Table 6.5.



Technology	1.2 $\mu$ m 2-layer metal CMOS
Chip Size	4.335mm $\times$ 4.671mm
Transistors	66,500
Power Supply	5V
Power Consumption	95.09mW
Performance	33MIPS

**Table 6.5:** MSL16A chip summary

### 6.2.3.1 Performance Estimation

The performance of the stack is critical as almost all instructions push/pop data to/from the stack. The average processing rate is expected to be 33 MIPS because on average the EXECUTE stage delay, the critical path delay, is less than 30ns (assuming the memory is fast enough). This was estimated from the HSPICE simulation results of the ALU. The internal power consumption of the chip, excluding power dissipated in the bonding pads, was found to be 95.09mW at peak by IRSIM.

### 6.2.3.2 Silicon Area

The chip layout is shown in Figure 6.4. All processor components (including bonding pads) were integrated in 4.335mm  $\times$  4.671mm (20.249mm<sup>2</sup>) and all transistors are minimum sized,  $6\lambda \times 2\lambda$  for PFETs and  $3\lambda \times 2\lambda$  for NFETs, except some are sized for higher performance. The area of the two stacks, which was already heavily optimized by manual layout, took up close to half of the total chip area. The layout of the processor is in no way optimal as the routings of the datapath components and control signals were done by the automatic router provided by Magic [41]. The floorplanning of MSL16A, which was done manually, is also far from perfect. The area inside the pad frame is not highly utilized but this was not critical as it was the smallest pad frame suitable for MSL16A on MOSIS's AMI 1.2 $\mu$ m 2-layer metal CMOS process.

