1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395 rue Wellington
Ottawa (Ontario)
K1A 0N4

Vous êtes un client régulier

Vous êtes un client régulier

NOTICE

AVIS

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

If pages are missing, contact the university which granted the degree.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

MICROPROCESSOR DESIGN FOR INSTRUMENT CONTROL

by

PETER M. NYASULU, B.Sc. (Eng)

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Engineering**

Department of Electronics

Carleton University

Ottawa, Canada

August 25, 1995

© copyright

1995, Peter M. Nyasulu

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Veuillez noter : Votre exemplaire

Our file : Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-08963-0

Name PETER NYASULU

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

PHILOSOPHY, RELIGION AND THEOLOGY

SUBJECT TERM

0544

U-M-I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical
Business Administration 0310
General 0272
Accounting 0770
Banking 0454
Management 0338
Marketing 0385
Canadian Studies
Economics 0501
General 0503
Agricultural 0505
Commerce Business 0508
Finance 0509
History 0510
Labor 0511
Theory 0358
Folklore 0366
Geography 0351
Gerontology
History 0578
General

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0723
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry 0485
General 0749
Agricultural 0486
Analytical 0487
Biochemistry 0488
Inorganic 0738
Nuclear 0490
Organic 0491
Pharmaceutical 0494
Physical 0495
Polymer 0754
Radiation 0405
Mathematics
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

MICROPROCESSOR DESIGN FOR INSTRUMENT CONTROL

Submitted by
Peter M. Nyasulu, B.Sc. (Eng)
in partial fulfillment of the requirements
for the degree of Master of Engineering.

Chair, Department of Electronics

Thesis Supervisor

Carleton University

August, 1995

The information used in this thesis comes in part from the research program of Dr. Tadeusz A. Kwasniewski and his associates in the VLSI in Communications group. The research results appearing in this thesis represent an integral part of the ongoing research program. All research results in this thesis including tables, graphs, and figures but excluding the narrative portions of the thesis are effectively incorporated into the research program and can be used by Dr. Kwasniewski and his associates for education and research purposes, including publication in the open literature with appropriate credits. Matters of intellectual property may be pursued collaboratively with Carleton University and Dr. Kwasniewski where and as appropriate.

ABSTRACT

Microprocessors have become an integral part of most control applications. However, the recent increase in the complexity of general-purpose microprocessors and microcontrollers has aroused the need to design low-cost and programmable architectures for low and medium scale applications. This thesis describes the design and implementation of a low-cost 16-bit RISC microprocessor macro cell for use in instrument control applications. The design is based on the Bar Tester application, an instrument used to characterize semiconductor telecommunication lasers. The processor is designed for reduced chip area and an optimized data path. It is laid out in 1.2 μm standard cell CMOS4S technology, and has a total chip area of 18.46 mm^2 and an internal execution speed of 25 MHz. It has a complexity of about 20,000 transistors. A fully automated synthesis design process has been followed, with VHDL used to produce a modular, semi-programmable and technology-independent design. Synopsys was used for logic design, synthesis and simulation, while the layout was produced using the auto-Place and Route tools in Cadence. The architecture presented in this thesis forms the core for the design of an integrated embedded instrumentation processor.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Tadeusz Kwasniewski, for his guidance and advice throughout my work.

I would also like to thank the sponsors of my scholarship, the Canadian Commonwealth Scholarship and Fellowship Plan, and my Program Manager, Cynthia Taha of Canadian Bureau for International Education for her useful tips during my stay here in Canada as a foreign student.

I also extend my thanks to my wife, Stella, for her patience and encouragement, and acknowledge her assistance in typing this thesis.

Last but not least I would like to thank our parents and families at home, for their prayers and love.

TABLE OF CONTENTS

	Page
ACCEPTANCE SHEET	ii
ABSTRACT	iv
ACKNOWLEDGEMENTS	v
TABLE OF CONTENTS	vi
LIST OF FIGURES	x
LIST OF TABLES	xii
LIST OF ABBREVIATIONS	xiii
 Chapter 1: INTRODUCTION	
1.1 Thesis Motivation and Objectives	1
1.2 Thesis Outline	4
 Chapter 2: DESIGN ISSUES	
2.1 Introduction	6
2.2 Microcontroller Design	
2.2.1 General Organization	7
2.2.2 Selected MCU Hardware Features	8
2.2.3 Instruction Sets	10
2.3 RISC Architectures	
2.3.1 Constraints in Microprocessor Design	12
2.3.2 Microprocessor Performance	13
2.3.3 The RISC Approach	14

2.3.4 Reduced Instruction Set	16
2.3.5 Streamlined Architecture	19
2.3.6 Memory Constraints	26
2.3.7 The KIM20: An Example RISC Microprocessor	30

Chapter 3: GENERAL ARCHITECTURAL DESIGN

3.1 Introduction	32
3.2 Application Considered: Bar Tester	32
3.3 Ideal Design	
3.3.1 Architectural Description	38
3.3.2 Performance	41
3.4. Previous Design	
3.4.1 Architectural Overview and Design Constraints	41
3.4.2 Instruction Set and Execution	44
3.4.3 Performance	44

Chapter 4: PROCESSOR ARCHITECTURAL DESIGN

4.1 Introduction	47
4.2 Programming Model	
4.2.1 Instruction Formats	49
4.2.2 Data Format	51
4.2.3 Internal Registers	52
4.2.4 The Instruction Set	53
4.3 Proposed Hardware Architecture	
4.3.1 The Data Path	56
4.3.2 The Pipeline	57
4.3.3 The Control Unit	59

4.3.4 Register Arithmetic and Logic Unit (RALU)	63
4.3.5 The Register File	64
4.3.6 The Program Counter Unit (PCU)	65
4.3.7 External Interface Unit	65
4.4 Instruction Execution	
4.4.1 Instruction Cycles	67
4.4.2 Compute and Internal Data Movements Instructions	68
4.4.3 Branch and Subroutine Instructions	70
4.4.4 External Memory (I/O) Instructions	70
4.4.5 The RESET Sequence	71
 Chapter 5: LOGIC DESIGN AND VERIFICATION	
5.1 Design Process and CAD Tools	
5.1.1 Design Flow	73
5.1.2 VHDL Behavioral Description and Simulation	75
5.1.3 Logic Synthesis	77
5.1.4 Timing Analysis and Gate-Level Simulation	77
5.1.5 EDIF Interface	78
5.1.6 Layout Design	78
5.1.7 CMC CMOS4S Standard Cell Library	79
5.2 Logic Design	
5.2.1 Detailed Architectural Organization	80
5.2.2 Architectural Components Design and Verification	
5.2.2.1 The Control Unit	82
5.2.2.2 The RALU	88
5.2.2.3 The External I/O Interface Unit	91
5.2.2.4 The Program Counter Unit (PCU)	92

5.2.2.5 The Register File	93
5.3 Timing Analysis	
5.3.1 Logic Delays and Critical Path Determination	94
5.3.2 Processor Timing and Speed Evaluation	94
 Chapter 6: SIMULATION AND LAYOUT DESIGN	
6.1 System Simulation	
6.1.1 System Simulation Architecture	97
6.1.2 RAM, EPROM and CLOCK Simulation	98
6.1.3 Software performance Evaluation	
6.1.3.1 Programming Example	98
6.1.3.2 Performance Evaluation	101
6.1.4 Sample Simulation Results	102
6.2 Layout Design	
6.2.1 Layouts of Functional Blocks	104
6.2.2 Processor Chip Layout	105
 Chapter 7: CONCLUSIONS	
7.1 Thesis Summary and Conclusion	106
7.2 Recommended Future Work	109
 Appendix A1: INSTRUCTION GLOSSARY	114
Appendix A2: DESIGN PHASES SAMPLE OUTPUTS	124
Appendix A3: GATE AND TRANSISTOR COUNT	126
REFERENCES	127
BIBLIOGRAPHY	134

LIST OF FIGURES

Fig.	Description	Page
2.1	Block Diagram of a Typical Microcontroller	7
2.2	Simple RISC Data Path	20
2.3	Four-Stage Pipeline Model	24
2.4	Register Windowing	27
2.5	Havard and von Neumann Architectures	29
2.6	KMI20 Data Path	30
2.7	KMI20 Single Instruction Format	31
3.1	Bar Tester System Architecture	35
3.2	Previous Design Processor Architecture	42
4.1	Instruction Formats	49
4.2	Data Format	51
4.3	Register Programming Model	52
4.4	The Data Path	56
4.5	The Instruction Pipeline	58
4.6	The Control Unit	61
4.7	The Register Arithmetic and Logic Unit (RALU)	63
4.8	The Interface Unit	66
4.9	Instruction Cycle #1	67
4.10	Compute Instruction Cycle #2	68
4.11	Compute Instruction Cycle #3	69
4.12	Compute Instruction Cycle #4	69
4.13	External Instructions Cycle #3	71
5.1	Design Flow	74

5.2	Detailed Processor Architecture	81
5.3	The Control Unit Logic Diagram	82
5.4	Pipeline Registers Logic Diagram	85
5.5	Clocking and Wait State Insertion	87
5.6	RALU Logic Diagram	89
5.7	External Interface Unit Logic Diagram	91
5.8	PC Unit Logic Diagram	92
5.9	Register File Logic Diagram	93
5.10	Processor Internal Timing	95
6.1	System Simulation Architecture	97
6.2	EPROM and CLOCK VHDL Behavioral Design	99
6.3	Example Test Program	100
6.4	Sample Simulation Results	103
6.5	Processor Chip Layout	105
A.1	Branch Decision Circuit Design Phases	125

LIST OF TABLES

Table	Description	Page
2.1	Processor Performance	13
2.2	KIM20 Instruction Set	31
3.1	On-Chip DPSRAM Memory Map	43
3.2	Processor Instruction Set	46
4.1	Instruction Formats Modifications	50
4.2	Opcode Bits Functions	51
4.3	Instruction Set Summary	55
4.4	Undecoded Control Signals	61
4.5	Decoded Control Signals	62
5.1	Processor External Interface Pins	80
5.2	Critical Paths Delays	94
6.1	Processor Performance Evaluation	101
6.2	Layouts of Functional Blocks	104
A.1	Processor Gate and Transistor Count	126

LIST OF ABBREVIATIONS

A/D(C)	Analog to Digital (Converter)
ALE	Address Latch Enable
ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
CAD	Computer-Aided Design
CISC	Complex Instruction Set Computer
CIU	Code Interface Unit
CLA	Carry Lookahead
CMOS	Complementary Metal Oxide Semiconductor
CPI	Clock Cycles Per Instruction
CPU	Central Processing Unit
D/A(C)	Digital to Analog (Converter)
DIU	Data Interface Unit
DMA	Direct Memory Access
DPSRAM	Dual Port Static Random Access Memory
DRAM	Dynamic Random Access Memory
DTACK	Data Acknowledge
ECL	Emitter Coupled Logic
EDIF	Electronic Design Interchange Format
EPROM	Erasable Programmable Read Only Memory
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GaAs	Gallium Arsenide

GPIB	General Purpose Interface Board
GPIO	General Purpose Input Output
IEEE	Institute of Electrical and Electronic Engineers
I/O	Input/Output
IPS	Instructions Per Second
IR	Instruction Register
LSSD	Level-Sensitive Scan Design
MCU	Microcontroller Unit
MIPS	Million Instructions Per Second
MPC	Microprogram Controller
NOP	No Operation
PC(U)	Program Counter (Unit)
PLA	Programmable Logic Array
PLD	Programmable Logic Devices
RALU	Register Arithmetic and Logic Unit
RAM	Random Access Memory
Rd	Destination Register
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SGE	Synopsys Graphical Environment
SP	Stack Pointer
SRAM	Static Random Access Memory
TTL	Transistor-Transistor Logic
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
VSS	VHDL System Simulator

Chapter 1

INTRODUCTION

1.1 THESIS MOTIVATION AND OBJECTIVES

There have been tremendous advances in microprocessor architectures since their first appearance in the 1970s. This evolution has been mainly in the area of computer microprocessors, in which the need to support a wide range of the ever-growing high-level languages is a major influencing factor. Microprocessor designers have also taken advantage of the increasing potential of silicon implementation technologies to implement advanced architectures and complex instruction sets. The underlying objective of this development has been to design microprocessors that are as powerful and as general-purpose as possible. However some of this progression has been rather more historical and technology-oriented than induced by scientific analyses or advances in application needs. It is therefore not surprising that the pioneering 4-bit and most of the 8-bit microprocessors have become extinct [39]. Even 16-bit microprocessors are being rapidly replaced by 32-bit and 64-bit units.

On the other hand, microcontrollers, used in instrumentation and other control applications, have not undergone a change as absolute as computer microprocessors. Actually, most of the advances in microcontrollers have been brought about by the fact that architectures designed for computer microprocessors are adapted for microcontroller design, rather than designing them from scratch. Therefore, unlike the other microprocessors, 4-bit microcontrollers are still used today because they are smaller and hence inexpensive single-chip solutions to a multitude of control tasks such as toys, appliances and simple instruments. Also, 8-bit and 16-bit microcontrollers are more widely accepted than

their corresponding microprocessors.

It is therefore apparent that while these advanced microprocessors may be necessary for computers and other control systems, most instrumentation control applications still require just the basic microprocessor. Their application codes use only a very small portion of the complex instruction sets provided on advanced microprocessors and general-purpose microcontrollers. The high speed of these processors, which is achieved by sophisticated architectural features such as superscalar [24, 31] and superpipelined [66, 43] designs, and by using expensive high-speed technologies such as GaAs [56] and BiCMOS [32], is not a must for most instrumentation control applications. Also, most of the capabilities of the highly integrated (and hence expensive) general-purpose microprocessors, such as expensive memory subsystems and floating-point arithmetic, are not utilized in most embedded control applications [44]. This gap between the microprocessor and the instrumentation control application needs has fueled a lot of interest in the design of embedded instrumentation control processors. The goal in such designs has been to produce a cost-effective processor for the application, while maintaining or improving the performance achieved by using a conventional microprocessor or microcontroller. Generally, the embedded microprocessor market is quite diverse and hence a family of chips is needed to support it. Therefore embedded processor cores, such as the Intel 80960-Next Generation [12], are designed to be modular to adapt to different applications. Application-optimized chips can then be made by adding application-oriented modules to this common high-performance low-cost core. Processors targeted for such applications are also usually designed for low-power and reduced complexity and hence reduced real estate because increased die area means greater power consumption and higher chip costs [1, 2, 5, 6]. In this thesis, these same goals are pursued. The main thesis objective is to explore cost-effective architectures for instrumentation control processors by designing such a low-cost, low-power processor core for a typical instrumentation control application around which different small- and medium-sized embedded instrumentation control

processors may be developed.

The application considered in this thesis is the Bar Tester system. This test instrument was developed by Michael Gazier [41] for the optoelectronics device group at Northern Telecom. Its purpose is to characterize semiconductor telecommunication lasers. It requires intensive low-level control, and has a number of circuit boards. It uses the Motorola MC68000 processor, operating at 10 MHz. This application was chosen because about a year ago, research was conducted by its author [41] to design a low-cost integrated embedded instrumentation processor for the Bar Tester system. The goal was to attain performance of the same order as the system's discrete component hardware, and also to design an open architecture to allow future upgrades. Even though the final design (hereafter referred to as the previous design) was not laid out and thoroughly simulated, intensive considerations were carried out for the initial data and control paths design. With first-hand experience, and a detailed study of the original MC68000 application code, the instruction set of the processor to be designed was produced.

This thesis took advantage of the existing design by assuming all the initial processor specifications and design constraints, without having to make an independent and lengthy detailed study of the application and its original code. However, completely different data and control paths are designed to further reduce the silicon real estate and increase the execution speed. To achieve this, the processor is designed with a 4-stage pipelined RISC architecture, employing hardwired control, as opposed to the semi-RISC microprogrammed control architecture used in the previous design. The control and execution algorithms used in the two designs are therefore very different. The input/output circuitry and the external interface is also designed differently. The new processor has separate memory spaces for code and data, and therefore uses two multiplexed buses (multiplexed Harvard) as opposed to the non-multiplexed von Neumann architecture used in the previous design. The instruction set has also been fundamentally revised, with the new set designed with formats and addressing modes which enhance RISC single-cycle execution

and efficient pipelining and decoding. Only a few elements of the previously designed instruction set, such as similarity to the MC68000 instructions, are retained. The design has been thoroughly simulated, the final chip laid out in 1.2 μm CMOS technology, and its performance compared to the initial system and the previous design. The designed processor runs at 25 MHz, has a complexity of about 20,000 transistors, and occupies a die area of 18.46 mm^2 .

After the design of the processor core, a number of other issues will be explored. These include the on-chip implementation of dynamic power management techniques, the extension of the on-chip data storage capabilities (RAM, ROM, registers, etc.), the determination of what and how much of the other components (timers, Digital to Analog Converters. Analog to Digital Converters, etc.) should be integrated on the same chip of a typical instrumentation control processor, and the design of programmable architectures. It is also important, as in most application specific designs, to make a detailed cost analysis before deciding to design and use a specifically designed processor like the one presented in this thesis, as opposed to using microcontrollers currently available on the market. Most of these issues however require further analysis and more time, and therefore will be left for future research work on the design.

1.2 THESIS OUTLINE

Chapter 1 is the introduction, outlining thesis motivation and objectives, as well as the thesis outline. The second chapter gives a brief description of the main issues relating to the architectural design of the processor presented in Chapter 4. Section 2.2 presents microcontroller general organization, hardware features and instruction sets. Following this is a relatively complete but brief coverage of RISC architectures. Issues such as RISC instruction sets, streamlined architectures and memory constraints are reviewed. An example of RISC microprocessors is also presented at the end of this chapter. Chapter 3

outlines the architecture and performance of the application, the ideal design, and the previous design, and therefore shows the influence of these designs on the current design. Design constraints and limitations are also outlined, and the designs are compared. The fourth chapter is a detailed description of the proposed processor architecture. Section 4.2 describes the programming model: data and instruction formats, instruction groups, internal registers and the complete instruction set. The proposed architecture is presented in section 4.3. This includes the data path, the pipeline model, the control unit, the Register Arithmetic and Logic Unit (RALU), the register file, the Program Counter Unit (PCU) and the interface circuitry. Section 4.4 gives a detailed description of instruction cycles and execution. Chapter 5 focuses on the actual logic design process. Issues covered include design tools, logic design of the processor and its architectural components, timing delays, clocking scheme and processor speed evaluation. Chapter 6 is a description of the system simulation and final layout design. This also includes behavioral simulation of components such as RAM, EPROM and the clock. Sample simulation results and performance evaluations are also included. The chapter concludes by giving the layout areas of the functional blocks, as well as the final processor chip layout. In chapter 7, thesis summary and conclusions are presented. This is followed by a suggested course of future work. The Appendices include a detailed processor instruction glossary, design (CAD) phases sample outputs, and the processor gate and transistor count.

Chapter 2

DESIGN ISSUES

2.1 INTRODUCTION

A number of issues have influenced the design of the Bar Tester Control processor. The original application code [41] shows that only a small and simple instruction set is required. Microprocessors such as MC68000 are designed for computer CPUs and usually have large and complex instruction sets to support high-level programming languages. Their performance direction emphasizes increased computing power and ability to store more data. On the other hand, this design emphasis is on integration, reduced chip area and flexible interface to external devices. Performance is optimized for a specific application, even though some design features have been incorporated to enable future upgrades and versatility.

It is therefore apparent that the overall design criteria follow that of a typical Microcontroller Unit (MCU). A brief overview of the design strategy for Microcontrollers is thus presented in section 2.2. The inherent small instruction set and the need for reduced chip area and application-specific optimization leads to the choice of a RISC (Reduced Instruction Set Computer) architecture. This choice is further driven by the widely accepted and significant performance advantage of RISC processors over CISC (Complex Instruction Set Computer) processors [43, 44]. The processor architecture and programming model closely follows this streamlined design approach. Principles of RISC architectures are previewed in section 2.3. A brief description of the KIM20 microprocessor [43] is included at the end of the chapter to highlight the application of these principles in the design of a typical application-specific RISC microprocessor.

2.2 MICROCONTROLLER DESIGN

2.2.1 General Organization

MCU's differ from other microprocessors mainly in I/O capability and instruction sets. Because the design emphasis in MCUs is on their interaction with the environment, both in sensing and control, much more real estate (in the form of I/O pins and chip area) is dedicated to I/O. For this reason, a typical MCU would incorporate communication devices, A/D and D/A converters, as well as timers on a single chip. Secondly, many of the instructions are oriented towards bit manipulation and testing, reflecting the need to perform logic operations on I/O ports on a single-bit basis. Fig 2.1 is a functional block diagram of a typical microcontroller unit.

Fig 2.1 Block Diagram of a Typical Microcontroller

2.2.2 Selected MCU Hardware Features

2.2.2.1 Input / Output

Microcontrollers use different methods for accessing input/output ports. The two most used approaches are memory-mapped I/O and separate I/O. These are typically used by Motorola and Intel, respectively. In memory-mapped I/O, I/O devices have dedicated addresses and thus read/writes to these devices are performed just like for memory. This offers lots of computational advantages in that all mathematical and logical operations that can be performed on the contents of memory can now be performed on I/O ports. For example, the contents of an I/O port can be read, incremented and written back to the I/O port with a single instruction, and data can be moved from one I/O port to another with a single move instruction. Normally, all I/O port addresses are placed in the same block of memory so only one major address needs to be decoded to identify I/O, with the lower-order bits used to access specific I/O ports. Separate I/O has an $\text{IO}/\overline{\text{MEM}}$ control line to select between I/O and memory operations. It is therefore possible to use identical address decoders for both I/O and memory with the final determination of which of these is accessed being determined by the status of the $\text{IO}/\overline{\text{MEM}}$ line.

2.2.2.2 Interrupts

Most microcontrollers implement interrupts in a way typical of other microprocessors. Nonmaskable interrupts are usually reserved for events that threaten the integrity of the microcontroller unit, such as power failure or stack overflow. Maskable interrupts are prioritized. Apart from hardware interrupts, software interrupts (also called exceptions or traps) can also be initiated from any place in a program by executing a software interrupt (SWI) as a line of code. These have the same effect as normal hardware interrupts in terms of automatically saving the machine's state before executing the interrupt routine.

2.2.2.3 ALU

The significant difference in the ALUs of microcontrollers and those of general purpose computers is the invariable inclusion of bit manipulation instructions. This is more efficient than using general logical instructions to perform the same functions. It is also common to find microcontrollers that allow branching based on testing particular bits of an input port as a single instruction. In some cases, separate logic units are included in parallel with the internal arithmetic units to speed execution and allow for parallel execution of instructions. An example of such a unit is a Boolean Processor (BP) that allows individual bits to be set, cleared, complemented, moved, tested and used in logic operations.

