1

National Library
of Canada

National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche 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.

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

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.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

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

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

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.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

THÈSES CANADIENNES
SUR MICROFICHE

3-315-22190-9

NAME OF AUTHOR/NOM DE L'AUTEUR CONRAD MICHAEL KARAM
TITLE OF THESIS/TITRE DE LA THÈSE THE DESIGN AND EVALUATION OF
A COMMON BUS MEMORY ACCESS
PROCESSOR
UNIVERSITY/UNIVERSITÉ CARLETON UNIVERSITY, OTTAWA, CANADA
DEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE M. Eng.
YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE GRADE 1984
NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE B. A. BOWEN

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies
of the film.

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la
thèse ni de longs extraits de celle-ci ne doivent être imprimés
ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/DATE APRIL 11/84 SIGNED/SIGNÉ Conrad Karam

PERMANENT ADDRESS/RÉSIDENCE FIXE 2228 HILLARY AVENUE
OTTAWA, ONTARIO, CANADA
K1H 7J1

THE DESIGN AND EVALUATION OF A COMMON BUS
MEMORY ACCESS PROCESSOR

by

© Gerald Michael Karam, B.A.Sc.(EE)

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 System and Computer Engineering
Carleton University
Ottawa, Ontario

March 1984

LICENCE TO CARLETON UNIVERSITY

In the interests of facilitating research by others at this institution and elsewhere, I hereby grant a licence to:

CARLETON UNIVERSITY

to make copies of my thesis:

Thesis on the development of a
system for the automatic processing

or substantial parts thereof, the copyright which is invested in me, provided that the licence is subject to the following conditions:

Only singled copies shall be made or authorized to be made at any one time, and only in response to a written request from the library of any university or similar institution on its own behalf or on behalf of one of its users.

This licence shall continue for the full term of the copyright, or for so long as may be legally permitted.

The Universal Copyright Notice shall appear on the title page of all copies made under the authority of this licence.

This licence does not permit the sale of authorized copies at a profit, but does permit the collection by the institution or institutions concerned of charges covering actual costs.

All copies under the authority of this licence shall bear a statement to the effect that the copy in question "is being made available in this form by the authority of the copyright owner solely for the purpose of private study and research and may not be copied or reproduced except as permitted by the copyright laws without written authority from the copyright owner."

The foregoing shall in no way preclude the granting by the author of a licence to the National Library of Canada to reproduce the thesis and to lend or to sell copies of the same. For this purpose it shall also be permissible for Carleton University to submit the abovementioned thesis to the National Library of Canada.

Signature of Witness

APR 11/66

Date

M. F. May

Degree

Small Town

Signature of Student

SYSTEMS & COMPUTER ENG.

Department of Student

The undersigned recommends to the Faculty of Graduate Studies and Research acceptance of the thesis

"The Design and Evaluation of a Common Bus Memory Access Processor"

submitted by Gerald Michael Karam, B.A.Sc.(EE), in partial fulfillment of the requirements for the degree of Master of Engineering.

Thesis Supervisor
Chairman,
Department of Systems and
Computer Engineering

Carleton University

April 1984

ABSTRACT

In this thesis we present the design and evaluation of a common bus memory access processor. The design phase entails: (1) the organization of the memory access processor-based common bus multiprocessor architecture and (2) a detailed design of the memory access processor. In the evaluation phase we show: (1) the realizability of the memory access processor in terms of a proposed single chip VLSI implementation and (2) that the memory access processor-based common bus multiprocessor does not exhibit performance losses when compared (using signal processing applications) to an equivalent conventional common bus multiprocessor architecture.

ACKNOWLEDGEMENTS

This is a long list for there are so many who contributed. First on the list are two men to whom I owe as much thanks for my philosophy on research as I do for the invaluable guidance I received in the many phases of this thesis. I speak, of course, about my co-supervisors, Drs. Moshe Krieger and Archie Bowen.

Secondly, I would like to thank Drs. Ray Buhr and Murray Woodside for their highly valued discussions and suggestions.

I would also like to take this opportunity to thank the other faculty, my colleagues and the support staff, all of whom have contributed both to my work and to the enjoyment of my stay.

My family and friends, I thank you for your constant and unrelenting support, not to mention for listening to my complaints about how much work I always had to do.

Finally, I thank my Annie, who as always, came through in the crunch, and with a smile.

I dedicate this thesis to the memory of my father.