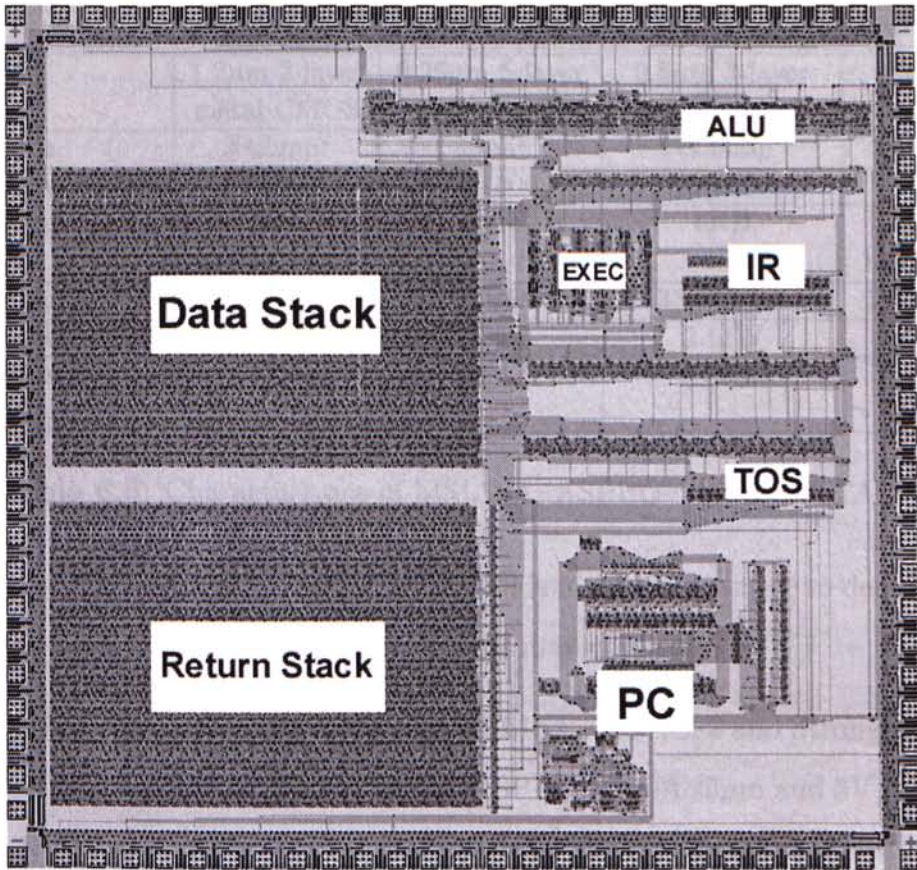


Figure 6.4: MSL16A chip image

### 6.2.3.3 Comparing with other Low Power Processors

As a basis for comparison among the four processors, Table 6.6 shows the characteristics of MSL16A, ASPRO-216 [22], TITAC-2 [9] and StrongARM 110 [13]. The TITAC-2 chip is a 32-bit microprocessor which was fabricated using  $0.5\mu$  CMOS standard cell technology and it occupied  $12.15\text{mm} \times 12.15\text{mm}$ . Similarly, ASPRO-216 is a QDI 16-bit RISC microprocessor targeted on a  $0.25\mu$  five layer metal CMOS technology and it occupied about  $4\text{mm}^2$ . The StrongARM 110 is a commercially available low power 32-bit microprocessor which was fabricated using  $0.35\mu\text{m}$  three-metal CMOS process with a die size of  $50\text{mm}^2$ .

The power consumption is very low although MSL16A contains 66,500 transistors and operates at 5V. The reason is that the pointer stack is a highly energy efficient design, and the two stacks dominate the transistor count and yet only



	MSL16A	ASPRO-216	TITAC-2	StrongARM 110
Process	1.2 $\mu$ m 2-layer metal CMOS	0.25 $\mu$ m 5-layer metal CMOS	0.5 $\mu$ m 3-layer metal CMOS	0.35 $\mu$ m 3-layer metal CMOS
Chip area	20.2mm <sup>2</sup>	4mm <sup>2</sup>	147.6mm <sup>2</sup>	50mm <sup>2</sup>
No. of transistors	66,500	400,000	496,400	2,500,000
Cache size	N/A	N/A	8KB	16KB data 16KB instruction
Performance	33MIPS@5V	200MIPS@3V	52.3MIPS@3.3V	185MIPS@1.65V
Dissipation	95.09mW	0.5W	2.11W	450mW
Power-Delay Product	2.882nJ	2.5nJ	40.34nJ	2.432nJ

**Table 6.6:** Characteristics of MSL16A, ASPRO-216 and TITAC-2

the active stack element consumes power. This is a good example to demonstrate the potential benefits asynchronous designs may enjoy.

For a fair comparison, the results of MSL16A, TITAC-2 and StrongARM 110 are scaled to the same technology as the ASPRO-216 (0.25 $\mu$ m and 3V) based on the *short channel devices general scaling model*. This is the most realistic model for today's situation as voltages and dimensions scale with different factors [61]. The performance is scaled by a factor of  $1/S$  (dimension) while the power consumption is scaled by a factor of  $1/U^2$  (voltage). Table 6.7 shows the scaled results.

	MSL16A	ASPRO-216	TITAC-2	StrongARM 110
Chip area	0.876mm <sup>2</sup>	4mm <sup>2</sup>	36.9mm <sup>2</sup>	25.51mm <sup>2</sup>
Performance	158.4MIPS	200MIPS	104.6MIPS	259MIPS
Dissipation	34.23mW	0.5W	1.744W	1.488W
Power-Delay Product	0.2161nJ	2.5nJ	16.67nJ	5.745nJ

**Table 6.7:** Scaled performance of MSL16A, ASPRO-216 and TITAC-2

It can be easily seen that, when scaled to the same technology, MSL16A is much smaller than the TITAC-2, ASPRO-216 and StrongARM 110. This can be deduced from the fact that both ASPRO-216 and TITAC-2 used at least six



times more transistors compared to MSL16A. MSL16A can be smaller not only because of its simple architecture, but also its high code density. 90% of the 2.5 million transistors in StrongARM 110 are in the two 16KB caches. Because of the high code density and minimal instruction set of the architecture, MSL16A does not require any cache to improve performance. This is why MSL16A could be implemented by using much less transistors.

The performance of MSL16A is lower in terms of MIPS but it is not a very good performance measure. MIPS depends on both the instruction set design and the instruction frequencies. ASPRO-216 provides a way to embed a specific hardware unit into the processor core while adding the instructions which control it in the instruction set and there are 64 possible custom instruction slots. The instruction set of the TITAC-2 is similar to MIPS R2000 and the StrongARM 110 implements the ARM<sup>2</sup> V4 instruction set. The four processors listed have very different instruction sets and the MIPS figures are provided for reference only.

The power dissipation of MSL16A is much lower than that of the other two processors and the lower transistor count cannot be the only reason to account for this. The power consumption is much more than six times lower than ASPRO-216. The pointer stack is a very low power design and the transistor count is dominated by the data and return stacks. In fact, the number of active transistors during run time is much smaller than 66,500 and thus the power consumption of MSL16A can be 14 times lower than ASPRO-216. On the other hand, simulation results showed the power of StrongARM 110 is dominated (43%) by its two large 16 kB caches. Again, as cache was not implemented in MSL16A and each 16-bit access fetches four instructions, the power of MSL16A is very low.