2.2.2.4 Timers

Timers are an integral part of control applications. For this reason, most MCUs support versatile timers that can be configured in real time by writing appropriate codes to control registers [40, 44]. These timers can be reprogrammed as to the duration they will measure; the type of interrupt that they will generate if programmed to generate interrupts at regular or deterministic intervals; and whether they will reload and continue to time the next period automatically. They can also be configured in real time to act as event counters or to count the duration between transitions on the inputs.

A second type of timers, called the watchdog timer, is also usually present in MCUs. This timer is used to restart the microcontroller if it fails to reset at regular intervals. When the MCU is operating properly, the executing code clears the timer at a regular programmed basis. Otherwise, it is assumed that the microcontroller has malfunctioned, and the MCU is restarted. This is especially important when the MCU is used in unattended environments.

2.2.2.5 Parallel And Serial I/O

A complete range of binary I/O, both parallel and serial, synchronous and asynchronous, is also available on most MCUs. The object is to allow effective communications with other similar devices in a distributed system of multiple processors and/or communicate easily with the operator. These communication interfaces are designed to be as autonomous as the timers, performing their functions and only interrupting the CPU when they have received a complete byte or block of data or when they are ready to transmit the next byte or block of data.

2.2.3 Instruction Sets

There have been few efforts to implement high-level languages in most MCUs. This has resulted in less complex instruction sets when compared to computer CPUs. Apart from I/O and bit manipulation instructions, MCUs usually support simplified or reduced versions of typical CPU instruction sets.

2.2.3.1 I/O Instructions

A number of MCUs use memory-mapped I/O (a notable difference are MCUs from Intel). This is reasonable when considered in the context of an MCU's use, which is I/O intensive and rarely involves a large memory. Memory-mapped I/O also streamlines the instruction set in that operations, and in some cases direct single-bit operations, can be performed directly on I/O rather than having to use an I/O instruction to bring data into the registers, operate on the data, and then restore the data to the I/O port. An example is the National Semiconductor NS32000 series of MCUs [40]. These microcontrollers have memory-mapped I/O and they have a MOV instruction that can transfer data from memory locations to memory locations without passing through the accumulator and hence I/O data can be transferred from one I/O address to another without the intervening steps of

loading data into the accumulator/registers. This feature is not available in MCUs such as MC6811, which have only load and store accumulator instructions. MCUs also support a number of shift and rotate instructions and bit-clearing and bit-setting instructions. The single-bit shift supported by most general MCUs is adequate for software serial I/O, but is time consuming if multiple bits need to be shifted.

2.2.3.2 ALU Instructions

Computational instructions of most MCUs include arithmetic operations of addition and subtraction and the basic logic operations: AND, OR and XOR. Integer (and in some cases, floating-point) multiply and divide can also be found in some MCUs. Application-specific MCUs, such as the NS32FX [40], usually support more advanced floating-point calculations such as multiply, divide, polynomial, dot product, scaling, vector and array operations, and other application-specific mathematical operations. However, for different MCUs, there is significant difference in the range of registers and memory locations over which these operations can be performed. Some MCUs, such as the NS32000 [40] allow logic operations directly on memory locations.

2.2.3.3 Bit Manipulation Instructions

In control applications, the ability to manipulate individual bits is very important, necessitating the inclusion of extensive bit manipulation capability in most MCUs. This includes instruction to directly clear accumulators and other registers, and set, clear and test individual bits. Like in ALU instructions, the significant difference across MCUs is the range of addresses over which the operations can be performed. Some MCUs, such as the MCS-51 [40], also include the SWAP instruction. SWAP interchanges the two 4-bit nibbles of a byte (or two 8-bit bytes of a word or two 16-bit words of a long word) in a register. This is particularly useful when doing decimal arithmetic, shifting out BCD numbers, or when 4-bit, 8-bit or 16-bit rotates are required.

2.2.3.4 Program Flow Control Instructions

MCUs support general branch and jump instructions as well as subroutine instructions. Branch-to addresses are usually relative to the PC while jump-to addresses are absolute. In addition to these jumps and branches, certain microcontrollers have other advanced program flow control instructions. Again, the NS3200 is an example of such MCUs. It has multiway branch (CASEi) as well as a breakpoint trap (BPT), flag trap (FLAG), supervisor calls (SVC), and calls to external procedures (CXP, CXPd). The CASEi instruction can directly implement such high-level instructions as C's "switch" statement. A control instruction, not common in CPUs but supported by some MCUs is the branch based on the status of individual bits of a memory or I/O address. This, again reflects the importance of bit manipulation in control applications.

2.3 RISC ARCHITECTURES

2.3.1 Constraints in Microprocessor Design

The three main features for designing a new microprocessor have previously been:

- 1. Relationship with High Level Languages:** to provide a set of complex instructions in order to reduce the gap between the machine object code and high-level languages/operating systems.
- 2. Upward Compatibility of Instruction Sets:** to enable programs to be easily transported from one earlier member of the microprocessor family to another one in an upward compatible fashion.
- 3. Processor Performance:** to improve performance, in terms of the speed and efficiency of program execution.

These requirements have resulted in an increase of both horizontal (data path) and vertical (use of microcode) processor complexity.

2.3.2 Microprocessor Performance

In microprocessors, performance is directly related to the time spent executing a given task, and is generally represented as a performance parameter P , which is calculated in terms of the instruction rate and is given by the equation,

$$P = \frac{F}{I \cdot CPI}$$

where,

F is the clock rate, and is equal to the inverse of the clock cycle time or period,

I is the number of instructions executed for the program, and

CPI is the average number of clock cycles per instruction.

If I is in millions of instructions, then P is in Millions of Instructions Per Second (MIPS). Table 2.1 below gives MIPS of some common microprocessors [43].

Microprocessor	Type	I	CPI	F (MHz)	P (MIPS)
Motorola 68030	CISC	1.0	5.2	16.67	3.21
Intel 80386	CISC	1.1	4.4	16.67	3.44
SPARC	RISC	1.2	1.3	16.67	10.69

Table 2.1 Processor Performance

Therefore, in general, performance of a processor is dependent on:

- (i) the number of instructions required to execute a particular program, I
- (ii) the average number of clock cycles required per instruction, CPI .
- (iii) the clock frequency, F

Each of these three characteristics is related to one particular technology domain.

The instruction count (I) is related to the adequacy of the instruction set to execute the

code produced by the compiler. A small instruction count indicates a small gap between the machine code and the high-level language. The average number of clock cycles per instruction (CPI) is related to the basic architectural model of the microprocessor. Most RISC processors require between 1 and 2 clock cycles per instruction, while CISC processors can use up to 10 clock cycles per instruction. The clock rate (F) is directly related to the hardware technology used for the implementation of the processor. The period ($1/F$) can vary from a few nanoseconds for Gallium Arsenide (GaAs) or Emitter Coupled Logic (ECL) based computers, to more than 10 times that for microprocessors using a more conventional technology, such as CMOS (Complementary Metal Oxide Semiconductor).

Some performance measurements refer also to “VAX MIPS”, where the performance of the processor is compared to the time needed by a VAX-780 to execute a given benchmark. By convention, the VAX has a performance of 1 MIPS and the performance metric is a ratio based on this definition. For example, a processor of 3.5 MIPS is 3.5 times faster than the VAX-780 in executing the benchmark.

Performance evaluation is however more complex than is presented here because it also depends on other factors such as benchmark programs, programming language, compiler technology, computer architecture and implementation technology.

2.3.3 The RISC Approach

Most CISC microprocessors are, in general, designed with a bottom-up approach in order to remain compatible with earlier versions and reduce the task of designing a new architecture from scratch. The previous architecture and instruction sets are studied and new instructions and functionalities are added to them for increased performance. On the other hand, the RISC design flow uses a top-down approach summarized in the following five basic steps:

1. Statistical analysis of typical application code in order to find the highly frequent oper-

ations.

2. Definition of the instruction set architecture. The latter must include only simple instructions unless there is a good reason for complexity.
3. Design of the data-path with related operators (ALU, shifters, etc.) in order to efficiently execute the selected instructions.
4. Add other instructions only because they measurably improve performance and do not break the pipeline scheme or increase the cycle time. Otherwise, these functions must be implemented in software as a sequence of simpler instructions.
5. Verify the instruction set performance and completeness by running application code on a software simulator. If some problems remain (poor performance and/or large compiled code), return to step 1.

It must, however, be pointed out that even though downward compatibility is generally considered to be a CISC tradition, most recent RISC processors are also designed to remain compatible with their predecessors [2, 3, 6], and hence employ both the top-down and bottom-up approaches.

The main success of RISC has mainly been due to its performance gains and simpler designs, thus allowing designers to design, verify and manufacture new high-performance products in a relatively short time [4]. The simple instructions in RISC machines allow simple fetch and decoding stages, but also mean that RISC programs require more main memory for storage and more instruction bandwidth for execution than CISC machines [16]. Despite these drawbacks, the RISC approach seems better suited for today's technology because of the inexpensive memory resource in the current technology and through the use of instruction cache to reduce fetch bandwidth requirements [15]. The following list describe the main RISC features that enhance performance through single-cycle execution and keeping the cycle-time as short as possible:

1. Relatively few instructions and single-cycle operations for every instruction.
2. Few addressing modes and a fixed instruction format.

3. Hardwired control without microcode.
4. Migration of complex functions to software.
5. Only Load and Store instructions should reference external memory.
6. Each instruction must fit a pipelined model that allows parts of several operations to be processed at the same time.
7. Delayed-branches enable conditional and unconditional branches to execute without flushing the pipeline.
8. The architecture must have at least thirty-two general-purpose registers and large cache memories.
9. The architecture must be tailored for a dedicated application field.
10. If a RISC processor is to be used for high-level languages, much of the static runtime complexity should be handled by an optimizing compiler.

2.3.4 Reduced Instruction Set

2.3.4.1 Instruction Groups

For most RISC processors, there are three basic instruction groups:

1. **Compute Instructions:** These generally include arithmetic, logical and shift operations. The computed result, which in most cases is a function of two source operands, is placed in a result register.
2. **Control Instructions:** The control group consists of conditional and unconditional branches, explicit call and return operations or jump-and-link operations. Control transfers are usually delayed until after execution of the next instruction, so that the pipeline is not emptied every time a control transfer occurs.
3. **Memory Access Instructions:** Load and Store are the only instructions that access external memory. Variations of these enable operations on different word-lengths. For

simpler hardware implementation, some processors use only word store and load instructions, plus byte extract and insert instructions. This is however less efficient than the former approach. Load and Store are the only instructions that violate the single-cycle constraint, because of the time required to access the memory while fetching the next instruction. One solution to this is to use the Harvard architecture which requires extra buses for data.

Apart from these groups, RISC processors usually incorporate an extra two groups of instructions:

1. **Application-Specific Instructions:** These are dedicated to the selected application or a given high-level language. Examples are instructions designed to efficiently support bit-manipulation for I/O control applications, and an efficient instruction set for symbolic processing.
2. **Miscellaneous Instructions:** This group includes instructions to read and write the contents of various control registers, such as the processor status word, the cache controller, or the memory management unit. It can also include generic coprocessor operations that enables optional coprocessors to execute concurrently with the RISC integer unit. This is used for floating-point calculations, where the integer unit and the floating-point unit (FPU) may or may not be implemented on the same chip.

2.3.4.2 Register-Oriented Instruction Set

To achieve single-cycle execution, the RISC approach specifies two complementary points:

1. **Load-Store, register-to-register intensive design:** all computational instructions involve registers. Memory accesses are made only with Load and Store instructions.
2. **A large register file reduces a program's need to load and store operands to and from external memory.**

Because access to external memory is slower than access to processor registers,

these two points strive to keep all data in registers for as long as possible. A large register set allows a program to keep operands readily available and reduces memory-bandwidth requirements. The load-store design explicitly limits the access to external memory. Researchers at the University of California at Berkeley (founders of the RISC-I and RISC-II architectures) argue that a computer does 70% operand accesses and 30% operations: for each operation, one or two source operands are required and the result is placed into another operand [43]. This shows how important it is for a computer system to have quick access to operands. Register files are usually the fastest storage structures, not only because they are small and usually integrated on the same chip, but also because addressing is made with a much shorter address than for an external memory or a cache system. Since at each cycle the RISC processor must fetch a new instruction, every access to external memory for loading or storing a value potentially breaks the pipeline execution and introduces latencies. This is the reason why RISC tries firstly to keep as many of its operands as possible in registers and, secondly, tries to limit external memory references. These points imply also that the compute instructions operate on registers rather than on external memory. Thus the resulting instruction formats must rely mainly on register addressing for source and destination operands.

2.3.4.3 Reduced Number of Instructions, Formats, and Addressing Modes

To simplify overall design of the RISC architecture, all instructions are made the same size, typically 32-bit, with a few (one to four) simplified fixed formats. This enables implementation of single-cycle execution and hardwired control with reduced design time and silicon real estate. The two typical formats, on which the original Berkeley RISC microprocessors were based, are:

1. **The Register Format:** This format generally includes the OpCode (operation code), a destination register, a first source register and a second source register or a small immediate value. This format enables instructions to use two registers or a register and

- a constant to compute an operation, and then place the result in the destination register.
2. The Immediate format: This enables instructions to include an OpCode, an explicit or implicit destination register, and a long immediate value. This format is generally used to calculate the memory address involved in control-transfer instructions (call, jump, etc.) or memory access and move immediate value instructions.

To further facilitate implementation of single-cycle operation and a short clock rate, RISC processors use a relatively small number of instructions (less than 100) and a small number (one or two) addressing modes. The PowerPC™ 604 [84] supports two simple memory addressing modes: register indirect with immediate index ($rA + \text{offset}$) and register indirect with index ($rA + rB$). This also simplifies the design of the control unit.

2.3.5 Streamlined Architecture

2.3.5.1 The Data Path

The data path, which is the computational part of a processor, consists of registers organized in a register file, the execution units such as the ALU, shifters, etc, and the communication paths between them. From the software perspective, it contains most of the essential data manipulation blocks of the processor. Fig 2.2 shows a simplified diagram of RISC data-path. Three buses are used for routing data between registers and the ALU: Src1 and Src2 are used to read data from the register file and send them to the corresponding inputs of the ALU, DEST takes the result of the ALU computation to be written back to the register file. Therefore, in most RISC architectures, the data path contains three critical paths: the time required to read the two operands from the register file, the time to compute the appropriate operation in the ALU, shifter or other unit, and the time required to store back the result into the register file. Thus, the designer must take care when implementing the register file and the execution units.

Fig 2.2 Simple RISC Data Path

2.3.5.2 Hardwired Control versus Microcode

Control units can be microcoded, i.e control signals read from a microprogrammed ROM or PLA, or they can be hardwired, i.e. implemented with standard gates or logic arrays. Each implementation has its own advantages and disadvantages. A more detailed review of microprogrammed control units, plus their advantages and disadvantages over hardwired controllers, can be found in [28] and [61]. In general, the advantages of micro-coded control units over hardwired control units include:

1. **Reduced design cost and time**, especially when implementing large and complex instruction sets. A hardwired controller requires a custom design whereas a micro-coded design simply requires programming the ROM.
2. **Flexibility**: In microcoded designs, it is easier to change machine language instruction sets and introduce new instructions even after field installation since this simply requires changing the control store ROM. Therefore, the same controller board, chip or slice can be used for different designs. This flexibility is further enhanced if the

ROM is organized as a two-level control structure, in which the microinstructions in the lower level storage unit (also called nanoinstructions) interpret microinstructions in the upper level storage unit in much the same way that ordinary microinstructions interpret machine language instructions. This use of nanocode permits flexibility in the design of upper level microinstructions as well as in the design of machine language instructions [28].

3. **Ease of development and maintenance:** Because of the reduced internal circuitry, microcoded controllers are inherently simpler and thus easier to develop and maintain than hardwired control units. Because of advances in VLSI technology concerning microprogrammed structures such as microcoded ROM's and PLA's, it is also now possible to migrate and embed in microcoded silicon not just traditional microprograms (i.e instruction-set interpretations) but also system software functions such as floating point arithmetic algorithms, high level languages, and even operating system primitives. This vertical migration from the software level to the firmware level of frequently used software not only provides significant advantage in speed, but also increased reliability, security and stability [30]. It is also a good technique to extend the instruction set of a system without much change in the hardware structure.

The disadvantages of microcode include:

1. **Speed:** Microprogrammed controllers are slower than hardwired controllers because of the time required to read microinstructions from the control store.
2. **Cost:** A microcoded design can be costly for a small design because of the minimum size requirement for a ROM.
3. **Defect tolerance redundancy:** In defect tolerant microprocessors, the more regular control units resulting from hardwired PLA implementations allow introduction of defect tolerance redundancy in a much simpler manner [19]. In this case, the control unit is partitioned into PLA's where each PLA can be designed to incorporate defect tolerance through the addition of spare product terms [20, 21]. On the other hand,

while a microprogrammed unit can have its control memory designed with spare rows/columns to support defect tolerance, in much the same way as is done in other memory IC's, its other additional circuits, such as the sequencer, the microcode decoder, and the microinstruction register, cannot employ such simple strategies because of their high degree of irregularity [19].

Most CISC microprocessors use microprogrammed control, while RISC microprocessors usually employ hardwired control. In hardwired control, instructions are decoded and issued directly using standard logic gates or logic arrays such as PLA's. Because of the simplicity and performance of the underlying hardware, hardwired control minimizes the average number of clock cycles required per instruction and the clock cycle time. In most RISC architectures decoding the incoming instruction is particularly easy, because of the simple instruction format with fields of fixed size and position. As an example, the RISC-II control section occupies only 10% of the chip area, compared to 68% for the Motorola MC68000 [43].

Regardless of the fundamental differences between RISC and CISC, and between microcoded and hardwired designs, it has been observed in [30] that RISC instructions are really very similar to vertical microinstructions of microprogrammed machines, except that the control store in vertical microcode is replaced by the instruction cache of RISC. In this sense, RISC appears to be another form of microcode in the control store. For this reason, Papachristou and Immaneni [30] argue that microcode may as well coexist with the hardwired RISC instructions in a real machine architecture. In this case, the microcode is viewed as an "attached" microarchitecture supporting a RISC-based processor to enhance overall performance. This phenomenon has been implemented in some designs such as the Intel 80960-Next Generation embedded control processor core [12], in which the vast majority of all instructions are directly issued like any other RISC machine (i.e. hardwired control), with the microcode used for a few instructions and to support fault, interrupt and debugging handling.

2.3.5.3 Pipelining

Pipelining allows execution of multiple instructions at the same time. Instructions are divided in time in “pipe stages” that can be simultaneously executed by the processor hardware units. At every clock cycle, a new instruction begins, and another one ends. The machine cycle is therefore determined by the slowest stage of the pipeline. In most RISC architectures, this stage is the instruction fetch stage, which accesses the program memory. Pipelines of two to four stages are common. Figure 2.3 shows a four-stage pipeline model, with the following execution order:

1. Fetch and decode the instruction.
2. Read source1 and source2 operands from the register file.
3. Perform an arithmetic, logical or shift operation on the operands.
4. Write back the result into the register file.

The initial RISC-I architecture had a simple two-stage pipeline, overlapping instruction fetch and execution. In this case it is assumed that an instruction-fetch memory cycle takes roughly the same time as a register read-operate-write cycle. RISC-II, and many other RISC processors such as KIM20 [43] use a three stage pipeline, the third stage being used for writing back the result into the register. It must be pointed out that these pipeline schemes require a two-port register file in order to simultaneously read the two source operands.

To solve the problem of disrupting the pipeline due to data and control hazards (i.e. data interdependencies and branches) other RISC processors use more advanced pipeline techniques such as data hazard detection and operand bypass [1, 3], as well as static and dynamic branch execution schemes (see section 2.3.5.4). However, to solve control hazards, the basic RISC approach relies mainly on the delayed-branch scheme, which is a software-oriented approach. Nearly all RISC architectures employ this method.

To further enhance performance, other RISC processors employ superscalar and superpipelined designs [31, 24, 43, 66]. In a superscalar design, a number of execution

units operate in parallel on different instructions. For example, the PowerPC™ 604 [84] can finish executing six instructions in parallel through one Floating Point Unit (FPU), one Branch Processing Unit (BPU), one Load/Store Unit (LSU), two Single-Cycle Integer Units (SCIUs), and one Multiple-Cycle Integer Unit (MCIU). This however requires more die area. The design complexity is also increased because of the complex multi-instruction dispatching logic and the need to design the register file with extra read and write ports to service all the execution units. In a superpipelined design, the number of pipelines is increased and all functional units are also pipelined, allowing a higher clock frequency. However, more hardware and complex control logic is required, and the number of cycles per instructions (CPI) is increased due to longer branch and load delays. For low-power designs, a shorter pipeline is better since it results in a lower clock frequency and thus lower power dissipation. The performance loss due to the lower clock speed is partly offset by the improved number of cycles per instruction due to shorter branch and load delays [2].

Fig 2.3 Four-Stage Pipeline Model

2.3.5.4 Branch Execution

In a conventional pipeline processor, a branch or jump to another area in a program generally cannot be done in only one machine cycle. Thus, these instructions invalidate the pipeline and force the controller to flush the pipe. Fetching instructions that are subsequently flushed from the pipeline before they can be executed can waste as much as 20% of the computer's memory bandwidth. Instead of flushing the next instruction from the pipeline, and thus wasting one cycle after every control transfer instruction, most pipelined RISC architectures employ a number of techniques to reduce the cost of branches. More information on some of the commonly used techniques can be found in [8, 33, 34, 35, 36, 54]. A few of these are summarized below:

1. **Delayed Branch:** In this mechanism, the branch instruction takes effect after the execution of the n ($n = 1, 2, 3, \dots$) instructions below it. The compiler is responsible for scheduling instructions in the n delay slots in order to minimize the execution time of the branch. Examples of processors employing this mechanism include the MIPS R4200 [2] and the MIPS R3000 [9]. As a variation of this technique, a processor can be provided with the possibility of nullifying the execution of the instructions in the delay slots, thereby increasing the number of delay slots that can be profitably used. This is called delayed branch with squashing, and is used in SPARC [10].
2. **Early Execution of Branches:** This reduces hazards caused by branches by executing some of its operations in advance. An example is the Motorola 68040 [11] which has an additional adder to compute the target address as soon as a branch instruction is fetched.
3. **Branch Prediction:** Branch instructions include bit(s) that are used by the compiler (static prediction) or the internal processor hardware (dynamic prediction) to predict the most likely result of the branch. Examples are the Intel 80960-Next Generation [12] and Hobbit [1]. Other processors, such as the Motorola 68060 [23], use branch caches or buffers to predict the direction of a branch based on past execution history.

4. **Multiple Prefetch:** This involves prefetching the instructions in either of the possible paths of the branch, thus making the target branch instruction available immediately after the result of the branch is known (regardless of the taken path). This is implemented in the Intel i486 [8, 13].
5. **Parallel Execution of Branches:** This overlaps the execution of branches with the execution of the rest of the instructions, as is the case in the IBM RS/6000 [8, 14].

Typically, advanced branch mechanisms, such as the COBRA [8], combine several of the above techniques to reduce the branch delay to zero. The PowerPC™ 604 [84] features dynamic branch prediction with speculative execution through two branches, a 64-entry fully-associative Branch Target Address Cache (BTAC), and a 52-entry Branch History Table (BHT) with four levels of prediction. The choice of the branch scheme to implement will depend on the specific processor architecture and design constraints.

2.3.6 Memory Constraints

Load and Store to external memory generally disrupt the pipeline if there is only one external bus. For this reason, RISC processors try to reduce accesses to external memory. However, at the same time, processor's memory bandwidth must be increased. Most RISC processors meet these requirements by implementing:

1. A large register set to reduce a program's need to store operands in external memory.
2. Large and high-speed cache memories for both data and program accesses.
3. Separate data and program buses for data accesses without disrupting the pipeline.

2.3.6.1 Overlapping Register Windows

In order to reduce memory accesses, and provide fast access to local and global variables during procedure calls, register windowing is used. The principal advantage of register windows over a fixed set of registers is that they reduce the number of load and store instructions required to execute a program, and also allow for fast trap handling. For

example, for large C programs, about 20% of the executed SPARC instructions are loads and stores, compared to 30% to 40% of executed instructions in RISC processors without register windows [10]. Figure 2.4 shows the overlapping register windows used by the RISC-I architecture [49, 52].

Fig 2.4 Register Windowing

There are 138 x 32-bit registers arranged in eight overlapping windows of 24 registers each. Each window is split so that six registers can be used for parameter passing during subroutine calls. A pointer is simply changed to select a group of six registers. To perform a simple call or return simply needs a change of pointer. Once again, the large number of registers is needed to minimize external memory accesses. This technique allows very fast procedure calls and is beneficial for real-time applications requiring fast responses. If the procedure calls require more than six variables, one register must be used to point to an array stored in external memory. This data must be loaded prior to any pro-

cessing and in this case the register windowing loses much of its performance. If all the overlapping windows are used, the system saves a window or all the complete register set to external memory. This, also, may negate any advantages that windowing initially gave.

2.3.6.2 Caches

On-chip cache memory is the basic solution to reduce processor-external memory traffic. Because this is one of the major goals of the RISC approach, cache memories have become an integral part of RISC-based computers. Cache memories are used to hold temporarily those portions of the contents of main memory which are currently in use. This information may be accessed in much less time than that located in main memory because caches use high-speed static RAM. Cache memory architectures have been successful because of the temporal and spatial locality properties. The former means that the information which will be in use in the near future is likely to be in use already (such as in program loops), while spatial locality means that portions of the address space which are in use generally consists of a fairly small number of individually contiguous segments of that address space. In other words, the next instruction references of the program are likely to be near the current ones.

Caches consists of a data buffer that holds instruction or data, a tag buffer which is a small memory that contains the address of the stored instruction or data, and the cache controller. On each memory access, the address generated by the CPU is compared to those stored in the tag buffers. If it is present, the data is fetched from the data buffers, otherwise it is fetched from the main memory. Then some data in the cache is removed to load in the ones fetched from the main memory; the removed data is copied back to memory if it was updated in the cache. Therefore, an optimized cache design maximizes the probability of finding a memory reference target in the cache (hit ratio), minimizes cache access time, the delay due to a miss (main memory access and cache update), and the overhead of updating main memory to maintain information consistency.

2.3.6.3 The Harvard Architecture

The Harvard architecture was first used in the Mark-I computer [43] by researchers at the University of Harvard. It solves the problem of disrupting the pipeline when executing data memory accesses by employing separate data and instruction buses. This increases the total processor bandwidth and allows for a fetch of one instruction every clock cycle, while accessing the main memory for load and store operations. Dual buses also make it possible to incorporate dual cache memories in the architectures: one for instructions and the other one for data. The major disadvantage of the Harvard architecture is the hardware cost due to increase in I/O pins. Some designs try to solve this by multiplexing address lines for instruction and data buses. This, however, defeats some of the benefits of the pure Harvard approach. Figure 2.5 compares these two approaches to the conventional von Neumann single-bus architecture.

Fig 2.5 Harvard and von Neumann Architectures

2.3.7 The KIM20: An Example RISC Microprocessor

The KIM20 [43] was designed for efficient execution of symbolic processing. The main goal was to design a reduced core of instructions that capture the fundamental primitives of symbolic processing, and to implement them directly into hardware. A pure RISC approach was followed, and the KIM20 architecture (Fig 2.6) was mainly based on the RISC-II data path.

The processor uses a 3-stage pipeline: Instruction fetch and Decode, Read operands and Compute, and Write back into register file. It has 138 registers organized as a circular buffer of overlapped register windows of 32 registers each. All instructions are 32-bit, and use a single format shown in Fig 2.7. Two 32-bit data formats are used to allow a homogeneous and compact coding of lists and trees, and a 32-bit numerical signed integer value. The KIM20 instruction set is shown in Table 2.2.

Fig 2.6 KIM20 Data Path

Instruction Rd, S1, [#]S2

Fig 2.7 KIM20 Single Instruction Format

ADD	integer addition	CALL	call function
SUB	integer subtract	RETURN	return from function
AND	logical AND	TRAP	software exception
OR	logical OR	RETE	return from exception
XOR	exclusive OR	JUMP	immediate jump
ROTATE	left rotate	BRANCH	indexed branching
SHIFTL	shift right logical	COND	condition test
SHIFTA	shift right arithmetic		
INSERT	byte insertion	FIRST	read the "first" field
EXTRACT	byte extraction	NEXT	read the "next" field
HASH	hash code processing	SETF	write the "first" field
HYPER	message routing	SETN	write the "next" field
CASE	case processing	WTAG	write the "tag" field
		NEW	cell allocation
		FREE	cell garbaging
PUSH	push onto LIFO stack		
POP	pop from LIFO stack	LOAD	read code memory
SEND	write the data memory	STORE	write code memory

Table 2.2 KIM20 Instruction Set

This chapter is not an exhaustive survey of all the design issues used in the design of the processor presented in this thesis. Most of the issues have been presented in brief summaries, and hence an attempt has been made to provide references for detailed or extra information on specific topics where this might be required. Otherwise, the design issues discussed in this chapter, together with the architectural features of both the initial and previous designs presented in the next chapter, form the basis of the design of the processor architecture discussed in Chapter 4.

Chapter 3

GENERAL ARCHITECTURAL DESIGN

3.1 INTRODUCTION

In this chapter, a complete description of the control application considered in this design is presented. The chapter also includes the specifications and design constraints of the integrated processor, as well as the ideal design for this specific control application. A summary of the previous design is also included in Section 3.4 to highlight the differences in the implementation strategy, architecture and performance with the current design (presented in Chapter 4). A detailed description of the previous design can be found in reference [41]. This reference also includes the author's description of the Bar Tester application, from which Section 3.2 is adapted. The material contained in this chapter therefore presents a comparative study of the differences from, and influence of the previous design, the ideal design, and the original processor specifications and design constraints on the design of the processor presented in this thesis.

3.2 APPLICATION CONSIDERED: BAR TESTER [41]

Due to part volume increase and limited testing resources, a new test instrument was developed by Michael Gazier [41] for the optoelectronics device group at Northern Telecom, and implemented into production by mid-1989. The purpose of this instrument is to characterize the light output power versus current of semiconductor 1.3 μm and 1.55 μm III-V InGaAsP telecommunication lasers. The peculiarity of this instrument is that instead of the usual test method consisting of full characterization of one laser chip at a time with a 0.1% duty-cycle to allow cooling of the device during testing, this system tests

10 lasers in parallel by interleaving the laser cooling time of each laser with test pulses to the other 9 lasers.

This test is possible as a result of several factors. First, the devices are presented in bar form rather than in chip form to the tester. Previously, each wafer of semiconductor lasers was scribed into bars of approximately 40 laser devices each. Each bar was then cleaved into 40 dice, or individual devices, which were then tested. A die measures about 10 by 15 thousandths of an inch and is difficult to manipulate. To test a die, it must be loaded into a test jig, the lasing facet aligned to the jig orientation, and the contact pad on the top surface aligned to two probes. One probe is used for current injection and the second for voltage sensing. Then, a pulsed current ramp is applied while the device's electrical and optical responses are captured. Generally, it is desirable to digitize the data and transfer it to a computer for analysis of various parameters such as resistance, lasing threshold current, and whether the device exhibits non-linearities referred to as kinks. Once analyzed, the pass devices must be sorted from the fail devices.

The so-called Bar Tester system simplifies several steps in this process. Manual labor is reduced by testing the lasers in bar form rather than in die form, as bars are much bigger and thus easier to handle than the individual dice. Bars are placed four at a time on a test jig allowing photodetectors to access both front facet and rear facet light emissions of the lasers. The jig contains a groove with a vacuum such that the bars are automatically centered and held in place. This jig is placed on a motorized stage which can move between the four areas of bar loading, picture acquisition, probing and inking. After the bars have been loaded into the jig, the stage moves the approximate edge of each bar under a picture acquisition area where a camera takes a snapshot and sends the image to the computer for determination of the probing area of the first laser in the bar. This process is repeated for each bar on the jig. With this information, the computer can instruct the stage to align the bars perfectly under the probing area. The probes are a set of 10 pairs of probes spaced to correspond to the lasers' contact areas' pitch. Of each set, one probe is

used to inject the current into the laser and one to sense the resulting voltage. This whole alignment process removes contact area alignment and individual device manipulation by the test system operator. In order to test them properly, the devices on the bar must be isolated from each other, either by the device structure or by a specific process step such as a trench isolation. Testing of semiconductor laser devices required a 0.1% duty cycle pulse train so as not to overheat the devices. Thus for a 5 microsecond pulse applied to the laser, the test system must wait 5 milliseconds before applying the next pulse. The Bar Tester probes 10 lasers simultaneously and thus can pulse the other 9 lasers during the cooling time required for the first laser. In fact, enough time remains after pulsing all 10 lasers before the next current pulse that the Bar Tester uses this time to send the acquired data to the main data storage computer. Thus, the normal system approach to testing one laser at a time is accelerated 10 times simply by interleaving device tests. Other accelerations ensue by uploading the data to the data storage computer simultaneously to test rather than after characterization and by uploading data packets to the data storage computer. This latter feature has the effect of engaging a DMA transfer mode on the main computer, thus accelerating data transfer even more. Finally, the system test has been sped up by accelerating the pulse timing from 10 microseconds in previous systems to 5 microseconds.

All these factors bring large speed improvements to device testing, the final one ensuing from the use of an inker. Although commonly used on silicon device testers, die inking is not used in laser device testers due to the small size of these devices and to the usual approach of individual and sequential device testing. Since the Bar Tester tests devices in bars of approximately 40 devices, inking has been incorporated as a final step before the devices are unloaded. Then, as devices are cleaved into individual die, they are easily sorted into pass and fail devices.

It is therefore evident that the Bar Tester application requires real time control of sequencers, ADCs, DACs' and analog electronics. Power control is required to switch vacuum and nitrogen solenoids and the inker for the system's mechanics. Communica-

tions hardware between the system and the data storage computer are also needed. It may be beneficial to have some local computation power to perform data manipulation such as data smoothing. It can be seen that “intensive” low-level control is needed to operate this system as defined.

Fig 3.1 Bar Tester System Architecture

Figure 3.1 outlines the implemented system architecture. The Database and Data Analysis Main Control Computer controls the entire system. System mechanics are labelled Physical System and Control and ElectroMechanical Physical System. The first includes the main stage with bar holding jig and prober, while the second includes relays for the inker, solenoids for vacuum and nitrogen control, and a camera for picture acquisition. The Bar Tester itself is composed of a digital system, Main Processor Boards, an

interface between the digital and analog systems, Auxiliary Analog Micro-Control Digital Board, and an analog section composed of Analog Boards.

The Main Processor Boards include two interfaces to the Database and Analysis Computer, namely a General Purpose Interface Board (GPIB) and a General Purpose Input Output (GPIO) interface board. These are standard instrumentation bus standards and are not discussed here. The processor used is a Motorola MC68000 operating at 10MHz with 8 KWords of static RAM and 8 KWords of EPROM and minor glue-logic. Circuitry to allow the processor to control I/O at high voltages and currents is interfaced with a parallel interface/timer and denoted by Parallel Power Control. The timer is used both for proper timing of the inker control and to ensure the 0.1% duty cycle for the pulse train. The picture acquiring camera uses outputs an encoded TTL level signal and is decoded by the Custom Camera Interface board. The Auxiliary Analog Micro-Control Digital Board includes some logic for interfacing the main microprocessor to the ADCs, DACs and multiplexers in the analog section as well as a microsequencer. The microsequencer is a state machine which generates precise timing signals for generating the 5 microsecond current pulse to the laser under test.

The pulse sequence is to halt the system processor by withholding the data transfer acknowledge DTACK signal, switching the pulse to the proper laser away from the dummy load, waiting for the response to settle, triggering the sample-and-hold circuitry, triggering the ADCs, switching the pulse back to the dummy load for transition minimization, and releasing the DTACK signal. The main processor is responsible for providing the 0.1% duty cycle timing for the pulse train and in fact uses its timer for this purpose. The Analog Boards are composed of six main sections as follows. A Current Source is controlled by a 16 bit number from the main processor. A sixteen channel Output Current Multiplexer switches this current between a dummy load, the ten probes to the device under test, or to various calibration loads. The Input Voltage Multiplexer switches the selected channel's voltage, including that from the ten laser voltage sensing probes, back

to the measurement electronics. The multiplexed channel is preprogrammed into the Analog Bus Address Decoder and controlled by the Microsequencer under main processor control. The light emitted from the laser under test is collected by a Front and Back Facet Detector. These are driven by a Front and Back Facet Detector Bias Circuitry with an output voltage proportional to the captured optical power. This voltage is connected to the measurement electronics which are composed by a Level Conditioner/Filter/Input Amplifier stage followed by a 4-Channel 16-bit Digitizer. The four channels are the current output to the device under test, the voltage sensed, and the front and back facet light detected. All devices such as timer, DACs, ADCs are memory mapped to the main processor, i.e. each device's control registers are "mapped" to a particular memory location.

It was proposed that the Main Processor Board without the EPROM and SRAM memories and the Auxiliary Analog Micro-Control Digital Board could be implemented as an integrated instrument controller chip. In addition, it was desirable to have a small on-chip SRAM for data storage. Performance was to be of the same order as the present described system, including the real-time sequencer. An open system allowing for future upgrades was also desirable. This would lower system cost and complexity of the Bar Tester by integrating as much of the digital circuitry as possible. The previous design, therefore, focused on the design of a CPU core for a low-cost embedded instrumentation processor such as this one, but did not dwell on design of the sequencer and other custom units as the architecture would easily be upgradable to accommodate such components.

3.3 IDEAL DESIGN

3.3.1 Architectural Description

The ideal processor architecture and performance for this instrumentation control application was defined at the beginning of the design [41]. A number of features were proposed to be included in the architecture. These features are briefly described below.

1. RISC Architecture.

Because of the small instruction set required for this application, the RISC architecture would reduce the overall design time as well as the chip area, thereby reducing the cost, relative to a CISC design. The single cycle operation and fixed instruction length would allow fast real-time control, with low and predictable latency for interrupt response. The real estate saved by using the RISC architecture would be used to implement on chip caches and custom I/O units such as the GPIB interface, the camera interface, a parallel interface and a timer. Register windowing would allow fast interrupt handling. In general a RISC architecture presents most of the advantages described in chapter 2. The fact that such an architecture would require a large external memory would not be much of a concern because of the low cost of memory. External memory would also allow for future upgrades using cheaper and bigger memory chips.

2. DPSRAM-Based Registers

The use of a dual port static RAM (DPSRAM) for register emulation instead of flip-flop-based registers would reduce chip real-estate. The area saved would be used to expand the DPSRAM in order to allow register windowing and extended on-chip data storage. Part of the DPSRAM could also be used as an instruction cache for the small number of data acquisition loops in the application code.

3. Fine-Grained Pipelining and Caches

Pipelined execution of instructions would increase the overall speed of the processor. This would mean overlapping data or instruction fetch with instruction decoding and execution or data storage. While a small instruction cache would allow storing of tight data acquisition loops on chip and thus speed up their execution, a data cache would not be beneficial since the data read from external I/O changes continuously, and is sent directly or with little processing to the analysis computer. Because of the low-duty cycle used in the testing, enough time remains to make these data transfers. However, a small data cache would still be used for the GPIB link, and also could be valuable if the processor is to be used for other related applications.

4. Custom I/O with "Smart" Modules

Separate and independent "smart" modules and I/O would off-load the processor from trivial operations and this would allow high-speed system operation even with a fairly low-performance low-cost processor. On-board smart modules for this system include the GPIB interface, the camera interface, a parallel interface and timer, and a hardware control unit with microsequencer. While most of these are specific to this application, and could therefore be implemented as discrete components if resources are limiting, the on-chip inclusion of a timer and a parallel interface would be especially advantageous since most control applications require the use of these components.

5. Non-Multiplexed Asynchronous Buses

A standard von Neumann architecture with asynchronous bus transfers and a non-multiplexed address/data bus offers minimal pin count and maximal flexibility. Two internal buses would allow simultaneous access to both operands of binary operations and these would easily be extended externally to allow the use of both buses as address and data buses. Even more ideal would be a system with separate external buses for both the

code and data. This would allow the processor to fetch a new instruction while performing an I/O operation of a previous instruction. This being an instrumentation control processor, where I/O operations are predominant, this feature would substantially increase the processor speed. But since this means an equally substantial increase in pin-count, this feature must be weighted against the overall chip cost.

6. Code Compatibility with MC68000

Code compatibility with the original MC68000 code would allow reuse of the already designed application code. This would reduce design time and present a major cost saving in implementing the system. Also, an instruction set with elaborate and efficient support of I/O operations would be ideal, not just for this application, but also for upgraded or other related control applications.

7. Other Features.

Another important consideration is to incorporate as much of the system glue logic on-chip as possible in order to reduce external components. Other desirable features are minimum pin-count, modular design for easy incorporation of LSSD test circuitry for each module, and an easily expandable architecture, both internally and externally. The processor should also be capable of performing general purpose computations. Advanced computational units such as a multiplier would not be necessary since another computer is used for analysis. Otherwise, much of the data processing can actually be done in the integer domain with ADD and shift operations only. In general, reduced chip area would be desirable.

3.3.2 Performance

The chip performance is desired to be at least of the same order as the previous discrete implementation. It must be recalled that the speed of the testing is determined by the width of the pulse train and the 0.1% duty cycle. Therefore applying a pulse width of 5 microseconds require 5 milliseconds for cooling before the next pulse train is applied. This means that substantial speed increases over the 10MHz used in the discrete implementation is not a major consideration. However, this would, once again, be a desirable feature if the processor is to be used later on the upgraded system or other faster control applications. The main design goals therefore remain to be low system cost, especially by reduced chip area, and maximum integration. Small on-chip data memory would also be desirable, but not a major concern since this application, like most other instrumentation control applications, does not require much internal data storage, since most of the processed data is continuously read from I/O devices.

3.4 PREVIOUS DESIGN

3.4.1 Architectural Overview and Design Constraints

Figure 3.2 is a block diagram of the processor that was previously designed to meet the specifications of the ideal design of section 3.3. The architecture consists mainly of four internal blocks. The Control Unit controls and synchronizes all operations on the chip. It consists of the microprogram controller (MPC), the flags and branch decision logic (BRA), and the instruction register (IR). The MPC generates control signals to sequence the whole system. The instruction register stores the instruction currently being processed, while the BRA preprocesses the instruction and recodes it into a 4-bit space for efficient and simpler microcode processing. The Dual Port Static RAM (DPSRAM) provides internal register emulation including the program counter (PC) and the stack pointer (SP). The

unassigned 174 memory words are left for use by application programs. Table 3.1 gives the memory map of the DPSRAM. A standard 16-bit Arithmetic and Logic Unit (ALU) provides simple Boolean and integer arithmetic operation processing. It also includes bit testing circuitry, as well as a single bit shifter. The ALU performs the basic logical operations of AND and OR, as well as addition/subtraction through a 16-bit carry-ripple adder. It also does the application-specific SWAP operation, in which the lower and upper bytes of an operand word are swapped.

Fig 3.2 Previous Design Processor Architecture

The System Bus Controller implements address decoding and handshaking with external and internal units, provides external bus interfacing, and arbitrates internal processing, even placing the processor on hold (generating wait states) when performing slow external I/O operations.

The design has three internal buses and three corresponding external buses. There are two internal data buses (BUSA and BUSB) to allow for simultaneous searching of ALU operands from the DPSRAM, and also to simultaneously generate data and address signals for external I/O operations. These buses are extended externally into the data and address buses respectively. The third bus is a control bus.

The processor was designed quasi-MC68000 in order to execute the original application code with minor modifications. This dictated the minimal number of internal registers, external buses, internal functionality and machine opcodes. The 16-bit bus width was chosen as a compromise between performance and chip area, especially since the analog hardware performs 16-bit digitizing.

Address	Functional Usage
0-7	Data Registers, D0-D7
8-15	Address Registers, A0-A7
16	Program Counter, PC
17	Stack Pointer, SP
18-23	Stack Registers (Programmable)
24-192	DPSRAM for Application Programs

Table 3.1 On-Chip DPSRAM Memory Map

3.4.2 Instruction Set and Execution

A set of 26 instructions was derived from the study of the original application code. Most of the instructions have a RISC-type register-to-register and memory-to-register structure. Two instructions are application specific for data acquisition environments. The complete instruction set is listed in Table 3.2. All instructions are one word long, except for long branches that have the offset in the second word.