The power-delay products of TITAC-2, ASPRO-216 and StrongARM 110

---

<sup>2</sup>ARM and StrongARM are registered trademarks of Advanced RISC Machines Ltd.

are higher as their architectures are more complex. TITAC-2 is a 32-bit processor with a 5-stage pipeline while ASPRO-216 supports out-of-order execution. StrongARM 110 is a 32-bit processor with features like load and store multiple instructions, auto-increment and auto-decrement for loads and stores, and conditional execution of all instructions. These make the energy required for each instruction higher compared to MSL16A. Low power-delay product means that MSL16A is a fast and energy efficient architecture. Moreover, MSL16A is small and these features make it eminently suited to low power embedded applications.

### 6.3 Summary

The FPGA based implementation of MSL16 was first presented. The results were encouraging and motivated the development of an asynchronous re-implementation. This chapter detailed the evaluation of the three stack designs presented in Chapter 5. The Pointer Stack was found to be the fastest and most power efficient. The Pointer Stack performed at least 20% faster and consumed at least 12.9% less energy than the other two designs in HSPICE simulations. Generating circuit layouts with CAST required less development effort but those layout were found to be about seven times larger than manual layout. Control circuits in MSL16A were realized automatically with CAST and all datapath elements were optimized by hand.

The performance estimation of MSL16A and a comparison with three other asynchronous processors, including a commercially available low power processor, were also presented in this chapter. The final layout of MSL16A was functionally verified using IRSIM. MSL16A was targeted to a  $1.2\mu\text{m}$  2-layer metal CMOS process and contains 66,500 transistors at the size of  $20.2\text{mm}^2$ . The estimated performance was 33 MIPS at 5V consuming a maximum of 95.09mW power. For a fair comparison, MSL16A, ASPRO-216, TITAC-2 and StrongARM 110 were

---

scaled to the same technology. MSL16A was found to consume the least amount of energy per instruction because of its high code density, simple instruction set and architecture. These showed that MSL16A is a fast and energy efficient design and meets the size and power limitations of portable devices.



# Chapter 7

## Conclusions

MSL16 was targeted for low power embedded applications as it offered good code density, high performance at a small area which meet the power consumption and size constraints of such applications. This thesis presented an asynchronous re-implementation of MSL16, called MSL16A, which is a quasi delay-insensitive Forth microprocessor developed based on Martin's synthesis method.

Previous low power asynchronous processors have applied mainly low level power efficiency techniques to maintain code compatibility with previous synchronous architectures. In this work, architectural considerations were explored to achieve low power consumption. The most radical feature was the use of 4-bit instructions to achieve high code density and low memory bandwidth. This is beneficial to power efficiency as the energy consumed in the memory system is in proportion to the amount of code that must be fetched and the memory may consume 50% of the total power of a low power system. The instruction set is minimal and the architecture of MSL16 was designed to efficiently execute the Forth language, which has a higher code density than C or assembly language. An asynchronous re-implementation of MSL16 was undertaken to explore the potentials of asynchronous logic for low power applications and to demonstrate the feasibility and practicability of using asynchronous circuits to meet the cost

and power constraints of the embedded market.

MSL16A was also developed to demonstrate the feasibility of building a low power embedded processor using asynchronous design techniques. The MSL16A design employs a two stage pipeline and incorporates a simple ripple-carry asynchronous ALU adder with a data dependent propagation time which gives mean performance at a low hardware cost. This architecture allows the use of slower memory and no cache without significantly degrading its performance. It has been shown that large caches are costly in terms of silicon area and power consumption. By eliminating the requirement of cache memory for performance, the chip power of MSL16A could be greatly reduced.

The stack design was critical in MSL16A as each instruction execution involves the use of the data stack or the return stack and no previous work on comparisons of asynchronous stack designs has been reported. Three custom designed asynchronous stacks, namely the Eager, Lazy and Pointer Stacks, were described and analyzed. It was shown that the Pointer Stack was the best solution for MSL16A as it was the most power and energy efficient of the three designs, but slightly larger in area than the Eager Stack.