Instructions execute in three phases: fetching the operands, processing the data and storing the result. This was supposed to take 2 to 3 clock cycles in the final microcode implementation. External memory accesses are two clock cycles, with wait states inserted when accessing slower devices. To enable proper read and write timing of the internal DPSRAM, a higher frequency external clock is subdivided into three phases. Three of these sub-clocks form a single clock cycle for the microcode sequence clock.

3.4.3 Performance

It was estimated that since all operations except external memory accesses execute within a minimal sequence of three single-phase clock cycles, then the performance of the previous processor was at least as good as the equivalent MC68000 performance at the same clock rate. The effective clock rate for the design was put at 16.7 MHz from the external clock of 25 MHz. It was concluded, therefore, that it is at least as efficient as the original Bar Tester System executing at a 10 MHz clock rate. It must however be mentioned that the actual microcode implemented was non-optimized, requiring much more than the estimated three clock cycles.

Glue logic integration is provided by incorporating address decoding and handshaking decoding on-chip. Similarly, on-chip memory is provided in the form of the DPSRAM. However, much of the integration capability is provided in the form of an open

architecture, allowing the addition of both internal and external units using the system bus controller-interface capabilities. None of the required internal custom units (Camera interface, GPIB interface, parallel I/O and timer) is added into the chip's structure. Interrupt capability and internal pipelining have not been implemented. The design is modular, allowing easy addition of LSSD type edge scanning for complete chip testing. The processor has a pin count of 51, with the overall chip area manually estimated to be 24 mm².

As mentioned in the previous chapters, the architecture of the current processor has been designed differently from the previous design in order to reduce the silicon real-estate and increase performance. However, details of the application (discrete-component design), the ideal design, and the previous design have been presented in this chapter to highlight the initial design specifications and constraints, as well as the influence that these have had on the current processor architectural design presented in Chapter 4.

Vector ID	Instruction	Description
0	MOVE.W Dsss, (Addd)	internal data move
1	MOVE.B (Asss), Dddd	internal data move
2	MOVE.W (Asss), Dddd	internal data move
3	MOVE.W #<X>, Dddd	external data move
4	MOVEA.W #<X>, Addd	external data move
5	BTST.B Dsss, (Addd)	bit testing
6	AND.W Dsss, Dddd	logical AND
7	OR.W Dsss, Dddd	logical OR
8	ADDA.W #<X>, Addd	external word addition
9	ADD.W Dsss, Dddd	Integer Addition
10	SUB.W Dsss, Dddd	Integer Subtraction
11	LSL.W #8, Dddd	logical shift left
12	LSR.W #8, Dddd	logical shift right
13	CMP.W Dsss, Dddd	compare
14	RTS	return from subroutine
15	JMP <X>	long jump
16	BRA <>>	short branch
17	BSR <X>	short branch to subroutine
18	BNE <>>	short branch if not equal
19	BEQ <>>	short branch if equal
20	BMI <X>	short branch if more than
21	BRA <X>	long branch
22	BSR <X>	long branch to subroutine
23	BNE <X>	long branch if not equal
24	BEQ <X>	long branch if equal
25	BMI <X>	long branch if more than

Table 3.2 Processor Instruction Set

Chapter 4

PROCESSOR ARCHITECTURAL DESIGN

4.1 INTRODUCTION

The original objective of the Bar Tester Core Processor design was to integrate the application's Main Processor Boards, the Auxiliary Analog Micro-Control Digital Board and other digital boards into a single chip. However, with the much reduced number of instructions required for the application, the need to replace the MC68000 as an overdesign also became necessary. Thus, the previous design [41] focussed on designing a processor to efficiently implement the instruction set required for this application, without implementing much of the integration. Only part of the original RAM (192 x 16-bit word) was designed as part of the microprocessor.

The aim of this design follows that of its predecessor. While the previous design used a semi-RISC microprogrammed control approach, the new processor is designed with a 4-stage pipelined full RISC architecture. This was to further reduce the real estate and improve the efficiency and speed [43, 50]. The design is based on the instruction set obtained by studying the original application code and the set implemented in the previous design. The 28 instructions are sufficient to implement the application code, with extra instructions added for flexibility, future expansions and generality. An attempt has been made, like in the design before this, to make the instructions very similar to the original MC68000's. This is done to reduce the task of program conversion.

The design of this 16-bit processor is based mainly on the RISC-I and RISC-II architectures, and the KIM20 microprocessor from the HECATE project. The processor is designed with a fixed 16-bit instruction format. It has been shown in [15] that reducing

instruction size for a RISC processor to sixteen bits increases code density with respect to a processor with a fixed 32-bit format. Therefore in cost-critical applications such as embedded control, where the cost of memory speed and capacity is a major concern, the smaller size of the 16-bit programs and higher performance with slow memory are obvious advantages [15]. In this case, the benefits of a denser 16-bit encoded program can easily exceed the path length (the total number of instructions in an execution trace) reduction achieved with a 32-bit format. Therefore the 16-bit instruction format exhibit significant cost-performance advantages over the 32-bit format under typical memory system performance constraints. A 4-stage pipeline has been used as opposed to the 3-stage pipeline of the RISC-II and KIM20 architectures. The shorter pipeline in the later designs reduces the number of cycles lost during pipeline flushing, yielding more efficient designs. However, more hardware is required for decoding and control.

No RAM is implemented on the chip. This is mainly due to the lack of RAM generators in the design tools. Only 16 registers, through which all instructions execute, are provided for data storage. A separate study of the layout design and overall performance will determine whether integrating the other blocks on the chip, and adding extra registers is cost effective.

The following pages give a detailed description of the design of the processor architecture. Firstly, the processor programming model is presented. This includes the instruction set architecture, the data formats, and the internal register model. This is then followed by the proposed hardware architecture. Specific hardware issues addressed include the control unit, the datapath (computational units, the register file, etc.), the external interface circuitry and the execution pipeline. The chapter concludes with a detailed description of the cycle-to-cycle execution schemes of the various processor instructions.

4.2 PROGRAMMING MODEL

4.2.1 Instruction Formats

Like most small RISC microprocessors [42], the instructions for this processor are designed for efficient pipelining and decoding. Except for MVI (Move Immediate Byte), which uses the immediate format, all other processor instructions are coded in a unique 16-bit word register format. This general format is only varied if certain bits or fields of the instruction word are not required by an instruction. In this case, the unused bits are either converted to condition bits to enhance the instruction, or just cleared. The use of predominantly only one instruction format has greatly reduced the size of and ease of designing the Instruction Pipeline and Decoding unit. The two instruction formats are shown in Fig 4.1.

Fig 4.1 Instruction Formats

Some of the fields of the register format are not required by other instructions. Table 4.1 outlines the modifications required for different instructions on this general format. On the other hand, all MVI instructions (MVI, MVIC, CLRW and CLRB) use the immediate format without any modification.

Instructions	Format Modifications
ADD, SUB, XOR, OR, AND	Exact (no modification)
LSL, LSR, SWAP, MOVEW	S1 is not required; bits (5:4) are used to select the shift function; bit (6) is not used (cleared)
CMP, BTST	Rd is not required; bits (10:8) are cleared. Bit (11) selects between BTST and CMP
BRANCHES, BSR	Rd is not required for branches; bits (10:8) are used for branch condition, and as Rd for BSR; bit (11) selects PC or S1 as base address
MOVEB, MOVEBC	S1 is not required; bits (5:4) are set to "11" for shift function; bit (6) is not used (cleared)
LOAD, LDI	S1 is not required; bit (6) selects LOAD or LDI; bits (5:4) selects are cleared
STORE	Rd is not required; bits (11:8) are cleared

Table 4.1 Instruction Format Modifications

To further reduce the size of the decoding unit, the instruction opcodes were selected in such a way as to use most of its bits as control signals. Table 4.2 gives the functions of the four opcode bits. Most of the other instructions executing through the ALU/Shifter have their opcode or condition bits set to conform with these bit settings.

A limited opcode space has been reserved for future expansion, and all reserved fields are set to zeroes. Since no traps have been implemented for reserved opcodes, executing unimplemented instructions will give unpredictable results.

Opcode Bits	Function
Bit (15)	Selects ALU or Shifter instructions
Bits (14:13)	Selects the ALU function: 00 - ADD/SUB 01 - XOR 10 - OR 11 - AND
Bit (12)	Selects ADD or SUB: 0 - ADD 1 - SUB

Table 4.2 Opcode Bits Functions

4.2.2 Data Format

Data used by the processor is either 16-bit or 8-bit long. All instructions and external memory addresses are 16-bit words. Only fixed point (integer) arithmetic is allowed on the processor. The data format for arithmetic operations is shown in Fig 4.2. Bit 15 is the sign bit, followed by 15 magnitude bits. Since the processor uses 2's complement representation, the sign and magnitude bits are treated alike in all computations. On the other hand, all the 8 bits of an 8-bit word are treated as magnitude bits.

Fig 4.2 Data Format

4.2.3 Internal Registers

The programming model is composed of 16 registers, 15 of which are general purpose registers, and register 15 is the Program Counter (PC). The PC is still visible to the second-level programmer, that is, it can still be read like all the other registers, and this fact can be taken advantage of by the programmer. It can not however be normally written to, except for a jump or branch operation. Any of the 15 general-purpose registers can be the destination register (Rd) or the operand 2 source register (S2), but only registers 0 to 7 can be the source of operand 1 (S1). Fig 4.2 shows the register model and the ALU flags. Z is the zero flag, N is the negative flag and C is the carry flag. None of the ALU flags can be read or written to.

Fig 4.3 Register Programming Model

In general, most microprocessors and microcontroller units, such as the M68HC11, support a stack which can be used by assembly language programmers for saving subroutine return addresses and register contents during interrupt processing, as well as for the storage of local or temporary variable values and subroutine parameter passing [51]. In this design, the stack is not implemented and hence subroutine instruction,

BSR, specify the register where the return address is to be saved and fetched, respectively. If required, the other two above described uses of the stack can be implemented by extending the register file to allow for register windowing [43, 52].

4.2.4 The Instruction Set

The processor is designed with 28 instructions, encoded in the two instruction formats. The syntax for the MVI (Move Immediate Byte) instructions is:

INSTR Rd, Imm.B

All other instructions use the syntax:

INSTR Rd, S1, [#]S2

where

- INSTR represents the mnemonic of the instruction. This is characterized by the opcode bits (bits 15 to 12) of the instruction word. In cases where instructions share one opcode, they are further differentiated by condition bits.
- Rd specifies the destination register among the 16 registers (bits 11 to 8 of the instruction word).
- Imm.B specifies an immediate 8-bit value (bits 7 to 0 of the instruction word).
- S1 specifies operand 1 source register among the first 8 registers (registers 0 to 7). The S1 field occupies bits 6 to 4 of the instruction word.
- # is an option which specifies the type of operand S2. When # (bit 7) is set to 0, operand S2 specifies the second source register. If it is set to 1, then S2 is an immediate 4-bit value.
- S2 specifies an immediate 4-bit value if # is set to 1, otherwise it specifies a second source register among the 16 registers. S2 field occupies bits 3 to 0 of the instruction word.

Unused fields or bits of the instruction word are cleared.

The processor's 28 instructions can be grouped in the four categories, depending on their functions and execution schemes:

1. **Arithmetic and Logical Operations:** All instructions in this category execute in one cycle through the Register Arithmetic Logic Unit (RALU). They include ADD, SUB, XOR, OR, AND and CMP, which execute through the ALU; LSL, LSR, and SWAP, executing through the Shifter, and BTST which uses the Bit Testing Circuit of the RALU.
2. **Internal Data Movements:** These are also RALU instructions. They use the Shifter "transfer through" function and thus execute in one cycle in exactly the same way as the other Shifter instructions. Data movement instructions include MOVEW, MOVEB, MOVEBC, MVI, MVIC, CLRW and CLRB.
3. **Flow Control Operations:** These are unconditional and conditional branches: BRA, JUMP, BEQ (BZ), BNE (BNZ), BOC, BGE, BLT and the subroutine instruction, BSR. All control instructions execute in 3 cycles. Branch instructions do not use the fourth cycle, while BSR use this cycle to save the return-to address in the register file. The JUMP instruction loads the PC with the content of S1. It is used, among other things, to return from procedure calls.
4. **External Memory Access:** These include LOAD, LDI and STORE. LDI execute in one cycle, its operand being the instruction word following the LDI instruction. Because of a provision of a separate bus for code memory, LOAD and STORE also execute in one machine cycle.

Table 4.3 gives a summary of the instruction set, outlining the instruction mnemonic, the operands required, the action performed by the instruction, and the number of cycles it takes to execute. Details of each instruction are included in Appendix A1. This includes the instruction syntax, format, description, operation and execution scheme. The effect on the flags registers of each instruction is also included.

Instruction	Operands	Operation	Cycles
ADD	Rd, S1, [#]S2	$Rd = S1 + S2$	1
SUB	Rd, S1, [#]S2	$Rd = S1 - S2$	1
XOR	Rd, S1, [#]S2	$Rd = S1 \text{ xor } S2$	1
OR	Rd, S1, [#]S2	$Rd = S1 \text{ or } S2$	1
AND	Rd, S1, [#]S2	$Rd = S1 \text{ and } S2$	1
CMP	S1, [#]S2	$S1 - S2$	1
BTST	S1, #S2	Test if bit number #S2 of S1 is set	1
LSL	Rd, S2	$Rd = S2$ shifted left 1 bit; $Rd(0) = 0$; $C = S2(15)$	1
LSR	Rd, S2	$Rd = S2$ shifted right 1 bit; $Rd(15) = 0$; $C = S2(0)$	1
SWAP	Rd, S2	$Rd(15:8) = S2(7:0)$; $Rd(7:0) = S2(15:8)$	1
MOVEW	Rd, S2	$Rd = S2$	1
MOVEB	Rd, S2	$Rd =$ lower byte of S2; Rd high byte left unchanged	1
MOVEBC	Rd, S2	$Rd =$ lower byte of S2; Rd high byte cleared	1
MVI	Rd, B	$Rd = B$; Rd high byte left unchanged	1
MVIC	Rd, B	$Rd = B$; Rd high byte cleared	1
CLRW	Rd	$Rd = \text{'0000'}$ H	1
CLRB	Rd	$Rd = \text{'00'}$ H; Rd high byte cleared	1
BRA	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$	3
JUMP	S1	$PC = S1$	3
BEQ	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$, if $Z = 1$	3
BNE	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$, if $Z = 0$	3
BOC	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$, if $C = 1$	3
BGE	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$, if $N = 0$	3
BLT	S1, [#]S2	$PC = S1 + S2$ or $PC = PC + S2$, if $N = 1$	3
BSR	Rd, S1, [#]S2	$Rd = PC + 1$; $PC = S1 + S2$ or $PC = PC + S2$	3
LOAD	Rd, S2	$Rd = \text{mem}(S2)$	1
LDI	Rd	$Rd =$ word at following the LDI instruction	1
STORE	S1, S2	$\text{mem}(S2) = S1$	1

Table 4.3 Instruction Set Summary

4.3 PROPOSED HARDWARE ARCHITECTURE

4.3.1. The Data Path

The Processor employs a regular data path inherited from the original RISC-II. Instruction fetching, decoding, timing and execution is controlled by the Control Unit. Almost all instructions follow the same data path during their executions. Fig. 4.4 is a simplified representation of the processor data path.

Fig 4.4 The Data Path

First the instruction is fetched from external memory and loaded into the instruction pipeline register. Then during the second clock period, the instruction operands are read. The source of these operands can be the register file, the feedback path, the first pipeline register, the Program Counter Unit (Branch PC value) or an immediate value (constant). After being read, operands are loaded into the RALU input registers. In the third stage, the computation in the RALU takes place, and the result of the operation is loaded into the RALU output register. The final and fourth stage involves writing the contents of the RALU output register into the register file.

The data paths for some instructions exhibit slight modifications to the general data path. At the end of the third stage of Branch instructions, the contents of the RALU output register are written into the PC. For external LOAD, the data loaded into the RALU output register in the third stage come from the external architecture through the Data Interface Unit. The contents of the RALU input registers are routed to the external architecture during the third stage of the STORE instruction.

For the internal architecture, it therefore becomes apparent that the access time of the register file and the time needed to perform a 16-bit signed addition/subtraction through the ALU are the two critical times for determining the cycle-time. Of course the external memory access time is very critical, but if this is not a limiting factor, the register file and the ALU are the critical paths in the general data path.

4.3.2 The Pipeline

The processor executes instructions in a standard 4-stage synchronous pipeline. This is used by a number of designs such as the Capitol Chip [47]. The pipeline was chosen over the 3-stage pipeline used by RISC-II and KIM20 microprocessors because of the later hardware overhead in decoding and timing. For a low-cost, low-power processor, a single pipeline represents the most efficient trade-off between power, die area and perfor-

mance compared with either a superpipelined or superscalar design [2]. Fig 4.5 shows the 4-stage pipeline scheme. The elements of the pipeline stages are:

1. The addressing (**fetch**) stage, where the Program Counter contents are placed on the external address bus to fetch the next instruction to be executed. The instruction thus obtained is loaded in the first instruction pipeline register. The PC value is then incremented, ready for the next instruction fetch.
2. The source (**read**) stage, in which the operands of the instructions are read and loaded in the RALU input registers. The sources of the operands are: the register file, the feed back from the RALU output register, the first pipeline register, the Program Counter or the Immediate Value. The source stage is controlled by the second instruction pipeline register (IR2).
3. The target (**execute**) stage is the third pipeline stage and is controlled by IR3. Here, the operands are operated on in the RALU and the result loaded into the RALU output register. Instructions requiring external operations (**LOAD** and **STORE**) perform these accesses during this stage.
4. The **write** stage uses the fourth instruction pipeline register to write the contents of the RALU output register into the register file. Instructions with nothing to write to the register file do nothing in this stage.

Fig 4.5 The Instruction Pipeline

All instructions, except branches, follow this pipeline scheme, thus executing in one machine cycle. Branches, on the other hand, 'flush' the pipeline in order to compute the branch-to address and load the PC. Thus, the two instructions following branch instructions are always cancelled, disrupting the pipeline and wasting two cycles. Because of this, three machine cycles are required to execute a valid branch.

In this architecture, the branch-to address is computed using the RALU. If an extra adder is provided to calculate the branch-to address, only once cycle of the pipeline would be wasted by branches. This would be enhanced by employing branch execution schemes described in chapter 2. These two approaches have however not been adopted in this design because of the extra hardware overhead. On the other hand, a dedicated PC incrementer is employed to ensure that the fetch stage of the pipeline can occur concurrently with the execution (RALU computation) stage. Finally, the MEM stage, which is usually placed between the EXECUTE stage and the WRITE stage, and is used to access the data cache [5, 6, 27, 42, 54], is not required in this pipeline scheme because no cache is implemented. To reduce timing and design complexity, operations in each pipeline stage are between flip-flops and data is latched on the negative edge of the clock [24].

4.3.3 The Control Unit

The general execution scheme which applies to most instructions, plus the simplicity and uniformity of the two instruction formats has made instruction decoding and execution control extremely easy. Actually, only those instructions deviating from the general execution format need be decoded. Also because of the small instruction set, most of the bits of the instruction word, even opcode bits, are directly used for control. Therefore, hardwired control is used because for control complexity of this order (less than 50 Moore-type equivalent states), it yields better speed and area characteristics than conventional PLA's or ROM microprogramming [71, 84].

The control unit, shown simplified in Fig 4.6, comprises the decoding and timing unit, the pipeline registers, the no operation (NOP) generator and the branch decision circuit. Both pipeline stages 1 and 2 have 16 registers each, stage 3 has 11 registers because # (bit 7) and S2 (bits 3:0) are not required in this stage. The last stage only needs Rd and hence has only 4 registers. Since the instruction execution is uniform for most instructions, the timing unit generates almost the same timing signals sequence in each stage. These signals include among others those for loading the PC, writing into the register file, loading the RALU output register, loading RALU input registers, and loading the instruction and NOP pipeline registers. An operation in a pipeline stage is cancelled when the NOP generator places a NOP ('1') in that stage NOP register. Since all the four stages are edge-triggered, the timing unit simply nullifies the edge if a NOP exists. The Branch condition circuit checks if a branch instruction in stage 2 is valid or not. Its output is used by the NOP generator to cancel certain pipeline stages.

Most control signals are generated by the Decoding Unit, the Timing Unit, the Branch Decision Circuit and the NOP Generator. Tables 4.3 and 4.4 are a complete list of the main control signals. Some signals require no decoding because they are derived directly from the instruction word bits. These are shown in Table 4.3. The decoding unit also performs address comparisons between S1 and Rd, and between S2 and Rd. This determines whether operands S1 and/or S2 should be routed from the RALU output register as opposed to the RALU input registers. This happens when an instruction uses as a source register the destination register of the previous instruction. When the operands are fetched in stage 2, the previous instruction result has not yet been written into the register file, but is in the RALU output register. It is therefore necessary to read this result from the output register rather than from the register file. Most of the other control signals generated by the Decoding Unit are used within the Control Unit by the other three blocks.

Fig 4.6 The Control Unit

Signal	Function	IR bits
ALUS	Selects ALU or Shifter	15
ALUFN	Selects ALU function: ADD/SUB, XOR, OR, AND	14:13
SHTFN	Selects Shifter function: LSL, LSR, SWAP, Transfer	5:4
ADDS	Selects ADD or SUB	12
ADDRI	Rd destination register address	11:8
ADDR1	S1 source register address	6:4
ADDR2	S2 source register address	3:0
CONSB	Immediate Byte	7:0

Table 4.4 Undecoded Control Signals

Signal	Function	Signal	Function
LDPC	Loads the Program Counter	ALE	Address Latch Enable
LDOR	Loads RALU output register	LDFG	Loads flags register
LDSX	Loads RALU Input registers	LDBR	Loads Branch PC register
LDIR	Loads Instruction registers	LDNP	Loads NOP register
WRL	Writes to register low byte	WRH	Writes to Rd high byte
IOS	External I/O Access Select	RW	External Read/Write
BTST	Bit Test Select	MVI	Move Immediate Byte
BR2VAL	Valid branch	BR2INV	Invalid branch
BSR4	Branch to Subroutine	NOP(n)	Stage (n) No Operation
LDI2	Load Immediate	LD3	External Load
BR2	Branch (Stage 2)	BR3	Branch (Stage 3)
LDFGEN	Enable Load Flags Registers	WRTHEN	Enable Write to High Byte
OREGS	Selects input to RALU output register: 0 = RALU 1 = External Data	RSBRS	Selects the external input to the PC: 0 = reset address 1 = Branch-to address
PCINS	Select input to PC 0 = PC incrementer 1 = External (RSBRS)	ZEROH	Selects immediate value 0 = Byte 1 = Nibble
ASEL	Selects source of operand S1: 00 = Register file 01 = Unused 10 = Output register 11 = Branch PC value	BSEL	Selects source of operand S2: 00 = Register file 01 = Immediate value 10 = Output register 11 = Instruction register

Table 4.5 Decoded Control signals

and performing a “transfer through” function which simply routes the input to the output. The last shifter function is employed by all internal MOVE instructions, making these instructions exactly like all the other computation (RALU) instructions. The bit shifted out in LSL and LSR is loaded into the carry flag register. All ALU and shifter functions modify the zero and negative flags, depending on the result, but the carry flag is only set by ADD/SUB and LSL/LSR. All other RALU operations clear the carry. The Bit testing circuit implements the application specific BTST instruction. It tests the status of the specified bit of the input word. If the bit is not set, the zero flag is set, otherwise it is cleared. The Bit Testing Circuit also sets the negative flag if the word tested is negative. The carry flag is always cleared. The result, zero and carry multiplexers select between the outputs of the various RALU blocks, while the zero checking circuit sets the zero flag if the RALU result is zero. The negative flag is always loaded from bit 15 of the result.

The RALU operates only on operands stored in its input registers and its output is loaded into its output register. Therefore, RALU operands must be loaded into its input registers during the read stage and its results loaded into the output and flags registers during the execution stage. Like some low-cost embedded control processors, such as the PA-RISC Controller [5], a floating point unit is not implemented to keep the chip cost low.

4.3.5 The Register File

The register file is a 16 x 16-bit wide array. It has two output busses and one input bus, allowing simultaneous accesses to three registers: one for writing and two for reading. The addresses of the three registers are decoded by three decoders. One of the output addresses is only 3-bit, and can therefore access only registers 0 to 7. The other two can access all the 16 registers. Register 0 to 14 are general purpose, and register 15 is the Program Counter (PC). However, all the 16 registers are treated the same: can both be read, even though writing to the PC requires executing a Branch/BSR instruction.

4.3.6 Program Counter Unit (PCU)

The PCU consists of a 16-bit incrementer and 2 16-bit registers for storing the two previous PC values for branch and subroutine instructions. The incrementer is used for incrementing the PC value after every instruction fetch.

The PCU also selects the source of the next PC value. This can be either the normally incremented PC value, the RALU computed value, which is currently in the RALU output register (for Branch and subroutine instructions), or the address 0000 (Hexadecimal), which is the address to jump to when a hardware RESET is initiated.

Since the computation of the branch-to address is done in stage 3 of the pipeline, it is necessary to save the address of the instruction following the branch instruction (i.e the PC +1 value when the branch instruction is in stage 1). This value is used in computing the branch-to address if the PC is selected as the branch base address. In this case the branch-to address is equal to this value plus the offset. The other PC value is also the PC value saved in the destination register Rd (for a return-to address) during a BSR instruction.

4.3.7 External Interface Unit

A large portion of the system cost and performance in embedded control applications is determined by the memory subsystem. Therefore, it is important to have a flexible memory interface for embedded controllers [44, 48]. In this design, the External Interface Unit is comprised of the Data Interface Unit (DIU) and the Code Interface Unit (CIU). The DIU implements a simple I/O interface of the processor to the external data memory and I/O. Fig. 4.8 shows a simplified diagram of the DIU. The chip select decoder does the on-chip select decoding of the two most significant bits of the address. Thus, four address spaces are internally decoded. If required, extra decoding can be done externally. The CIU interfaces to the code memory. Separate memory spaces are provided for code and data. These are accessed via separate 16-bit multiplexed buses, DATA and CODE, respectively.

The external data access handshaking algorithm is initiated by the processor driv-

ing the I/O Access Select signal (IOS) high. It then places the address of the external device on the internal address bus, ADDR, and drives read/write (RW) signal. This takes place when the Address Latch Enable signal (ALE) goes high. The DIU buffers the address onto the external DATA bus, decodes the address, and drives the device chip select, CS(n), low. The addressed memory or I/O device must latch the address at the falling edge of ALE. For LOAD, the processor then waits for one of the external devices to assert its Data Acknowledge signal, DTACK(n). When this happens, the DIU buffers the read data onto the internal data bus and negates IOS and ALE. An external write is executed the same way, with the processor placing data on the internal data bus when ALE goes low.

A wait state generator is provided to cater for slow devices and to reduce internal timing constraints. The interface unit also drives both buses to a high-impedance state when no external device is selected, allowing use of the external buses by other processors.

Fig 4.8 Data Interface Unit (DIU)

4.4 INSTRUCTION EXECUTION

4.4.1 Instruction Cycles

All instructions execute in a 4-stage pipeline. The first cycle of all instructions is the same. It is the instruction fetch stage, in which the address of the instruction to be fetched is applied on the external code address bus from the PC. The instruction read is then loaded into the first pipeline register of the Control Unit. Simultaneously, the value of the PC is incremented by the PC incrementer and loaded back into the PC to point at the address of the next instruction to be fetched. Fig 4.9 shows this cycle.

Fig 4.9 Instruction Cycle #1

The next 3 cycles vary for different instruction groups. Also, some instructions do not use the fourth instruction cycle since they do not require any data to be written into the register file. Instruction cycles 2 to 4 are described in the following sections.

4.4.2 Compute and Internal Data Movements Instructions

The second cycle of these instructions involve reading the operands to be computed on and placing them in the ALU input registers. There are six possible read cases:

- (i) Both operands read from the register file (Fig 4.10(a)).
- (ii) S1 read from the register file, and S2 is an immediate value (Fig 4.10(b)).
- (iii) S1 read from the register file and S2 caught from the feedback bus (Fig 4.10(c)).
- (iv) S1 received from the feedback bus and S2 read from the register file (Fig 4.10(d)).
- (v) S1 received from the feedback bus and S2 is an immediate value (Fig 4.10(e)).
- (vi) Both Operands read from the feedback bus (Fig 4.10(f)).

Fig 4.10 Compute instruction Cycle #2

During the third cycle, the RALU computation is performed on the operands in the input registers, and the result is loaded into the RALU output register. Fig 4.11 illustrates this cycle.

Fig 4.11 Compute Instruction cycle #3

The fourth and last cycle of compute and move instructions is shown in Fig 4.12. In this cycle, the result of the computation in cycle #3 is written into the register file. For CMP and BTST instructions, the fourth cycle is not used since no result needs to be written into the register file.

Fig 4.12 Compute Instruction Cycle #4

4.4.3 Branch and Subroutine Instructions

The second cycle of branch instructions involves reading the branch base address (operand S1) and the branch offset (operand S2) and loading them into the RALU input registers. The sources of these operands include all the sources valid for compute instructions plus a third source for operand 1: the Branch PC value in the PC Unit. This is the address of the instruction following the branch instruction and can be selected as the base address. Therefore cycle number 2 of branch instructions is the same in execution as that of compute instructions (Fig 4.10).

The third cycle is the computation of the branch-to address. This is done by adding the two operands in the RALU input registers and placing the result in its output register. Once again, this is exactly the same as that for compute instructions (Fig 4.11). The end of this cycle is also used to write the contents of the RALU output register into the PC register. Cycle #4 is not used by branch instructions, but is used by BSR to write the return-to address into the register file. Since this address is loaded into the RALU output register at the beginning of stage 4, writing it to the register file is executed in the same way as shown in Fig 4.12 for compute instructions.

4.4.4 External Memory (I/O) Instructions

The target operand of the LDI instruction is the word following the LDI instruction word in the program memory. This word is loaded in the first pipeline instruction register (IR1) just like any other instruction word. LDI second cycle reads this word from IR1. Thereafter, LDI execution is exactly the same as with a typical internal move.

The second cycle of LOAD and STORE instructions involves reading the external memory address and, for STORE, the word to be stored from the register file, and loading them in the RALU input register. This is a typical second cycle shown in Fig 4.10. Both of these instructions then do the external memory access in the third cycle. For STORE, the

contents of RALU input register A are written to an external address pointed to by the contents of RALU input register B. For LOAD, the word at the address pointed to by register B are read and loaded into the RALU output register. Fig 4.15 illustrates this cycle. The fourth cycle of LOAD writes the contents of the RALU output register into the register file. The STORE instruction does not use the fourth cycle since nothing needs to be written into the register file.

Fig 4.13 STORE and LOAD Instruction Cycle #3

4.4.5 The RESET Sequence

The RESET pin is used to start the processor by placing the address 0000H in the PC. Thus, programmers must place their first instruction at this location. If programmers wish to place their programs elsewhere in the external architecture, they must place a GO TO instruction (JUMP or unconditional branch, BRA) at the address 0000H. The RESET sequence does not clear any of the 15 general purpose registers. Therefore, if required, the registers must be cleared with an initialization program.

In this chapter, the detailed processor architectural specifications have been presented. This was the first design step, and was then followed by the VHDL behavioral and structural descriptions of the various architectural components and the processor as a whole. This VHDL design is described in the next chapter, together with all the other logic design, synthesis and verification processes.

Chapter 5

LOGIC DESIGN AND VERIFICATION

5.1 DESIGN PROCESS AND CAD TOOLS

5.1.1 Design Flow

Although a full-custom design approach results in higher performance and smaller die area [42], design automation and flexible design strategies are usually used in the design of processors to reduce the design time and to provide a processor core which is highly adaptable to different applications [5, 6, 55]. Therefore, the design flow format used for this processor is that of a typical automated synthesis design. The detailed steps and tools are however specific to Synopsys and Cadence, the two CAD tools used in this case for logic design and layout design respectively. A top-down design methodology has been followed. This is particularly useful for rapid exploration techniques during the early design-planning phases. Generally, to allow for optimization during the final layout phases, design environments usually employ a mixture of the top-down and bottom-up methodologies [7]. However, the bottom-up methodology requires the use of predesigned custom designs, and hence has not been employed in this design in order to reduce the design time.

Fig 5.1 shows the details of the design flow. The first step, which is the determination of the architecture from the processor specifications, is described in detail in Chapter 4. The other steps, plus the tools used, are described in the following sections. The rest of the chapter details the actual logic design and verification of the different architectural components presented in Chapter 4.

Fig 5.1 Design Flow

5.1.2 VHDL Behavioral Description and Simulation

VHDL has the advantage as a specification and synthesis language in that it can describe hardware at various levels of abstraction, from the architectural (behavioral) level down to the structural level [7, 58]. It also provides a generic design entry platform in that the design may be created in VHDL and the target implementation technology chosen later [59]. In this case, the synthesis tool can be used to map the design to any specific target technology, thus forgoing the task of a complete logic redesign should the initial intended target technology become obsolete or unavailable. Lastly, VHDL synthesis tools allow designers to follow a true top-down design methodology and also allow a quick comparison of trade-offs in implementing the same design in different technologies or styles [60]. In general, high-level synthesis allows designers to describe a design in a purely behavioral form, devoid of implementation details, and then synthesize the design with CAD tools. This achieves higher productivity gains since the design process is moved to higher abstraction levels, where designers can specify, model, verify, synthesize, simulate and debug designs in less time [7]. It is for these reasons, plus the general advantages of automated synthesis, that VHDL synthesis tools were employed in the design of this processor. The choice of VHDL over Verilog as the hardware description language for this design was justified by the author's level of knowledge of the two languages, and by the fact that VHDL is also an IEEE standard and is becoming more popular among designers and hence has extensive commercial support [7].

VHDL behavioral description was the initial design step in which a detailed specification of the processor was developed by several iterative steps of partitioning and refinements. The verification of each of the such steps is one of the most time consuming and critical phases of the automated design flow [55]. Each processor component was designed, optimized, simulated and synthesized with several iterations until an acceptable level of each of its essential characteristics (area, critical path delay, regularity, etc.) was achieved. In such designs, variations in VHDL coding styles have significant effect on the

logic synthesis optimization process. This is known as the problem of syntactic variance [7]. Generally, a structured, object-oriented style, that separates control and operations, is recommended for the writing of behavioral descriptions for synthesis of area-efficient hardware [62]. Also, since VHDL's semantics are primarily designed for simulation, high-level synthesis with VHDL is fairly difficult since most synthesis tools do not support all the VHDL constructs and attributes [7]. It is therefore important that the designer have very good knowledge of, or experience with the VHDL language, as well as the translation, optimization and technology-mapping techniques used by the specific synthesis tool. In order to predict the results of the synthesis process, and hence effectively guide the tool in synthesizing more efficient and simpler hardware, one also needs to have a fairly good knowledge of conventional hardware design techniques. All these factors have been extensively and effectively applied in this design.

After the behavior of each architectural component had been described in VHDL, and simulated to verify its functionality, the components were connected up, using structural VHDL, and then the whole processor was simulated to verify its functionality. Therefore, problems relating to the functionality and interconnection of the processor functional blocks were detected and corrected at this stage. This structural design specification was introduced at this early design stage, as opposed to using pure behavioral description, because the synthesis task becomes much easier and more efficient as the amount of structural detail increases [7]. The design phase of behavioral description and simulation is a highly iterative and hence time-consuming process, and the design is moved to the logic synthesis phase only when all the errors, relating to the functionality of the individual architectural components and the whole structurally-interconnected design, are corrected. The CAD tools that were used for behavioral description and simulation are the Synopsys VHDL Compiler [68] and the Synopsys VSS Expert Simulator [74, 75, 76], respectively. Since most stages in the design cycle are automated, the success and efficiency of the design depend very much on this stage.

5.1.3 Logic Synthesis

High-level synthesis consists of converting the abstract VHDL behavioral descriptions of the processor into a register-transfer level (RTL) design implementing that behavior. Each component in the synthesized RTL netlist is then designed into the target implementation technology by means of logic synthesis and technology mapping tools. In this design, the VHDL behavioral descriptions were synthesized and mapped into the CMOS4S standard cell technology using the Synopsys DC Expert Design Compiler [69, 71] and the compiled CMOS4S technology and symbol libraries. The CMOS4S technology library, which is required for logic synthesis and mapping, was created from the Verilog models of the standard cells, while the symbol library, which is used in the generation of the schematic from the synthesized netlist, was imported through the EDIF interface from their Cadence Composer Schematics. Both libraries were compiled into Synopsys format using the Synopsys Library Compiler [78]. Area, timing and other constraints are set during this stage. The schematics are viewed using the Synopsys Design Analyzer [70], which also provides graphical interface to other Synopsys tools including the Design Compiler, the VHDL compiler, the Test Compiler and the Design time. By iterating between the logic synthesis and the behavioral description phases, one can synthesize logic with minimum hardware. This is important to minimize the average number of transistors in order to reduce internal switching and hence dissipated power [22].

5.1.4 Timing Analysis and Gate-Level Simulation

Various timing properties of the schematic were analyzed using Synopsys Design Time [72], interfaced through the Design Analyzer. These include maximum and minimum delay paths, rise and fall times and critical paths. The schematic, together with the two CMOS4S libraries were then imported to Synopsys Graphical Environment (SGE) [77]. In SGE, the gate-level VHDL description of the schematic was automatically generated and then simulated using the VSS Expert simulator. The gate-level simulations are

particularly important since they also give the timing information of the design. These timing parameters are derived from the delays of the imported gates, plus sophisticated predictions of the final place and route stage. The technology file was also compiled again into the SGE format using the Library Compiler, and the gate-level VHDL simulation libraries were derived from this. The design flow then iterated between gate-level simulation, timing analysis, behavioral description and logic synthesis phases until satisfactory functionality, timing and general electrical rules were met. This concluded the logic design process.

5.1.5 EDIF Interface

The EDIF (Electronic Design Interchange Format) Interface [73] was used to import all the graphical information of the CMOS4S symbol library from the Cadence Composer schematics to Synopsys, and also to import the final processor schematics from Synopsys (after the logic design phase) to Cadence for the layout design. Because of some small differences in the EDIF supported by Synopsys and Cadence, EDIF files had to be preprocessed before transferring them across the two CAD tools.

5.1.6 Layout Design

The layout was produced using the schematics imported through EDIF from Synopsys Design Analyzer into Cadence Composer schematic viewer [80, 81]. Preview Cell Ensemble[82] was used to floorplan the design and place and route it. The layout is created by using 1.2 μm CMOS4S standard cell technology. The static standard cell technology and the design environments were chosen in order to achieve a fast and compact design while retaining the quick design cycle associated with the ASIC design methodology [37, 44, 45, 53]. In a low-cost and low-power processor like the one presented in this thesis, the use of fully static logic helps to simplify the design and avoid the higher power dissipation of dynamic logic due to precharge requirements [22, 25]. A static design is

also a major advantage if dynamic power management is to be implemented since a dynamic design would require saving the state of the chip if clocks are disabled during normal operation to save power [22]. Thus designing with mostly static circuits enables the processor to enter an extremely low power standby mode by simply freezing the clock [1, 25] or by reducing the clock frequency to a very small fraction of the peak rate using software control [6, 22, 23, 25, 26]. In general, full static CMOS is employed in many processor designs because of its stability, reliability, scalability and cost [24].

5.1.7 CMC CMOS4S Standard Cell Library [79]

CMOS4S is a 1.2 micron CMOS technology provided by Northern Telecom Limited and accessed through Canadian Microelectronic Corporation (CMC). This technology features twin wells, two metal layers, two polysilicon layers and an optional EPROM layer. The cell library consists of a basic set of static logic functions including basic logic gates, latches, flip-flops, multiplexors and several cells for implementing level-sensitive scan design (LSSD) techniques. In addition to the standard cells, the library contains three families of I/O pads, each containing input, output, bidirectional and analog pads operating at both CMOS and TTL levels. All of the I/O pads contain circuitry to protect against electrostatic damage. The cell library was designed using Cadence Edge and is now integrated in the Cadence 4.2 environment. The following views are provided for each cell in the library: layout, abstract (for place and route), extracted, schematic, symbol, and verilog, as well as a functional level schematic (funsch) for other cells.

5.2 LOGIC DESIGN

5.2.1 Detailed Architectural Organization

Fig 5.2 shows a detailed architecture of the processor, with complete internal component interconnections and external interface. The different functional blocks shown are discussed in chapter 4, and the functions of the processor input/output pins are summarized in Table 5.1. The following sections give a brief description of the logic design and verification of the main architectural components. DATA(15:0) and INST(15:0) are 16-bit multiplexed buses, providing access to the external data memory space and the instruction memory space respectively.

Pin	Direction	Description
VDD	Input	Power Supply
GND	Input	Power Supply
RST	Input	Reset
EXTCLK	Input	External Clock
IRACK	Input	Instruction Acknowledge
DTACK(3:0)	Input	Data Acknowledge
CS(3:0)	Output	Chip Select
RW	Output	Read/Write
ALE	Output	Address Latch Enable
INST(15:0)	Input/Output	Instruction Bus
DATA(15:0)	Input/Output	Data Bus

Table 5.1 Processor External Interface Pins

2 of / de 2

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Fig 5.2 Detailed Processor Architecture

5.2.2 Architectural Components Design and Verification

5.2.2.1 The Control Unit

The Control Unit generates all the control and timing signals of the processor. It is composed of six functional blocks: the Clock Generator, the Pipeline Registers, the Instruction Decoder, the Branch Decision Circuit, the No-Operation Generator and the Timing Circuit. The Control Unit logic design was synthesized from the structural VHDL description of the six interconnected blocks. Fig 5.3 is the logic diagram of the Control Unit. The design of its logic components is summarized in the following sections.

Fig 5.3 The Control Unit Logic Diagram

1. The Instruction Decoder

The Instruction Decoder decodes specific instructions and generates a number of control signals as shown in Fig 5.3. Because of the simplified and uniform instruction execution scheme, and the use of some opcode bits as control signals, only those instructions whose execution sequence deviate from this general format, and/or generate special control signals, need to be decoded. These instructions include BRANCHes, LOAD/STORE, MVI, and CMP/BTST. The decoding circuit was designed from the behavioral VHDL description of the decoding process. The following are example lines from the Decoder VHDL code:

```
-- ** Instructions decoded from the instruction word
MOVEB2 <= IR2(15) and IR2(14);      -- decode a general MOVE byte
:
:
-- ** Instructions decoded from other decoded signals
MVI <= MOVEB2 and IR2(13);          -- decode Move Immediate
:
:
-- ** Control signals from instruction decoded signals
if MOVEB2 = '1' then-- if on MOVE byte, then
    ZEROH <= '1';                  -- clear high byte of the operand
else
    ZEROH <= '0';
end if;
:
:
```

2. The No-Operation Generator (NOOP)

This circuit cancels the nth pipeline stage of specific instructions by setting the NOP(n) signal to high. An example is cancelling stage 4 (Writing into the Register File) when executing CMP. The NOOP circuit was also synthesized from its VHDL behavioral description, in much the same format and simplicity as the Timing Circuit. Fig 5.3 shows the inputs to this circuit, and hence the instructions that generate no operations. Details of instructions no-operations are described in Chapter 4 under Instruction Execution.

3. The Timing Circuit

The timing circuit generates two sets of signals. The first set controls the execution of pipeline stages, depending on the no-operation signals, NOP(4:1). These signals are LDIR (Load instruction), LDSX (Load RALU Input Registers), LDOR (Load RALU Output Register), and WRL/WRH (Write to low/high byte of the Register File). The second set are general-purpose register enable signals, consisting of LDNP (Load NOP registers), LDFG (Load flags registers), LDPC (Load the Program Counter), and LDBR (Load Branch PC values). These register enable signals are set to '0' even when they are don't cares in order to reduce node transition activities, and hence power, in the registers [57]. The timing circuit was also designed with behavioral VHDL description. The code lines below illustrates the general format of such behavioral design:

```

if NOP(2) = '1' then
    LDSX <= '0';
else
    LDSX <= CLK3;
end if;
-- if no operation in stage 2, then
-- don't load RALU input registers,
-- otherwise
-- load, using clock CLK3.
```

4. The Branch Decision Circuit

The Branch Decision Circuit determines whether a branch is valid or not. It does this by comparing the branch instructions condition bits, IR(10:8), and the status of the flags register. Very simple VHDL code was used to describe this, and is illustrated below:

```

if BRCOND = "000" or
   (BRCOND = "001" and FLG(2) = '1') or
   :
   :
   (BRCOND = "111" then
       BRA <= '1';
   else
       BRA <= '0';
   end if;
BR2VAL <= BRCODE and BRA;
BR2INV <= BRCODE and (not BRA);
-- BRA/JUMP; unconditional
-- BEQ (BZ); equal or zero
:
:
-- RESERVED; unconditional
-- Branch condition valid,
-- otherwise
-- Branch condition not valid
-- Valid branch (stage 2)
-- Invalid branch (stage 2)
```

5. The Pipeline Registers

The Pipeline Registers are composed of instruction and NOP registers. Instruction registers contain the instruction words for each of the four pipeline stages. For stages 3 and 4, only part of the 16-bit word is stored since these stages do not require the other bits of the instruction word. The NOP registers store the status of the no-operation, NOP(n), of the four stages. The value of NOP(n) is derived either from the No-Operation Generator, or from the no-operation status of the previous stages, NOP(n-1). Apart from these two sets of registers, there is also a register to propagate the status of WRTHEN control signal into the following stage. All registers used are d flip-flops, triggered on the falling edge of their enable signals. Since the d flip-flops in the CMOS4S standard cell library do not have a reset signal, this capability was included by redesigning the registers using VHDL behavioral description. Otherwise, the Pipeline Registers block was coded structurally, and the logic design details are shown in Fig 5.4.