The estimated performance, power consumption and chip area showed that the MSL16A design has advantages over previous designs. MSL16A contains about 66,500 transistors incorporated in  $20.2\text{mm}^2$  and the expected performance is 33 MIPS, using  $1.2\mu\text{m}$  CMOS technology, for a power consumption of 95mW. Compared with a commercial low power processor, StrongARM 110 and two previously reported asynchronous microprocessors, the ASPRO-216 and ECSTAC, it was shown that, when scaled to the same technology, MSL16A is the smallest, having the lowest power-delay product and lowest power consumption among the four processors. MSL16A's small area and low power-delay product indicate it is both area and energy efficient and thus well suited to low power applications. The success of MSL16A is attributed to its simple stack based architecture



and high code density. It is clear that such an architecture closely matches the highly constrained power and cost requirements of battery operated, portable and embedded applications.

## 7.1 Future Work

The layout of the MSL16A processor presented in the thesis could certainly be improved, resulting in smaller area. A smaller chip would definitely help reduce the total system cost, which is a highly constrained factor of portable systems. In order to reduce the area of MSL16A, better floorplanning and layout of MSL16A are required.

The stacks consumed most of the chip area although the layout was made manually. One possible approach to reduce the stack size is to transmit data using the bundled-data technique (presented in Chapter 2). This would result in a smaller stack but this would also violate the delay assumption made by the quasi delay-insensitive circuit style (unbounded wire delays). In a bundled-data system, the datapath uses a single wire for each bit and provides a single timing control signal. The control signal must be greater than longest delay of any of the data bits and some suitable safety margin are generally added but this would lower the performance of the processor. This would be a good way to investigate the tradeoffs between area and performance of the two data representation techniques.

Interrupts or exceptions should be added to MSL16 since they are of high importance in embedded systems. Stack machines have good interrupt performance since the cost of state saving and restoring, and returning to the interrupted routine is low. RISC machines must go through a complex pipeline saving and restoring procedure to avoid losing information of partially executed instruc-



tions. Moreover, some registers are usually saved to create working room for the interrupt routine in the register file. MSL16 has a simple pipeline and only the address of the next instruction needs to be stored. The data or return stack do not need to be saved and the interrupt service routine may push its information on top of the stacks without destroying the interrupted program. This means that MSL16 can treat an interrupt as a hardware generated procedure call. As the cost of procedure calls is very low in MSL16, fast interrupt processing time can be achieved.

The overall benefits of asynchronous designs are hard to qualify. Although they gave advantages in speed and perhaps power consumption, they are larger in area and more difficult to design. A synchronous implementation of MSL16A using the same technology would allow for a direct comparison between synchronous and asynchronous design methodologies.

# Bibliography

- [1] D. Liu and C. Svensson, "Power consumption estimation in cmos vlsi chips," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 663–670, June 1994.
- [2] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The design of an asynchronous microprocessor," in *Advanced Research in VLSI* (C. L. Seitz, ed.), pp. 351–373, MIT Press, 1989.
- [3] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee, "A 100-MIPS GaAs asynchronous microprocessor," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 43–49, 1994.
- [4] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings, "The design of an asynchronous MIPS R3000 microprocessor," in *Advanced Research in VLSI*, pp. 164–181, Sept. 1997.
- [5] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: Design of a quasi-delay-insensitive microprocessor," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 50–63, 1994.
- [6] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "AMULET1: A micropipelined ARM," in *Proceedings IEEE Computer Conference (COMPCON)*, pp. 476–485, Mar. 1994.

- 
- [7] S. V. Morton, S. S. Appleton, and M. J. Liebelt, "ECSTAC: A fast asynchronous microprocessor," in *Asynchronous Design Methodologies*, pp. 180–189, IEEE Computer Society Press, May 1995.
- [8] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver, "AMULET2e: An asynchronous embedded controller," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 290–299, IEEE Computer Society Press, Apr. 1997.
- [9] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model," in *Proc. International Conf. Computer Design (ICCD)*, pp. 288–294, Oct. 1997.
- [10] S. B. Furber, J. D. Garside, and D. A. Gilbert, "AMULET3: A high-performance self-timed ARM microprocessor," in *Proc. International Conf. Computer Design (ICCD)*, pp. 247–252, Oct. 1998.
- [11] K.-R. Cho, K. Okura, and K. Asada, "Design of a 32-bit fully asynchronous microprocessor (FAM)," in *Proc. of the Midwest Symposium on Circuits and Systems* (R. W. Newcomb, B. Geller, and M. E. Zaghloul, eds.), pp. 1500–1503, IEEE Computer Society Press, Aug. 1992.
- [12] T. D. Burd and R. W. Brodersen, "Processor design for portable systems," *Journal of VLSI Signal Processing*, vol. 13(2/3), pp. 203–222, August/September 1996.
- [13] J. Montanaro and e al, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1703–1714, Nov. 1996.
- [14] P. H. W. Leong, P. K. Tsang, and T. K. Lee, "A FPGA based Forth microprocessor," in *IEEE Symposium on FPGAs for Custom Computing Ma-*



*chines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 254–255, IEEE Computer Society Press, Apr. 1998.