Fig 5.4 Pipeline Registers

6. The Clock Generator Circuit

The Clock Generator Circuit generates four clocks from the external clock, and also inserts wait states when the processor is accessing slow I/O and/or memory devices. It is composed of two functional blocks: a 4-bit Pulse Shifter and a Wait State Generator. The four clocks are produced when a pulse is shifted on the positive edge of MPCLK (the internal version of the external clock, EXTCLK). The wait state generator regulates MPCLK by negating it when the processor is performing an I/O operation and the DTACK signal is not yet asserted by the time CLK(3) rises. The following code lines illustrate the VHDL design implementation of the Clock Generator Circuit:

```
-- ** 4-bit Shifter
  if (MPCLK'event and MPCLK = '1') then           -- on rising edge,
    CLK(3) <= CLK(2);                             -- shift the pulses
    CLK(2) <= CLK(1);
    CLK(1) <= CLK(0);
    CLK(0) <= CLK(3);
  end if;

-- ** Wait-State Generator
  IOACKLAT <= (not EXTCLK) and CLK(3);
  if (IOACKLAT'event and IOACKLAT = '1') then
    if (DTACK = '1' and IOS = '1') or IRACK = '1'; -- on slow access
      RDY <= '0';
    else
      RDY <= '1';
    end if;
  end if;

  MPCLK <= RDY and EXTCLK;
```

The above code lines do not include the initialization (reset) and declaration lines, and neither do they show all the exact variables and syntax make-up of the Clock Generator VHDL design. Figure 5.5 shows the four internal clocks, and the wait state insertion sequence. The phase one clock, CLK0, is also used as the Address Latch Enable signal (ALE). This is used externally for latching the address on both the data and instruction buses (DATA and INST). The DTACK and IRACK signals are sampled when CLK3 is

high. If an external instruction is executing, and DTACK is not yet asserted on the negative edge of EXTCLK when CLK3 is high, a wait state is inserted. For code memory, this happens when IRACK is not asserted when CLK3 is high. For fast memory (I/O) which assert DTACK/IRACK earlier than when CLK3 goes high, they must keep these signals asserted until at least on the falling edge of EXTCLK when CLK3 is high. Otherwise it will be interpreted as a slow access and wait states will be inserted.

Fig 5.5 Clocking and Wait State Insertion

There is a wide variation of the amount of circuitry needed for clock generation on different microprocessors [38]. Some microprocessors include a clock generator on-chip, thus requiring only a crystal and other minor discrete components to be connected to their external clock pin. On the other hand, some microprocessors/MCU's require an external clock generator chip to supply the precise clock waveforms. This chip can be full custom-designed, or it can be semicustom, using gate-arrays [63], etc. However, in this processor design, a very simple clock generator circuit is implemented on-chip as described above. Since the final integrated processor design will require more elaborate and extended clock/timer functionalities, the clock circuit currently implemented is more for architectural simulation and performance evaluation purposes than providing exhaustive timing requirements of the final design. For example, the final clock circuit will have to be designed with a phase-locked loop (PLL) and with more controlled on-chip distribution schemes in order to improve clock stability and reduce on-chip clock skews. This is outside the scope of our initial design goals, whose emphasis has been to prescribe the basic architecture of the processor core. Therefore, the clock circuit is one of the few architectural components that will have to be updated and conventionally designed when future integration of the processor is pursued.

5.2.2.2 The Register Arithmetic and Logic Unit (RALU)

The RALU is composed of a 16-bit ALU, a 1-bit shifter, a bit-testing circuit and the flags registers. It also contains logic for conditional carry generation, zero result checking, and multiplexing. Figure 5.6 shows the details of the implemented circuit. This, again, was designed and synthesized from the structural VHDL description of the RALU circuitry.

Most of the functional blocks of the RALU have been implemented using standard designs. There are numerous schemes for improving the performance of general-purpose

ALUs. The more recent ones include high performance 3-1 Interlock Collapsing ALUs [64], reduced-area schemes for Carry-Select Adders [65], Superpipelined Adders [66], and other new concepts in the design of Carry-Lookahead Adders [67]. However, a simple and more conventional ALU is employed in this processor mainly because of design simplicity and regularity, as well as easy mapping into VHDL. The 16-bit ALU uses a standard 16-bit Carry-LookAhead (CLA) adder for addition and subtraction. The 4-bit CLA circuitry, upon which the adder is based, was designed from the behavioral VHDL description of standard propagate and generate, group carry generate, sum, and carry lookahead circuitries. The RALU employs a regular array of 16 uniform standard cells to implement XOR, OR and AND logical functions. The CMOS4S library does not have AND- and OR-gates, and hence the logical functions of AND and OR are realized using NAND and NOR gates, followed by inverters.

Fig 5.6 RALU Logic Diagram

The shifter is composed of sixteen 4-input multiplexers, that allows 1-bit left and right shifts, as well as high and low bytes swap. The fourth function of the shifter is to transfer the input operand to the output. The Bit-testing circuit is designed using a decoder-like VHDL description as shown below:

```

if OP1(INT(OP2)) = '0' then
    BTST <= '1';
else
    BTST <= '0';
end if;

```

The BTST signal is set to '1' when the tested bit is '0' so as to use this signal to directly set the zero flag. Similarly, the zero-checking circuit is implemented with behavioral design, that sets a special zero signal whenever the 16-bit input word is zero. The carry circuit generates the carry signal depending on the selected RALU function, and is therefore implemented as a conditional circuit, in much the same way as the other RALU multiplexers. Since a number of instructions execute through the RALU, the carry circuit basically determines which of these instructions should update the carry flag. All logical instructions (XOR, OR and AND), all shift instructions (LSL, LSR, SWAP and MOVEs) and BTST instruction will automatically clear the carry flag. The only instructions that update the carry flag as a result of the RALU computation are ADD, SUB and CMP. Branches and Subroutine instructions, plus LOAD and STORE, do not change any of the three flags. Therefore when executing these instructions, the LDFG signal is negated in the Timing Circuit. The flags registers are three negative edge-triggered d-flip flops, redesigned like instruction and NOP registers to include the reset capability. The RALU inputs are always routed from its input registers and its output is always stored in an output register and hence the name Register Arithmetic and Logic Unit (RALU). These registers, shown in Figure 5.2 as registers A, B and O, are 16-bit d flip-flops, without reset, and are preceded by multiplexers for their input selection.

5.2.2.3 External I/O Interface Unit

The interface unit has two functional blocks: the Code Interface Unit (CIU), which provides interface to the code (instructions) memory, and the Data Interface Unit (DIU), which is a general interface to all other external memory and I/O devices. The Code Interface Unit is composed of one 16-bit tristate buffer and a 16-bit transceiver (bidirectional buffer). This ensures that the INST bus is both multiplexed and buffered for reading the code memory. The Data Interface Unit is an extension of the CIU, allowing both read and write. The DIU also generates four Chip Select signals by decoding the two most significant bits of the address. The DTACK multiplexer selects the DTACK signal from one of the four devices (selected by the Chip Select signal). The Interface Unit top-level design shown in Figure 5.7 was synthesized from a structural VHDL description.

Fig 5.7 External Interface Unit Logic Diagram

5.2.2.4 The Program Counter Unit (PCU)

The synthesized logic diagram of the PC unit is shown in Figure 5.8. The functions of the unit can easily be deduced from its functional blocks. The current value of the Program Counter, PCO(15:0), is incremented using a standard 16-bit Carry-Lookahead incrementer. Normally, the next value of the Program Counter is taken from the output of the incrementer. The two bus multiplexers select the next value of the Program Counter, PCI(15:0), as either the reset address, 0x0000, or the branch-to address, ROUT(15:0), which is calculated by the general-purpose RALU, or the normally incremented PC value. The two 16-bit registers, which are again an array of negative edge-triggered d flip-flops, are used to store the last two values of the Program Counter. PC1(15:0) is used as the base address when the PC is selected as such by branch and subroutine instructions. PC2(15:0) is the value saved as the return-to address by subroutine instructions. Of course, both PC1 and PC2 essentially contain the same value (the address of the instruction following the executing branch/subroutine instruction). Two registers are however necessary since these two values are required in different pipeline stages.

Fig 5.8 PC Unit Logic Diagram

5.2.2.5 The Register File

The register file (Fig 5.9) is implemented with an array of sixteen 16-bit registers. These registers are designed with two enable signals to allow for independent writing to the high- and low-order bytes. An address decoder, implemented as a standard 4-to-16 decoder, is used to select the register and the byte (using WRTL B and WRTHB) to write to. During a read operation, the two output multiplexers select the output from one of the sixteen valid registers. The first output multiplexer is connected to only the first eight registers because the address of the first output can only be 0 to 7. The other multiplexer is however connected to all 16 registers, including register 15, which is the Program Counter. From this, it is evident that reading the Program Counter register is possible, but it cannot be written to like the other fourteen registers. However, the value of the PC can be changed by using unconditional branch or jump instructions.

Fig 5.9 Register File Logic Diagram

5.3 TIMING ANALYSIS

5.3.1 Logic Delays and Critical Path Determination

The instruction execution critical path was determined from the critical paths of the individual pipeline stages. The timing analysis of all possible operations in each stage was performed using the Synopsys Design Time software package. The critical paths timings were then verified with gate-level simulations, and the results are summarized in Table 5.2. Stage 3 is the critical stage, requiring the longest time (28.07 ns).

Pipeline Stage	Critical Path Operation	Critical Signal	Maximum Delay (ns)
1	Increment PC	PCI(15:0)	15.02
2	Read from Register File	BOU(15:0)	4.5
3	ALU Zero Flag	ZF	28.07
4	Write to Register File	LCnn(15:0)	5.4

Table 5.2 Critical Path Delays

5.3.2 Processor Timing and Speed Evaluation

The processor internal timing signals are shown in Fig 5.10. All the signals shown are derived from the clock signals of Fig 5.5. T1 is the time required for the slowest stage to execute (28.07 ns), and is rounded off to 30 ns. T2 is the minimum time between LDFG and LDNP. It is the time required to execute through the Branch Decision Circuit and the No-Operation Generator. T2 is measured with reference to LDFG to ensure that the flags set by the previous instruction are used by the current branch instruction to check whether

it is valid or not. T_2 was measured to be 5.96 ns, but is set to 10 ns to allow for global timing tolerances due to layout parasitics. The machine cycle period ($T_1 + T_2$) is therefore calculated to be 40 ns, resulting in the external clock period (T) of 10 ns. The minimum value of T_3 is 10 ns, ensuring that data is written to the register file before the RALU output register is updated by LDOR. The internal machine cycle speed is therefore 25 MHz, with an external clock running at 100 MHz.

Fig 5.10 Processor Internal Timing

In this chapter, the logic design of the individual architectural components has been presented. This consists of the iterative design process of VHDL description, simulation and synthesis. The output of this design phase was the detailed schematics of the processor circuits. These schematics were then imported into the Cadence design platform for the final layout design. The details of the layout design phase are presented in Chapter 6, together with the system simulation (simulation of the processor with the external clock and external data and instruction memory). The system simulation was however conducted as part of the logic design phase, using the Synopsys design tools, and therefore preceded the layout design phase.

Chapter 6

SIMULATION AND LAYOUT DESIGN

6.1 SYSTEM SIMULATION

6.1.1 System Simulation Architecture

The processor was simulated with code running from a simulated EPROM. Fig 6.1 shows the system simulation architecture, which consists of the gate-level model of the synthesized microprocessor circuit, plus the behavioral VHDL entities of the EPROM, RAM, Clock and Address Latch circuits. The detailed behavioral simulations of the unsynthesized blocks are described in the next section. The whole system was simulated using Synopsys VSS Expert Simulator, just like in the logic design verification phase.

Fig 6.1 System Simulation Architecture

6.1.2 RAM, EPROM and CLOCK Simulations

The functionality verification of the processor requires an external clock, address latches and code and data memory. However, the design of these components is not part of this thesis, and hence it was not necessary to use the synthesized versions of these circuits. Instead, their behavior was described in VHDL and simulated together with the gate-level processor circuit. Fig 6.2 shows the detailed implementation of the behavioral design of the EPROM and the CLOCK circuits. The RAM is designed similar to the EPROM, with modifications to allow for both read and write. The address latches are implemented as standard transparent latches such as the 74LS373, because of their speed advantage over edge-triggered registers. The output reflects the input when EN is high, and is latched when EN goes low. The delays d1 and d2 for the EPROM and RAM codes were replaced with actual timing values during simulations to emulate different memory (I/O) speeds. Use of such behavioral models allow running the actual processor machine code in testing the processor's functionalities, thus saving time in generating the input test vectors and collecting the processor test outputs [46, 53].

6.1.3 Software Performance Evaluation

6.1.3.1 Programming Example

Fig 6.3 shows one of the programs that was run on the processor (from the EPROM) during the simulations. The fundamental goal of these programs was to verify the execution of each of the 28 instructions. However, the programs were also designed in such a way as to test a number of code issues such as depth of subroutine calls, similarity of the test program to typical application codes, effect of 'too many' and, in some cases, consecutive branches, etc. Because of the memory-mapped I/O, and the provision of a separate bus for code memory, the test programs were designed with no special attention to external memory (I/O) instructions.

```

entity EPROM is
  port( EN      : in STD_LOGIC;
        ADDR    : in STD_LOGIC_VECTOR ( 6 downto 0);
        DATA    : out STD_LOGIC_VECTOR (15 downto 0);
        DTACK    : out STD_LOGIC);
end EPROM;

architecture A of EPROM is
begin
  process (EN,ADDR)
  begin
    if EN = '0' then
      DTACK <= '0' after d1 ns;
      if ADDR = "0000000" then
        DATA <= "1010 1110 0100 0000" after d2 ns;
      elsif ADDR = "0000010" then
        DATA <= "0000 0000 0000 0111" after d2 ns;
        :
        :
      end if;
    else
      DTACK <= '1' after d1 ns;
      DATA <= "ZZZZ ZZZZ ZZZZ ZZZZ" after d2 ns;
    end if;
  end process;
end A;

```

NOTE: d1 is set to less than 40 ns to emulate fast memory, and to greater than or equal to 40 ns to emulate slow memory
d2 is set to 1 ns for simulation purposes (actual value depends on specific memory IC)

```

entity CLOCK is
  port( CLKOUT : out STD_LOGIC);
end CLOCK;

architecture A of CLOCK is
  constant PERIOD : time := 10 ns;
  constant HALF_PERIOD : time := PERIOD / 2;
  signal CLK : STD_LOGIC := '1';
begin
  CLKGEN : process (CLK)
  begin
    wait for HALF_PERIOD;
    CLK <= not CLK;
  end process CLKGEN;
  CLKOUT <= CLK;
end A;

```

Fig 6.2 EPROM and CLOCK VHDL Behavioral Designs

Label	Address	Mnemonic	Code	Comment
;main:	0000000:	LDI 14	AE40	main program
	0000010:	0007	0007	
	0000100:	LDI 13	AD40	
	0000110:	4014	4014	
	0001000:	MOVEW 7, 13	B73D	
	0001010:	SUB 10, 7, 13	1A7D	
	0001100:	SWAP 12, 13	BC2D	
	0001110:	ADD 11, 7, #2	0BF2	
	0010000:	XOR 8, 7, 12	287C	
	0010010:	LSL 9, 11	B90B	
	0010100:	AND 14, 7, 13	6E7D	
	0010110:	BSR 0, PC, #8	4888	
	0011000:	MVI 7, #33	E733	
	0011010:	LDI 5	A540	
;end:	0011100:	FFFE	FFFE	end
	0011110:	BRA PC, 5	8805	goto end
;sub1:	0100000:	MOVEB 8, 11	C83B	subroutine 1
	0100010:	MOVEBC 10, 11	DA3B	
	0100100:	CLRW 9	F900	
	0100110:	MVIC 12, #3	FC03	
	0101000:	STORE 7, 12	707C	
	0101010:	CMP 7, 11	907B	
	0101100:	BEQ PC, #2	8982	
	0101110:	LOAD 8, 12	A80C	
	0110000:	BSR 3, PC, #2	4B82	
	0110010:	JUMP 0	8080	
;sub2:	0110100:	BTST 7, #3	98F3	subroutine 2
	0110110:	BZ PC, #2	8982	
	0111000:	CLRB 14	EE00	
	0111010:	MVIC 7, #2	F702	
	0111100:	BSR 6, PC, #2	4E82	
	0111110:	JUMP 3	80B0	
;sub3:	1000000:	OR 7, 7, #3	57F3	subroutine 3
	1000010:	JUMP 6	80E0	

Fig 6.3 Example Test Program

6.1.3.2 Performance Evaluation

Because of lack of performance evaluation tools, software simulators, and conventional benchmark programs, performance was evaluated simply by comparing the time required for the test programs to run on the processor with that required on its preceding designs. This was done by multiplying the number of cycles required to run the whole program with the period of the machine cycle. Table 6.1 shows the results, based on the test program of Fig 6.3. It must, however, be pointed out that even with unlimited resources, the use of benchmarks, such as SPEC, which are widely accepted criterion for comparing microprocessors performance, is less useful for embedded applications because of the huge diversity of such applications [44].

Processor Under Test	Processor Frequency	Number of Instructions	Number of Cycles	CPI	IPS
Current Processor	25 MHz	31	49	1.58	510412
Previous Design ⁺	10 MHz	31	93	3.0	107526
Original Design ⁺	10 MHz	31	148	4.77	67626

⁺ Refer to Chapter 3 for details on these designs

Table 6.1 Processor Performance Evaluation

CPI is the average number of clock cycles per instruction, and is calculated using the equation below (page 13). The calculations shown are for the current processor.

$$\text{CPI} = \frac{\text{No. Of Clock Cycles}}{\text{No. Of Instructions}} = \frac{49}{31} = 1.58$$

IPS is the number of Instructions Per Second. It is normally given in terms of Million Instructions Per Second (MIPS). The calculation for IPS is performed using the following equation:

$$\text{IPS} = \frac{\text{Processor Frequency}}{I \times \text{CPI}} = \frac{25 \times 10^6}{31 \times 1.58} = 510412$$

The evaluation described above is by no means the ideal procedure, neither is it an exhaustive one. A number of factors, such as choice of the test program, the processor architecture, and the implementation technology, have to be taken into account before an absolute performance analysis can be carried out. With limited resources, a simple analysis like the one presented here still suffices.

6.1.4 Sample Simulation Results

Figures 6.4 shows sample simulation results when running the test code of Fig 6.3. The results show the execution of code starting at the address 0100100 of subroutine 1. Six machine cycles are captured, and hence full execution of the instructions CLRW 9, MVIC 12, #3 and STORE 7, 12 can be observed. Only selected signals are shown. Notice the timing signals and the updating of instruction and NOP registers, as well as the values of the RALU input registers (AOUT and BOUT) and the output register (ROUT). The execution of a STORE instruction with a slow external memory (with access time equal to 42 ns) is also highlighted. Note also the wait state insertion sequence, and the bus transfer handshaking using ALE, CS, IOS and DTACKO signals.

Fig 6.4 Sample Simulation Results

6.2 LAYOUT DESIGN

6.2.1 Layout of Functional Blocks

Before laying out the whole chip, the layouts of the individual main functional blocks were produced. An analysis was then performed to evaluate the area occupied by each block. From this, it is easy to measure how much saving in area is achieved by an individual block, such as the control unit, over the conventional reference designs. If certain blocks occupy less area than originally projected, or as often encountered, then the saved area can be used to either increase the functionality of the other blocks, or introduce on the chip extra components such as timers. This inclusion of extra components is not pursued in this thesis because it requires further studies, and hence is left for future work. The area saved by a reduced and simplified control unit was used to increase the size of register file from 8 to 16 registers. This is not final, and with more analysis, the register file could be further increased to allow for increased data storage and an extended internal stack implementation. Table 6.2 shows the layout sizes of the processor main blocks.

Processor Block	Width (mm)	Height (mm)	Area (mm ²)	Percentage
Control Unit	0.7444	0.8939	0.665434	11.1%
RALU	0.9122	1.2413	1.132418	18.9%
Register File	1.5126	1.7678	2.674105	44.7%
PC Unit	0.7167	0.8152	0.584270	9.8%
Interface Unit	0.5372	0.6292	0.338056	5.7%
Register A	0.4266	0.4067	0.173523	2.9%
Register B	0.4734	0.5169	0.244749	4.1%
Register O	0.4013	0.4198	0.168506	2.8%
Total Area	-	-	5.981064	100.0%

Table 6.2 Layout of Functional Blocks

6.2.2 Processor Chip Layout

The chip was laid out as a single flattened layout, as opposed to routing up the individual blocks, in order to reduce the overall chip area. Fig 6.5 shows the final processor chip layout. The overall area is 18.46 mm^2 ($4.12 \text{ mm} \times 4.48 \text{ mm}$). The core area is however only 7.85 mm^2 ($2.38 \text{ mm} \times 3.29 \text{ mm}$). The total area shown in the Table 6.2 (5.98 mm^2) is not equal to the final chip layout area since it does not include inter-block routing and I/O pads. It was evaluated by simply adding the individual layouts.

On average, the Cell Ensemble Place & Route tool uses half of the total area for routing. This routing overhead is generally reduced in technologies with more metal layers. A detailed layout of a very simple processor block is illustrated in Fig A.1 of Appendix A2. The chip layout shown in Fig 6.5 was however drawn from the abstract views of the cells in order to minimize the time for plotting the detailed layout.

Fig 6.5 Processor Chip Layout

Chapter 7

CONCLUSIONS

7.1 THESIS SUMMARY AND CONCLUSION

Microprocessors play a vital role in the design of instrumentation and measurement equipment. Typically, microprocessor-based controllers are used to provide more elegant system interface, allow more automatic and advanced data collection, measurements, processing, and analysis, enhance remote control capabilities and improve user interface. This thesis summarizes the results of the design of such a processor. The main design objective was to produce a low-cost architecture, which would later be integrated with other application-specific hardware units such as timers. Therefore a typical instrument control application had to be considered. The Bar Tester was chosen because of its advantages for a design project like this one. Firstly, background work has been done on the design of the integrated system for the Bar-Tester System. While the data and control paths of the two designs are significantly different (the previous design followed a semi-RISC approach) much of the design of the instruction set is based on the studies and findings of the previous design. Secondly, the Bar-Tester application offers a moderately high level of control complexity. Therefore a processor designed for it would enjoy wide usage in a number of small and medium scale instruments, for which this design is targeted. Throughout this thesis, the current design is compared to both the previous design and the original discrete component system. Much of the designed processor performance evaluation is restricted to these comparisons.

The processor is designed with an optimized set of 28 instructions. To simplify the architecture, the number of external instructions has been reduced to only two. To cater for

this limitation, a number of internal data movements instructions are provided. The microprocessor is designed with a full RISC architecture. A detailed summary of the RISC design approach is presented in the thesis to highlight the advantages of RISC architectures, such as reduced area, single-cycle execution, and hardwired control. This type of architecture is essential in the design of a simple and low-cost processor by reducing the silicon real-estate while increasing the execution speed.

The following is a summary of the characteristics of the designed processor:

- 26 general-purpose instructions, 2 application-specific instructions, 2 instruction formats. This set is sufficient for execution of the application code, allows for future upgrades, and ensures a simplified and minimum architecture.
- A typical 16-bit RISC data path, with a 16 x 16 register file and a standard RALU, with bit testing circuitry.
- Hardwired control, with on chip clocking and timing circuitry.
- 4-stage instruction execution pipeline, overlapping instruction fetch, read operands, execute, and write to register file.
- Separate address spaces for code and data, accessed with two separate 16-bit multiplexed buses.
- Asynchronous bus transfers, with wait state insertion to allow for slow I/O access.
- On-chip address decoding for generating four Chip Select signals, and selecting one out of the four incoming Data Acknowledge signals.
- Internal processor (machine cycle) speed of 25 MHz, running from a 100MHz external clock. This machine cycle speed, together with the single-cycle instruction execution, gives the designed processor a speed improvement of about five times over the previous design, and more than seven times over the original discrete component system.
- Implemented in 1.2 μm CMC (standard cell) CMOS4S technology, with a chip area of 4.12 mm x 4.48 mm (18.46 mm.²), complexity of about 20,000 transistors, and 49 external pins. This area is less than half of the area that would have been synthesized

from the previous design, with a more optimal area percentage composition than that achieved in the previous design. The control unit occupies less than 10% of the chip and the register file occupies about 40% of the total area.

It must be pointed out that a large number of processors are currently available on the market, which could be used in this application. However, this thesis is focussed on the design of a low-cost architecture which is tailored specifically for instrument control applications. The emphasis has been shifted from the conventional objective of making the processor as general purpose as possible to designing a processor which has reduced silicon area and executes the instrument control application code efficiently. This, therefore means that a cost/feasibility study should be carried out before the designed chip can be implemented over a purchased processor in any control application. Also, while the processor is designed based on a specific application, the resulting architecture will be used for future studies in determining the extent of on-chip processor integration for such applications, as well as other issues such as functionality upgrading and control capabilities. It is for this reason that the laid out chip is implemented without testability circuitry, extended on-chip data storage or RAM, custom hardware units, etc.

The processor is designed with minimum complexity, resulting in a low-power and low-cost design. Besides this, several power management techniques employed in many low-power embedded processors have been included in the future work recommendations. This work therefore also provides an insight into the design of low-power processing units for low-power applications such as portable telecommunications devices. The other major achievement of our work is that it has demonstrated a complete microprocessor design methodology in the newly available design environment. To our knowledge, it is the first such complete microprocessor design at a Canadian University.

7.2 RECOMMENDED FUTURE WORK

Much of the work presented in this thesis forms the basis of the recommended future work on the processor. In general, studies have to be performed to determine the extent to which the on-chip integration of custom units is carried out. This will then be followed by the design and implementation of the selected units onto the processor chip. This might also warrant modifications to the current processor architecture and the instruction set. In this case, more application code examples will have to be examined to better understand the requirements of a typical instrumentation control processor and hence design a more optimal instruction set. However, should such modifications become a necessity, it is recommended that they be kept to a minimum in order to maintain the initial goals of a low-cost and low-power design. As a guideline, it is recommended that units which are general to most instrument control applications, such as timers and interface units, be given priority in case chip area becomes a constraint. On-chip integration can be extended to upgrading the current processor functionalities, such as incorporating a more elaborate shifter for the RALU, implementing separate adders and other circuits for efficient execution of branch instructions, extending the register file to enhance on-chip data storage and emulated cache functionalities, implementing the current register file as an on-chip RAM to reduce silicon area, and extending the general functionalities of the Interface Unit. The biggest challenge in all this is to achieve such integration while maintaining the initial goal of designing a low-cost architecture. The design of programmed architectures for instrument control, based on the design presented in this thesis, and the inclusion of testability circuitry, would be another recommended extension to this work. A number of specific future work recommendations are detailed in the following paragraphs.

One of the two major objectives of our work was to design a low-power architecture. Though much of the work presented in this thesis is focussed on designing the basic architecture with minimum complexity, and hence low power, the principal future extension to this work will be to implement several power management techniques to further

reduce the processor power dissipation. A number of such techniques have been studied and a few are recommended here as future work on this processor design. Since power consumption is proportional to $C_L V_{DD}^2 f_p$ (where C_L is the output capacitive load, V_{DD} is the supply voltage, and f_p is the output switching frequency), each of these can be reduced to minimize power. C_L can be reduced by using an improved process technology. This can result in significant power reduction if the improvement in feature size is substantial. For example, in [26], the use of a 0.64 μm CMOS technology, compared to the previously used 0.8 μm technology saved approximately 25% of the total gate, junction and metal wire capacitances, thus reducing the microprocessor power dissipation from 12.5 W to 9.4 W. Similarly, our design could be implemented in BATMOS [83], the 0.8 μm BiCMOS technology which is now available on our design environment. This would reduce the capacitance, and hence switching power by approximately 30%. However, this would increase the design cost and time because of the higher implementation cost of the BiCMOS technology when compared to the CMOS technology, and also due to the need to design the BiCMOS standard cells and synthesis libraries which are currently not available on our design environment. Therefore the reduction in power dissipation has to be carefully weighted against the increase in the design costs before this option is adopted. The second method is to reduce the supply voltage from 5 V to 3.3 V. This is very common in most recent low-power microprocessors [2, 6, 22, 23, 26] and can scale back power dissipation by as much as 60% without significantly slowing circuit speed [22]. This method is however currently not in much consideration for our design because of the unavailability and cost of the 3.3 V technology. The third approach, which involves reducing f_p , is highly recommended for this processor design because of the reduced implementation costs when compared to the two methods described above. This mainly constitutes dynamic power minimization, in which circuits not required to support the activities of the current clock cycle are dynamically shut [22, 23]. Generally, it is also ensured that the minimum and required logic operate at a given time with minimum node activity [29]. For

example, during branch stall cycles, only the logic required to resolve the stall should operate, thus eliminating re-accessing the register file (or cache) or switching the data buses [2, 3]. Also, constant values can be chosen as the default inputs for multiplexers, instead of inputs fed from random data buses [26]. In this case, when the multiplexer is not in use, its output remains in a steady state to avoid unnecessary switching of signals. Again the area penalty that this dynamic power management would cause have to be evaluated against its power-reduction benefits in order to maintain the initial design goals. A second approach in reducing f_p in this design is to combine features of both RISC and non-RISC machines. This has been successfully implemented in some commercial low-power and low-cost microprocessors such as the Hobbit [1]. Specific to our design, one could implement a variable-length instruction format in order to maximize code density [1, 17]. This reduces the number of memory accesses and processor clock cycles necessary to complete a task, thereby minimizing power consumption. The processor could also be designed to perform the CISC-type memory-to-memory operations rather than just the register-to-register operations typical of RISC microprocessors. Operating directly on memory requires fewer instructions for a given task than a load-store architecture, thereby saving code space and I/O bandwidth [1]. Even though this reduces the processor performance, the non-aggressive performance (speed) requirements of these application-specific microprocessors, and the need for low-power and low-cost designs, especially in embedded and portable applications, usually translates in designing these processors for reduced silicon real estate and minimized power dissipation [1, 2], without thriving to meet the performance levels of other RISC processors which are targeted for Unix-based workstation or general-purpose markets.

Another important future consideration is the extension of the on-chip data storage capabilities of the processor. One general disadvantage of using the 16-bit instruction encoding over the 32-bit format is the limited number of registers that can be referenced and the number of operands per instruction. The sixteen registers that are currently refer-

enced by the processor instructions have been considered adequate for this design. However if future upgrades require referencing more than sixteen registers, the 16-bit instructions could be coded so that the second operand source register is also the destination register. Doing so would have almost no performance penalties since, despite its popularity in many 32-bit RISC instruction sets, the ability to specify a destination register distinct from the operand source registers is widely believed to have little tangible benefits [15]. Alternatively, the extra registers could be added into the architecture by implementing register mapping techniques such as the Register Connection Method [18]. In this method, a set of extended registers are added into the architecture and are accessed by instructions that apparently reference core registers. The register mapping process (i.e. whether instructions reference core or extended registers) is performed by special “register connect” instructions and the mapping information is stored in a table. Another method to extend the on-chip data storage would be to implement an on-chip data cache. In order to reduce the real estate overhead, this cache could be implemented as a small stack cache that replaces the register file [1]. Therefore, the cache has to be designed as a dual-ported memory so that two operands can be fetched and one stored in a single cycle. Instructions can access the stack cache using either the stack offset (an offset from the Stack Pointer) or memory addressing modes. This would ensure that the stack cache size is transparent to the instruction set, thus allowing future processor upgrades to use a larger cache for improved performance without affecting software compatibility.

In order to allow future extension of the instruction set and support complex instructions (such as floating point arithmetic) without considerable change in the hardware structure, a small microcode could be introduced to coexist with the RISC instructions as proposed by Papachristou and Immaneni in [30] and implemented in the Intel 80960-Next Generation processor [12] (see also Section 2.3.4.2). Since the obvious cheap implementation of such instructions would be at the software level (hardware implementation of instructions with complex sequencing constructs is generally not cost-effective for

low-cost processors), the introduction of microcode would allow the migration of such instructions from software to firmware, thus providing significant advantages in speed as well as flexibility for future extensions. The speed degradation that the introduction of microcode would cause to the existing RISC instructions, plus the area overhead of the microcode itself will have to be carefully evaluated to ensure that the design goals are not violated.

Finally, to obtain higher chip density and increased performance, the layout of the datapath circuit could be fully custom-designed (hand-crafted layout), while retaining the standard cell synthesized control path. This is done in some embedded control processors [5] to achieve an even better compromise in performance, design time and die area.

Appendix A1

INSTRUCTION GLOSSARY

ADD SUB	Integer Signed Addition Integer Signed Subtraction
Syntax	ADD Rd, S1, [#]S2 SUB Rd, S1, [#]S2
Operation	Rd = S1 + [#]S2 Rd = S1 - [#]S2
Description	ADD adds the content of source register S1, with the content of source register [#]S2, into the destination register Rd. SUB subtracts the content of source register [#]S2 from the content of source register S1, into the destination register Rd.
Flags	Z Set if the result is zero, otherwise cleared. N Set if the result is negative, otherwise cleared. C Set if carry generated, otherwise cleared.

Format

Execution

XOR	Logical Exclusive OR
OR	Logical OR
AND	Logical AND

Syntax	XOR Rd, S1, [#]S2 OR Rd, S1, [#]S2 AND Rd, S1, [#]S2
---------------	--

Operation	Rd = S1 xor [#]S2 Rd = S1 or [#]S2 Rd = S1 and [#]S2
------------------	--

Description	Computes the logical operations of XOR, OR and AND, respectively, between the content of source register S1, and the content of source register [#]S2, into the destination register Rd.
--------------------	--

Flags	Z Set if the result is zero, otherwise cleared. N Set if the result is negative, otherwise cleared. C Always cleared.
--------------	---

Format

Execution

CMP Integer Signed Compare

BTST Logical Bit Test

Syntax **CMP** S1, [#]S2
 BTST S1, [#]S2

Operation S1 - [#]S2
 test bit S1[#S2]

Description **CMP** subtracts the content of source register [#]S2 from the content of source register S1, and updates the flags registers.
BTST tests if bit number #S2 of the content of source register S1 is set or not, and then updates the flags registers.

Flags Z (CMP) Set if the result is zero, otherwise cleared.
 Z (BTST) Set if the tested bit is zero, otherwise cleared.
 N (CMP) Set if the result is negative, otherwise cleared.
 N (BTST) Always cleared.
 C (CMP) Set if carry is generated, otherwise cleared.
 C (BTST) Always cleared.

Format

Execution

MOVEB Move Byte

MOVEBC Move Byte (Clear High Byte)

Syntax MOVEB Rd, S2
 MOVEBC Rd, S2

Operation Rd = lower byte of S2; High byte of Rd is left unchanged.
 Rd = lower byte of S2; High byte of Rd is cleared.

Description **MOVEB** copies the lower byte of the source register S2 into the destination register Rd.
 The high byte of the destination register is left unchanged.
 MOVEBC copies the lower byte of the source register S2 into the destination register Rd.
 The high byte of the destination register is cleared.

Flags **Z** Set if the byte is zero, otherwise cleared.
 N Always cleared.
 C Always cleared.

Format

Execution

MVI **Move Immediate Byte**
MVIC **Move Immediate Byte (Clear High Byte)**
CLRB **Clear Byte**
CLRW **Clear Word**

Syntax **MVI Rd, #S2**
 MVIC Rd, #S2
 CLRB Rd
 CLRW Rd

Operation lower byte of Rd = #S2; High byte of Rd is left unchanged.
 lower byte of Rd = #S2; High byte of Rd is cleared.
 lower byte of Rd = "00"H; High byte of Rd is left unchanged.
 Rd = "0000"H (Rd is cleared).

Description **MVI** copies the immediate byte #S2 into the lower byte of the destination register Rd. The high byte of the destination register is left unchanged.
MVIC copies the immediate byte #S2 into the lower byte of the destination register Rd. The high byte of the destination register is cleared.
CLRB Clears only the lower byte of the destination register Rd.
CLRW Clears both bytes of the destination register Rd.

Flags **Z** (MVI/MVIC) Set if the byte is zero, otherwise cleared.
 Z (CLRB/CLRW) Always set.
 N Always cleared.
 C Always cleared.

Format

		Rd: Destination Register		Immediate Byte
		Opcode		
MVI	:	1110 XXXX	XXXXXXXXXX	
MVIC	:	1111 XXXX	XXXXXXXXXX	
CLRB	:	1110 XXXX	0000 0000	
CLRW	:	1111 XXXX	0000 0000	

Execution

LOAD Load External Memory Word

LDI Load Immediate Word

Syntax LOAD Rd, S2
 LDI Rd

Operation Rd = MEM(S2)
 Rd = instruction word following LDI

Description **LOAD** reads a 16-bit word of external memory at the address contained in the source register S2, and stores the word in the destination register Rd.
LDI reads an immediate 16-bit word at a location following the LDI instruction, and stores it in the destination register Rd. The LDI instruction takes one cycle to complete but it cancels the next "instruction" since this is the immediate word. By letting this word to be loaded as an instruction, LDI can read it from the first instruction register.

Flags

- Z (LOAD) Not Changed.
- Z (LDI) Set if the word is zero, otherwise cleared.
- N (LOAD) Not Changed.
- N (LDI) Set if the word is negative, otherwise cleared.
- C (LOAD) Not Changed.
- C (LDI) Always cleared.

Format

Execution

STORE Write External Memory

Syntax STORE S1, S2

Operation MEM(S2) = S1

Description Performs a write of a 16-bit word to external memory at the address contained in the source register S2. The word to be stored is contained in the source register S1.

Flags Z Not Changed.
 N Not Changed.
 C Not Changed.

Format

Execution

JUMP	Unconditional Jump
BRA	Unconditional Indexed Branch
BEQ	Branch if Equal or Zero (BZ)
BNE	Branch if Not Equal or Not Zero (BNZ)
BOC	Branch On Carry
BGF	Branch if Greater Than or Equal To
BLT	Branch if Less Than

Syntax **BRANCH S1, [#]S2** (where BRANCH represents any of the above branch mnemonics)
JUMP S1

Operation $PC = S1 + [#]S2$ or $PC = PC + [#]S2$ if branch condition is true.
for JUMP, $PC = S1$.

Description Computes a branch operation to the address resulting from the addition of the source registers S1 and [#]S2 if the tested condition is true. BRA and JUMP always execute. Branches need three cycles to complete and therefore automatically cancel the next two instructions.

Flags Z Not Changed.
N Not Changed.
C Not Changed.

Format

Branch Condition	Base Address is S1 or PC	Opcode	#: S2 Decision Bit	S1: Base Address Register	S2: Offset Source Register or 4-Bit Value Offset
JUMP:	1000 0 000 1 XXX 0000				(unconditional)
BRA :	1000 X 000 X XXX XXXX				(unconditional)
BEQ :	1000 X 001 X XXX XXXX				(if Z is set)
BNE :	1000 X 010 X XXX XXXX				(if Z is not set)
BOC :	1000 X 011 X XXX XXXX				(if C is set)
BGE :	1000 X 100 X XXX XXXX				(if N is not set)
BLT :	1000 X 101 X XXX XXXX				(if N is set)

Execution

BSR Branch to Subroutine

Syntax BSR Rd, S1, [#]S2

Operation $Rd = PC + 1$; $PC = S1 + \left[\# \right] S2$ or $PC = PC + \left[\# \right] S2$.

Description Computes a branch operation to the subroutine at the address resulting from the addition of the source register S1 and [#]S2. BSR needs three cycles to complete and therefore automatically cancels the next two instructions. The PC value to return to is saved in the destination register Rd.

Flags Z Not Changed.
 N Not Changed.
 C Not Changed.

Format

Execution

Appendix A2

DESIGN PHASES SAMPLE OUTPUTS

Fig A.1 shows sample outputs of the main CAD tools used in the design of the processor. The text files for the VHDL descriptions were generated in the Unix-based Emacs text editor. The Synopsys Design Analyzer, which provides the graphical as well as the command-line interface to the other Synopsys tools, was also used to view the synthesized schematics. A sample of the schematic in the Design Analyzer window is illustrated in Fig A.1, together with the results of the Synopsys Design Time critical path analysis (shown as dark outlines in this figure, but otherwise comes in color on the actual window). Versions of this schematic, exported to Synopsys Graphical Environment (SGE) and to Cadence Composer, are also shown. An observation of particular interest is the fact that the EDIF interface preserves the exact graphical details of the schematics when transferring them across the different CAD environments, other than simply saving the connectivity and instantiation of the netlist. This is particularly useful in schematic viewers where specific grid details (such as spacing between an instance input lines, etc.) have to be maintained for the schematic to be correct. This therefore allows the designer to specify such details in the source design environment (in this case, Synopsys), and simply transfer the design using EDIF without having to adjust them again in the target environment (Cadence). Fig A.1 also shows the auto-layout of the Branch Decision Circuit, plus the commands that were used for the EDIF interface.

Fig A.1 Branch Decision Circuit Design Phases

Appendix A3

GATE AND TRANSISTOR COUNT

The count for the number of gates was done in Synopsys Design Analyzer. The number of transistors per gate was derived from the standard cells schematics in Cadence Composer. The total number of transistors for the processor was then calculated by multiplying the number of each type of gates with its respective transistor count. Table A.1 shows the details of this gate and transistor count. The gates used in the design are: inverters (inv), n-input nand gates (nand2, nand3, nand4), n-input nor gates (nor2, nor3, nor4), inverting and non-inverting 2-input multiplexers (mux21i and mux21), d flip-flops (dff), exclusive-or gates (xor), and tristate buffers (bufz).

	inv	nand2	nand3	nand4	nor2	nor3	nor4	mux21i	mux21	dff	xor	bufz	Total
Control Unit	104	82	12	2	16	4	6	7	0	58	7	0	298
RALU	114	99	0	6	34	17	8	1	130	3	48	0	460
Register File	7	14	0	0	34	0	0	0	252	256	0	0	563
PC Unit	57	72	0	4	16	0	0	0	32	32	32	0	245
Interface Unit	11	11	3	0	2	0	0	2	17	0	0	96	142
Registers A,B,O	0	0	0	0	0	0	0	0	124	48	0	0	172
Microprocessor	293	278	15	12	102	21	14	10	555	397	87	96	1880
Transistors/Gate	2	4	6	8	4	6	8	10	12	22	12	8	8.5
Total Transistors	586	1112	90	96	408	126	112	100	6660	8734	1044	768	19836

Table A.1 Processor Gate and Transistor Count

REFERENCES

1. P. V. Argade, et al, "Hobbit: A High-Performance, Low-Power Microprocessor", *IEEE COMPCON*, Spring, 1993, pp 88-95.
2. B. Zivkov, et al, "R4200: A High-Performance MIPS Microprocessor for Portables", *IEEE COMPCON*, Spring, 1994, pp 18-25.
3. N. Yeung, et al, "Unified Datapath: An Innovative Approach to the Design of a Low-Cost, Low-Power, High-Performance Microprocessor", *IEEE COMPCON*, Spring, 1994, pp 32-37.
4. M. Gupta and B. Zivkov, "A Structured Verification Approach for MIPS Microprocessors: A Case Study of the R4200", *IEEE COMPCON*, Spring, 1994, pp 26-31.
5. S. R. Wang, et al, "A 66 MHz PA-RISC™ Embedded Controller with 80486DX-Like Bus Interface", *IEEE COMPCON*, Spring, 1994, pp 53-57.
6. T. Okada, et al, "A PA-RISC™ Microprocessor PA/50L for Low-Cost Systems", *IEEE COMPCON*, Spring, 1994, pp 47-52.
7. D. Gajski, "Introduction to High-Level Synthesis", *IEEE Design and Test of Computers*, Winter, 1994, pp 45-54.
8. A. Gonzalez and J. Llaberia, "Reducing Branch Delay to Zero in Pipelined Processors", *IEEE Transactions on Computers*, Vol. 2, No. 3, March, 1993, pp 363-371.
9. T. Riordan, et al, "System Design Using the MIPS R3000/3010 RISC Chipset", *Proc. 34th IEEE Computer Society Conference, COMPCON'89*, February, 1989, pp 494-498.
10. R. B. Garner, et al, "The Scalable Processor Architecture (SPARC)", *Proc. 33rd IEEE International Computer Society Conference, COMPCON'88*, February, 1988, pp 278-283.