- [15] Forth Inc, “Federal express supertracker.” <http://www.forth.com>, 1997.
- [16] *IEEE Standard 1275-1994 — Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*. IEEE, 1994. IEEE 1275 Technical Committee.
- [17] J. Rash, “Space related applications of forth.” <http://groucho.gsfc.nasa.gov/forth>, 1997.
- [18] C. H. Moore, “The evolution of FORTH, an unusual language,” *Byte: the small systems journal*, vol. 5, pp. 76–92, Aug. 1980.
- [19] P. Koopman, *Stack computers : the new wave*. Ellis Horwood series in computers and their applications, Chichester : E. Horwood, 1989.
- [20] C. H. Ting and C. H. Moore, “Mup21 – a high performance misc processor,” *Forth Dimensions*, (also available at <http://www.dnai.com/jfox/mup21.html>), Jan. 1995.
- [21] A. J. Martin, “Programming in VLSI: From communicating processes to delay-insensitive circuits,” in *Developments in Concurrency and Communication* (C. A. R. Hoare, ed.), UT Year of Programming Series, pp. 1–64, Addison-Wesley, 1990.
- [22] M. Renaudin, P. Vivet, and F. Robin, “ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor,” in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 22–31, 1998.
- [23] S. Hauck, “Asynchronous design methodologies: An overview,” *Proceedings of the IEEE*, vol. 83, pp. 69–93, Jan. 1995.

- [24] G. Gopalakrishnan and P. Jain, "Some recent asynchronous system design methodologies," Tech. Rep. UUCS-TR-90-016, Dept. of Computer Science, Univ. of Utah, Oct. 1990.
- [25] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. A. Conrad, D. E. Dever, B. Gieseke, S. M. N. Hassoun, G. W. Hoepfner, K. Kuchler, M. Ladd, B. M. Leary, L. Madden, E. J. McLellan, D. R. Meyer, J. Montanaro, D. A. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam, "A 200-MHz 64-bit dual-issue CMOS microprocessor," *Digital Technical Journal of Digital Equipment Corporation*, vol. 4, pp. 35–50, Fall 1992.
- [26] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits.," *IEEE Transactions on Computers*, vol. C-22, pp. 421–422, Apr. 1973.
- [27] C. Mead and L. Conway, *Introduction to VLSI Systems*. London: Addison-Wesley, 1980.
- [28] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, John Wiley & Sons, Inc., 1969.
- [29] D. A. Huffman, "The synthesis of sequential switching circuits," in *Sequential Machines: Selected Papers* (E. F. Moore, ed.), Addison-Wesley, 1964.
- [30] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Advanced Research in VLSI* (W. J. Dally, ed.), pp. 263–278, MIT Press, 1990.
- [31] J. Liu, "The design of an asynchronous multiplier," MSc Thesis., Department of Computer Science, University of Manchester, Sept. 1995.
- [32] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, Aug. 1978.

- 
- [33] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, June 1994.
- [34] S. B. Furber, *VLSI RISC Architecture and Organization*. USA: Dekker, 1989.
- [35] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.
- [36] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA: Morgan Kaufmann Publishers, 1994.
- [37] P. Koopman, "Why stack machines?," *Computer Architecture News*, vol. 21(1), Mar. 1993.
- [38] L. Brodie, *Starting Forth: an introduction to the Forth language and operating system for beginners and professionals*. Prentice Hall, 1981.
- [39] B. Muench and C. Ting, "eforth 1.0 source code." <ftp://ftp.forth.org/pub/Forth/compilers/cross/eForth>, 1990.
- [40] *ANS Forth Standard – document X3.215-1994*. American National Standards Institute, 1994. X3J14 Technical Committee.
- [41] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI layout system," in *ACM IEEE 21st Design Automation Conference*, (Los Angeles, Ca., USA), pp. 152–159, IEEE Computer Society Press, June 1984.
- [42] A. J. Martin, "A synthesis method for self-timed VLSI circuits," in *Proc. International Conf. Computer Design (ICCD)*, (Rye Brook, NY), pp. 224–229, IEEE Computer Society Press, 1987.



- [43] A. Salz and M. Horowitz, "IRSIM: An incremental MOS switch-level simulator," in *Proceedings of the 26th ACM/IEEE Design Automation Conference* (A. S. IEEE, ed.), (Las Vegas, NV), pp. 173–178, ACM Press, June 1989.
- [44] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Harvard University Press, Apr. 1959.
- [45] M. Shams, J. Ebergen, and M. Elmasry, "A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element," in *International Symposium on Low Power Electronics and Design*, pp. 93–96, Aug. 1996.
- [46] M. Shams, J. C. Ebergen, and M. I. Elmasry, "Optimizing CMOS implementations of the C-element," in *Proc. International Conf. Computer Design (ICCD)*, pp. 700–705, Oct. 1997.
- [47] S. B. Furber and O. A. Petlin, "Designing C-elements for testability," Tech. Rep. Technical Report UMCS-95-10-2, Department of Computer Science, University of Manchester, 1995.
- [48] M. Valencia, M. J. Bellido, J. L. Huertas, A. J. Acosta, and S. Sanchez-Solano, "Modular asynchronous arbiter insensitive to metastability," *IEEE Transactions on Computers*, vol. 44, pp. 1456–1461, Dec. 1995.
- [49] M. B. Tasic, M. K. Stojcev, and G. L. Djordjevic, "Asynchronous controller for token-ring mutual exclusion: delay-insensitive arbiter cell," in *Proc. of the 21st International Conference on Microelectronics*, pp. 819–822, Sept. 1997.
- [50] M. B. Tasic, M. K. Stojcev, and G. L. Djordjevic, "Asynchronous controller for token-ring mutual exclusion: ring design," in *Proc. of the 21st International Conference on Microelectronics*, pp. 823–826, Sept. 1997.

- [51] S. M. Mahmud and S.-U. Alam, "A new arbitration circuit for synchronous multiple bus multiprocessor systems," in *Proceedings of the First International Conference on Systems Integration* (R. T. Ng, Peter A.; Ramamoorthy, C.V.; Seifert, Laurence C.; Yeh, ed.), (Morristown, NJ), pp. 57–62, IEEE Computer Society Press, Apr. 1990.
- [52] A. Yakovlev, A. Petrov, and L. Lavagno, "A low latency asynchronous arbitration circuit," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 372–377, Sept. 1994.
- [53] Man pages for Caltech Asynchronous Synthesis Tools, 1993.
- [54] J. D. Garside, "A CMOS VLSI implementation of an asynchronous ALU," in *Asynchronous Design Methodologies* (S. Furber and M. Edwards, eds.), vol. A-28 of *IFIP Transactions*, pp. 181–207, Elsevier Science Publishers, 1993.
- [55] A. J. Martin, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, pp. 119–137, July 1992.
- [56] D. J. Kinniment, "An evaluation of asynchronous addition," *IEEE Transactions on VLSI Systems*, vol. 4, pp. 137–140, Mar. 1996.
- [57] D. Johnson and V. Akella, "Design and analysis of asynchronous adders," *IEE Proceedings, Computers and Digital Techniques*, vol. 145, no. 1, pp. 1–8, 1998.
- [58] D. J. Kinniment, J. D. Garside, and B. Gao, "A comparison of power consumption in some CMOS adder circuits," in *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 106–118, 1995.
- [59] M. A. Franklin and T. Pan, "Performance comparison of asynchronous adders," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 117–125, Nov. 1994.

- 
- [60] T. K. Lee, *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1995. Technical report CS-TR-95-07.
- [61] J. M. Rabaey, *Digital Integrated Circuits A Design Perspective*. Upper Saddle River, USA: Prentice-Hall Verlag, 1996.



# Publications

- P.H.W. Leong, P.K. Tsang, and T.K. Lee. A FPGA based Forth microprocessor. In Kenneth L. Pocek and Jeffery Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 254–255, Los Alamitos, CA, April 1998. IEEE Computer Society Press.
- P.K. Tsang, C.C. Cheung, K.H. Leung, T.K. Lee, and P.H.W. Leong. MSL16A: An Asynchronous Forth Microprocessor. In *Proc. of the IEEE Region 10 International Conference (TENCON)*, volume 2, pages 1079–1082, September 1999.



CUHK Libraries



003803841