11. R. W. Edenfield, "The 68040 processor. Part1, Design and Implementation", *IEEE Micro*, Vol. 10, No. 1, February, 1990, pp 66-78.
12. G. Hinton, "80960-Next Generation", *Proc. 34th IEEE Computer Society Conference, COMPCON'89*, February, 1989, pp 13-17.
13. J. H. Crawford, "The i486 CPU: Executing Instructions in One Clock Cycle", *IEEE Micro*, Vol. 10, No. 1, February, 1990. pp 27-36.
14. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor", *IBM Journal of Research & Development*, Vol. 34, No. 1, January, 1990, pp 37-58.
15. J. Bunda, et al, "16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors", *20th Annual International Symposium on Computer architecture*, 1993, pp 237-245.
16. P. Steenkiste, "The Impact of Code Density on Instruction Cache Performance", *16th Annual International Symposium on Computer Architecture*, 1989, pp 252-259.
17. D. Patterson, et al, "Architecture of a VLSI Instruction Cache for a RISC", *10th Annual Symposium on Computer Architecture*, June, 1983, pp 108-116.
18. T. Kiyohara, et al, "Register Connection: A New Approach to Adding Registers into Instruction Set Architectures", *20th Annual International Symposium on Computer architecture*, 1993, pp 247-256.
19. R. Leveugle, et al, "The Hyeti Defect Tolerant Microprocessor: A Practical Experiment and its Cost-Effectiveness Analysis", *IEEE Transactions on computers*, Vol. 43, No. 12, December, 1994, pp 1398-1406.
20. S. Y. Kuo and W. K. Fuchs, "Fault Diagnosis and Spare Allocation for Yield Enhancement in Large Reconfigurable PLA's", *IEEE Transactions on Computers*, Vol. 41, February, 1992, pp 221-226.
21. C. L. Wey, "On Yield Considerations for the Design of Redundant Programmable Logic Arrays", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-7, April, 1988, pp 528-535.
22. S. Gary, et al, "PowerPC 603™, A Microprocessor for Portable Computers", *IEEE*

- Design & Test of Computers*, Winter, 1994, pp 14-23.
23. J. Circello and F. Goodrich, "The Motorola 68060 Microprocessor", *IEEE COMP-CON*, Spring, 1993, pp 73-78.
 24. N. Ikumi, et al, "A 300 MIPS, 300 MFLOPS Four-Issue CMOS Superscalar Microprocessor", *IEEE International Solid-State Circuits Conference*, 1994, pp 204-205.
 25. R. Bechade, et al, "A 32-Bit 66 MHz 1.8 W Microprocessor", *IEEE International Solid-State Circuits Conference*, 1994, pp 208-209.
 26. N. K. Yeung, et al, "The Design of a 55 SPECint92 RISC Processor under 2 W", *IEEE International Solid-State Circuits Conference*, 1994, pp 206-207.
 27. E. Rashid, et al, "A CMOS RISC CPU with ON-Chip Parallel Cache", *IEEE International Solid-State Circuits Conference*, 1994, pp 210-211.
 28. T. G. Rauscher and P. M. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments", *IEEE Transactions on Computers*, Vol. C-29, No. 1, January, 1980, pp 2-19.
 29. Z. J. Lemnios and K. J. Gabriel, "Low-Power Electronics", *IEEE Design & Test of Computers*, Winter, 1994, pp 8-13.
 30. C. A. Papachristou and V. R. Immaneni, "Vertical Migration of Software Functions and Algorithms Using Enhanced Microsequencing", *IEEE Transactions on Computers*, Vol. 42, No. 1, January, 1993.
 31. D. Pham, "A 3.0 W 75 SPECint92 85 SPECfp92 Superscalar RISC Microprocessor", *IEEE International Solid-State Circuits Conference*, 1994, pp 212-213.
 32. S. Tanaka, et al, "A 120 MHz BiCMOS Superscalar RISC Processor", *Symposium on VLSI Circuits*, 1993, pp 9-10.
 33. A. Talcot, et al, "The Impact of Unresolved Branches on Branch Prediction Scheme Performance", *21st Annual International Symposium on Computer Architecture*, 1994, pp 2-11.
 34. B. Calder and D. Grunwald, "Fast and Accurate Instruction Fetch and Branch pre-

- diction", *21st Annual International Symposium on Computer Architecture*, 1994, pp 2-11.
35. C. H. Perleberg and A. J. Smith, "Branch Target Buffer Design and Optimization", *IEEE Transactions on Computers*, Vol. 42, No. 4, April, 1993, pp 396-410.
 36. T. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History", *20th Annual International Symposium on Computer Architecture*, 1993, pp 257-266.
 37. Charles Roth, et al, "The PowerPC™ 604 Microprocessor Design Methodology", *IEEE International Conference on Computer Design*, October, 1994, pp 404-408.
 38. D. A. Protopapas, "Microcomputer Hardware Design", Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
 39. M. Slater, "Microprocessor-Based Design", Prentice-Hall, New Jersey, 1989.
 40. K. Hintz and D. Tabak, "Microcontrollers: Architecture Implementation and Programming", McGraw-Hill Inc., New York, 1992.
 41. M. A. Gazier, "Processor Core Design for Instrument Control", *MEng Thesis*, Carleton University, 1993.
 42. A. Abnous, et al, "Design and Implementation of the 'Tiny RISC' Microprocessor", *Microprocessors and Microsystems*, Vol. 16, 1992, pp 187-193.
 43. J. Heudin and C. Panetto, "RISC Architectures", Chapman & Hall, London, 1992.
 44. Bob Caulk, et al, "The Architecture of the LR33000: A MIPS Compatible RISC Processor for Embedded Control Applications", *IEEE International Conference on Computer Design*, October, 1991, pp 333-335.
 45. Raymond Peck and Jay Patel, "Design Methodology for MIPS Compatible Embedded Control Processor," *IEEE International Conference on Computer Design*, October, 1991 pp 324-328.
 46. Darren Jones, et al, "Verification Techniques for a MIPS Compatible Embedded Control Processor", *IEEE International Conference on Computer Design*, October,

1991 pp 329-332.

47. W. G. Spruth, "Design of a Microprocessor", Springer-Verlag, Berlin, 1989.
48. Masahito Matsou, et al, "A 32-bit Superscalar Microprocessor with 64-bit Processing and High Bandwidth DRAM Interface", *IEEE International Conference on Computer Design*, October, 1994, pp 203-210
49. L. Ciminiera and A. Velenzano, "Advanced Microprocessor Architectures", Addison-Wesley, England, 1987.
50. M. Aagaard and M. Leeser, "A Framework for Specifying and Designing Pipelines", *IEEE International Conference on Computer Design*, October, 1993, pp 548-551.
51. G. Doughman, "Use of Stack Simplifies M68HC11 Programming", *Microprocessors and Microsystems*, Vol. 16, 1992, pp 207-210.
52. A. M. Veronis, "Survey of Advanced Microprocessors", Van Nostrand Reinhold, New York, 1991.
53. Tim Brodnax, "The PowerPC 601 Design Methodology", *IEEE International Conference on Computer Design*, October, 1993, pp 248-252.
54. H. Ando, et al, "Speculative Execution and Reducing Branch Penalty in a Parallel Issue Machine," *IEEE International Conference on Computer Design*, October, 1993, pp 106-113.
55. J. Schack, et al, "KISS - 16V1: A 16 Bit Signal Processor", *IEEE International Symposium on Circuits and Systems*, Vol. 4, 1991, pp 1905-1908.
56. J. A. Tierno, et al, "A 100-MIPS GaAs Asynchronous Microprocessor", *IEEE Design and Test of Computers*, Summer 1994, pp 43-49.
57. A. Raghunathan and N.K. Jha, "Behavioral Synthesis for Low Power", *IEEE International Conference on Computer Design*, October, 1994, pp 318-322.
58. V. Nagasamy, et al, "Specification, Planning, and Synthesis in a VHDL Design Environment", *IEEE Design and Test of Computers*, 1992, pp 58-68.
59. C. Jay, "VHDL and Synthesis Tools Provide a Generic Design Entry Platform into

- FPGAs, PLDs and ASICs", *Microprocessors and Microsystems*, Vol. 17, September 1993, pp 391-398.
60. Bob Reese, "Use of VHDL Synthesis in an Advanced Digital Design Course", *IEEE SouthEast Con'92*, Vol. 2, 1992, pp 509-512.
 61. J. R. Armstrong and F. G. Gray, "Structured Logic Design with VHDL", PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993.
 62. L. Clonts, et al, "Writing Area-Efficient Hardware Descriptions for Logic Synthesis", *IEEE SouthEast Con'92*, Vol. 2, 1992, pp 505-508.
 63. E. Ling, "A Semicustom Clock/Timer Chip", *IEEE International Symposium on Circuits and Systems*, Vol. 4, 1991, pp 1909-1911.
 64. J. Phillips and S. Vassiliadis, "High- Performance 3-1 Interlock Collapsing ALU's", *IEEE Transactions on Computers*, Vol. 43, No. 3, March 1994, pp 257-268.
 65. A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders", *IEEE Transactions on Computers*, Vol. 42, No. 10, October 1993, pp 1163-1168.
 66. I. H. Unwala and E. E. Swartzlander, Jr., "Superpipelined Adder Designs", *IEEE International Symposium on Circuits and Systems*, Vol. 3, 1993, pp 1841-1844.
 67. Z. Wang et al, "New Concepts for the Design of Carry Lookahead Adders", *IEEE International Symposium on Circuits and Systems*, Vol. 3, 1993, pp 1837-1840.
 68. "VHDL Compiler Reference Manual", Version 3.0, Synopsys Inc., Nov., 1992.
 69. "Design Compiler Reference Manual", Version 3.0, Synopsys Inc., Nov., 1992.
 70. "Design Analyzer Reference Manual", Version 3.0, Synopsys Inc., Nov., 1992.
 71. "Command Reference for Synthesis", Version 3.0, Synopsys Inc., Nov., 1992.
 72. "Design Time Reference Manual", Version 3.0, Synopsys Inc., Nov., 1992.
 73. "EDIF Interface User Guide", Version 3.0, Synopsys Inc., Nov., 1992.
 74. "VSS Expert Core Programs", Version 3.0, Synopsys Inc., Nov., 1992.
 75. "VSS Family Packages", Version 3.0, Synopsys Inc., Nov., 1992.
 76. "Command Reference for Simulation", Version 3.0, Synopsys Inc., Nov., 1992.

77. "Synopsys Graphical Environment User Guide", V. 3.0, Synopsys, Nov., 1992.
78. "Library Compiler Reference Manual", Vol. 1 & 2, V. 3.0, Synopsys, Nov., 1992.
79. "CMC CMOS4S Environment V0.3 for Cadence 4.2", Cadence Inc., Dec., 1993.
80. "Design Framework II User Guide", Cadence Design Systems Inc., June, 1993.
81. "Design Entry: Composer User Guide", Cadence Design Systems Inc., June, 1993.
82. "Preview Cell Ensemble", Cadence Design Systems Inc., September, 1993.
83. "Primer on CMC 0.8-Micron BiCMOS, A Version of NTE BATMOS", Canadian Microelectronics Corporation, January, 1992.
84. "PowerPC™ 604 RISC Microprocessor Technical Manual", IBM Corp. and Motorola Inc., 1994.

BIBLIOGRAPHY

1. M. Muller, "ARM6: A High-Performance Low-Power Consumption Macrocell", *IEEE COMPCON*, Spring, 1993, pp 80-87.
2. C. Su, et al, "Saving Power in the Control Path of Embedded Processors", *IEEE Design & Test of Computers*, Winter, 1994, pp 24-30.
3. A. Smailagic, "A Case Study in Embedded-System Design: The VuMan 2 Wearable Computer", *IEEE Design & Test of Computers*, September, 1993, pp 56-67.
4. R. Ernst, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test of Computers*, December, 1993, pp 64-75.
5. K. Hua, et al, "Designing High-Performance Processors Using real address Prediction", *IEEE Transactions on Computers*, Vol. 42, No. 9, September, 1993.
6. F. Corella, "Automated Verification of Behavioral Equivalence of Microprocessors", *IEEE Transactions on Computers*, Vol. 43, No. 1, January, 1994.
7. R. Walker, "The Status of High-Level Synthesis", *IEEE Design & Test of Computers*, Winter, 1994, pp 42-43.
8. I. Park, et al, "Two Complementary Approaches for Microcode Bit optimization", *IEEE Transactions on Computers*, Vol. 43, No. 2, February, 1994.
9. F. Arakawa, et al, "A CMOS 50 MHz CISC Superscalar Microprocessor", *Symposium on VLSI Circuits*, 1993, pp 11-12.
10. R. Barnett, "Microprocessor System Design Techniques" Sigma Press, Cheshire, 1991.
11. S. Surya, et al, "Architectural Performance Verification: PowerPC™ Processors", *IEEE International Conference on Computer Design*, October, 1994 pp 344-347
12. E.L. Hannon, "POWER2 Architecture and Performance", *IEEE International Con-*

- ference on Computer Design*, October, 1994, pp 336-339.
13. N. A. Alexandridis, "Microprocessor System Design Concepts", Computer Science Press, Maryland, 1984.
 14. S. G. Shiva, "Computer Design and Architecture", Harper Collins, New York, 1991.
 15. G. Kuru, et al, "An Advanced Dataflow Processor Architecture Based on a Multiple Input Node Concept", *IEEE South East Con '93*, 1993.
 16. P. Spasov, "Microcontroller Technology", Regent/Prentice Hall, New Jersey, 1993.
 17. S. Athan, "Design and Development of a 68000-Based Self-Contained Sub-Sea Data Acquisition and Control System", *IEEE International Symposium on Circuits and Systems*, 1991, pp 270-272.
 18. B. Naveen, et al, "An Automatic Netlist-to-Schematic Generator", *IEEE Design and Test of Computers*, March, 1993, pp 36-41.
 19. S. Summerfield, "Design Methodology of VLSI with Multiple Valued Logic", *IEEE International Symposium on Circuits and Systems*, Vol. 3, 1993, pp 1702-1705.
 20. G. Jimenez, et al, "RISC-based Architectures for Multiple Robot System", *Microprocessors and Microsystems*, Vol. 16, 1992, pp 177-185.
 21. Francois Anceau, "The Architecture of Microprocessors", Addison-Wesley, 1986.
 22. C.R. Moore, et al, "IBM Single Chip RISC Processor (RSC)", *IEEE International Conference on Computer Design*, October, 1992, pp 200-204.
 23. S. Narita, et al, "Design Methodology for GMICRO™/500 TRON Microprocessor", *IEEE International Conference on Computer Design*, October, 1993, pp 253-257.
 24. A. Saini, "Design of the Intel Pentium™ Processor", *IEEE International Conference on Computer Design*, October, 1993, 258-261.
 25. Louis Baker, "VHDL Programming with Advanced Topics", John-Wiley Inc., 1993.
 26. Jayaram Bhasker, "A VHDL Primer", Prentice Hall, New Jersey, 1992.
 27. R. Lipsett, C. Shaefar and C. Ussery, "VHDL: Hardware Description and Design", Kluwer Academic Publishers, Massachusetts, 1989.

28. C. Wang, et al, "Area and Performance comparison of Pipelined RISC Processors Implementing Different precise Interrupt Methods," *IEEE International Conference on Computer Design*, October, 1993, pp 102-105.
29. P.D. Franzone, "System-Level Specification of Instruction sets," *IEEE International Conference on Computer Design*, October, 1993, pp 552-557.
30. G.X. Hua, et al, "Formal Semantics of VHDL for Verification of Circuit Designs", *IEEE International Conference on Computer Design*, October, 1993, pp 446-449.
31. B. Mitra, et al, "Estimating the Complexity of Synthesized Designs from FSM Specifications", *IEEE Design and Test of Computers*, March 1993, pp 30-35.
32. M. Lynch, "Microprogrammed State Machine Design", CRC Press, Florida, 1993.
33. L. Goodby, et al, "Microarchitectural Synthesis of Performance-Constrained, Low-Power VLSI Designs", *IEEE International Conference on Computer Design*, October, 1994 pp 323-326.
34. R.K. Gupta and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems", *IEEE Design and Test of Computers*, September, 1993, pp 29-40.
35. D.W. Franke and M.K. Purvis, "Design Automation Technology for CoDesign: Status and Directions", *IEEE International Symposium on Circuits and Systems*, Vol. 6, 1992, pp 2669-2672.
36. M.K. Purvis and D.W. Franke, "An Overview of Hardware/Software Codesign", *IEEE Symposium on Circuits and Systems*, Vol. 6, 1992, pp 2665-2668.
37. S. Heath, "Microprocessor Architectures and Systems: RISC, CISC and DSP", Newnes, Oxford, 1991.
38. E. Aardoom and P. Stravers, "An Application Specific Processor for a Multi-System Navigation Receiver", *IEEE International Conference on Computer Design*, October, 1992, pp 128-131.
39. S. B. Morrison, et al, "A high Speed data Acquisition System with a dual port RAM Microprocessor interface", *Microprocessor and Microsystems*, Vol. 15, April 1991,

pp 155-159.

40. R. Sproull, et al, "The Counter flow Pipeline Processor Architecture", *IEEE Design and Test of Computers*, 1994, pp 48-59.
41. S. Desai, "The Architecture of the LR33020 GraphX Processor: A MIPS-RISC based X-Terminal Controller", *IEEE International Conference on Computer Design*, October, 1992, pp 205-208.
42. T. Nanya, et al, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor", *IEEE Design and Test of Computers*, 1994, pp 50-63.
43. B. Alhalabi, et al, "A General Purpose VLSI Chip for Robot Axis Motion Controller", *IEEE International Symposium on Circuits and Systems*, Vol. 6, 1992, pp 3005-3008.
44. W.C. Bruce, et al, "Case Study of 1149.1 Microprocessor Implementations", *Microprocessor and Microsystems*, Vol. 17, 1993, pp 26 -275.
45. M. Gruver, et al "AS/400 PowerPC™-Compatible Semi-Custom Technology", *IEEE International Conference on Computer Design*, October, 1994, pp 197-202
46. J.M. Borkenhagen, et al, "AS/400 TM 64-bit Power PC™-Compatible Processor Implementation", *IEEE International Conference on Computer Design*, October, 1994, pp 192-196.
47. C. Chen and A. Kumar, "Comments on 'Area-Time Optimal Adder design'", *IEEE Transactions on Computers*, Vol. 43, No. 4, April, 1994, pp 507-511.
48. V. Kantabutra, "A Recursive Carry-Lookahead/Carry-Select Hybrid Adder", *IEEE Transactions on Computers*, Vol. 42, No. 12, December, 1993, pp 1495-1499.
49. V. Kantabutra, "Designing Optimum One-Level Carry-Skip adders", *IEEE Transactions on Computers*, Vol. 42, No. 6, June, 1993, pp 759-764.
50. H. C. Torng and M. Day, "Interrupt Handling for Out-of-Order Execution Processors", *IEEE Transactions on Computers*, Vol. 42, No. 1, January, 1993, pp 122-127.
51. K. Suzuki, et al, "A 500 MHz 32-Bit 0.4 μ m CMOS RISC Processor LST", *Internation-*

- tional Solid-State Circuits Conference*, 1994, pp 214-215.
52. J. Schutz, "A 3.3 V 0.6 μ m BiCMOS Superscalar Microprocessor", *International Solid-State Circuits Conference*, 1994, pp 202-203.
 53. T. Agerwala, "Microprogram Optimization: A Survey", *IEEE Transactions on Computers*, Vol. C-25, No. 10, October, 1976, pp 962-973.
 54. S. Isoda, "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", *IEEE Transactions on Computers*, Vol. C-32, No. 10, October, 1983, pp 922-932.
 55. S. Davidson, et al, "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Transactions on Computers*, Vol. C-30, No. 7, July, 1981, pp 460-477.
 56. C. Rao and N. Biswas, "On the Minimization of Wordwidth in the Control Memory of a Microprogrammed Digital Computer", *IEEE Transactions on Computers*, Vol. C-32, No. 9, September, 1983, pp 863-868.
 57. J. Baer and B. Koyama, "On the Minimization of the Width of the Control Memory of Microprogrammed Processors", *IEEE Transactions on Computers*, Vol. C-28, No. 4, April, 1979, pp 310-318.
 58. M. Salinas, et al, "Implementation-Independent Model of an Instruction Set Architecture in VHDL", *IEEE Design & Test of Computers*, September, 1993, pp 42-54.
 59. F. Boeri and M. Auguin, "OPSILA: A Vector and Parallel Processor", *IEEE Transactions on Computers*, Vol. 42, No. 1, January, 1993, pp 76-82.
 60. R. Gupta and G. Micheli, "Hardware-Software Cosynthesis for Digital Systems", *IEEE Design & Test of Computers*, September, 1993, pp 29-40.
 61. E. Aas, et al, "Quantifying Design quality Through Design Experiments", *IEEE Design & Test of Computers*, Spring, 1994, pp 27-37.
 62. D. Thomas, et al, "A Model and Methodology for Hardware-Software Codesign", *IEEE Design & Test of Computers*, September, 1993, pp 6-15.

63. "M68H16 Family CPU16: Central Processing Unit Reference Manual", Motorola Inc., 1991.
64. J. Handy, "The Cache Memory Handbook", Academic Press, Inc., San Diego, 1993.
65. H. J. Mitchell, "32-Bit Microprocessors", Blackwell Scientific Publications, London, 1991.
66. M. S. Lucas, et al, "The Impact of the Personal Computer on Instrumentation and Measurements", *IEEE Instrument and Measurement Conference*, 1986, pp 231-235.
67. W. R. Hill, "Low Cost Microprocessor Based Wattmeter", *IEEE Instrument and Measurement Conference*, 1986, pp 243-245.
68. A. Kareem and S. Balakrishnan, "Applications of a Digital Signal Processor in Test Instruments", *IEEE Instrument and Measurement Conference*, 1988, pp 405-409.
69. A. Shah, et al, "PC Based Smart Interface System", *IEEE Instrument and Measurement Conference*, 1986, pp 246-249.
70. G. Wale and D. Rustici, "Firmware for an Intelligent Digital Multimeter", *IEEE Instrument and Measurement Conference*, 1988, pp 322-328.

END

0 | 4 - 0 6 - 9 | 6

FIN