CONTENTS

ABSTRACT		iii
ACKNOWLEDGEMENTS		iv
 <u>Chapter</u>		 <u>page</u>
I. INTRODUCTION		1
II. SPECIAL PURPOSE MULTIPROCESSOR/MULTIPROCESS ARCHITECTURAL CHARACTERISTICS FOR SIMPLIFIED DESIGN		6
Simplified Software Design		6
Simplified Hardware Design		8
General Remarks		9
III. OVERVIEW OF THE MEMORY ACCESS UNIT (MAU) BASED MULTIPROCESSOR ARCHITECTURE		11
Data Flow Description		11
High Level Hardware Organization		15
IV. MEMORY ACCESS UNIT (MAU) DESCRIPTION		20
High Level Data Flow Description		21
Functional Modules (f-modules)		22
Address Modification ALU		22
Address Register Module		23
PE FIFO Queue Module		23
Process Management Module		24
PE Data Transfer Module		24
PE Status Module		24
Control Modules (c-modules)		25
MAU Module		25
INF Module		25
PES Module		26
Specialized Execution Features		30
Process Management		30
Switching PE Types		33
Interrupt Handling		34
Multiple Instruction Stream Interleaving		35
Handling the Procedural Dependence of Instructions		36
Control Aspects		37
Functional Block Description		39

V.	PROPOSED SINGLE CHIP (VLSI) IMPLEMENTATION OF THE MEMORY ACCESS UNIT (MAU)	44
	Instruction Set Definition	44
	General Organization	44
	Addressing Modes	46
	MAU Instructions	48
	PE Instructions	49
	Design of Functional Blocks	51
	Address Registers and PE Status Registers	51
	Address Modification ALU	53
	PE FIFO Queue Block	57
	Process Management Block	60
	PE Data Transfer Block	62
	Design of Resource Arbiters	63
	Chip Pinouts	65
	MAU Operations: Execution Time and Control	
	Store Estimates	67
	Execution Time Estimates	67
	Control Store Estimates	68
	Summary of Design Estimates	69
VI.	PERFORMANCE EVALUATION OF THE MAU-BASED MULTIPROCESSOR ARCHITECTURE	71
	Multiprocessor Performance Models	72
	Signal Processing Application Model	77
	The Derivation of the Apparent Control	
	Time Function (γ_c)	78
	Simulation of an MAU-based Multiprocessor	
	Machine	79
	Complex FFT Performance Comparison	84
VII.	CONCLUSIONS	86
	REFERENCES	89

	<u>Appendix</u>	<u>page</u>
A.	GRAPHICAL DESIGN NOTATION BY BUHR [BUHR84]	93
B.	DETAILED INSTRUCTION DESCRIPTIONS	95
C.	EXAMPLE MAU OPERATION EXECUTION TIME AND CONTROL STORE ESTIMATE CALCULATION	99
D.	MAU OPERATIONS: EXECUTION TIME AND CONTROL STORE ESTIMATES	100

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
1. MAU-based Multiprocessor Architecture: Data Flow Diagram	12
2. MAU-based Multiprocessor Architecture: High Level Hardware Configuration	17
3. MAU-based Multiprocessor Architecture: Structure Diagram	18
4. MAU High Level Data Flow Diagram	22
5. MAU Run-time Process State Graph	32
6. MAU Structure Graph	38
7. MAU Resource Request Graph	39
8. MAU Functional Block Diagram	40
9. PE FIFO Queue Block: Detailed Functional Block Diagram	41
10. Process Management Block: Detailed Functional Block Diagram	43
11. MAU-based Multiprocessor: General Instruction Format	45
12. Multi-Operand Instruction Example: "MAU ADD"	47
13. Logical-to-Physical Address Transformation	48
14. Basic Register Cell and Controller	52
15. Address Modification ALU High Level Organization	55
16. Two Bit Slice of Arithmetic-Logic Subunit (bits 15 and 16)	56
17. FIFO Queue Controller and 1-Bit, 3-Level FIFO Queue Example	58

18.	High Level Structure of the PE Data Transfer Block	62
19.	Two-Input, Master Request/Acknowledge Asynchronous Arbiter Organization	64
20.	Performance Comparison Hardware Configurations . . .	72
21.	Performance Comparison Models	74
22.	Structure Diagram of MAU simulation (Tasks and Monitors only)	80
23.	Graphs of γ_c versus K Comparing Theoretical and Simulation γ_c values (T_d fixed, family in N) . .	83
24.	Complex FFT Performance Comparison Bus Constraint Graphs	85

LIST OF TABLES

<u>Table</u>	<u>page</u>
1. MAU C-Module Psuedo-Ada Execution Flow Description	27
2. INF C-Module Psuedo-Ada Execution Flow Description	28
3. PES C-Module Psuedo-Ada Execution Flow Description	29
4. MAU-based Multiprocessor Instruction Set Addressing Modes	47
5. MAU Instruction Descriptions (Concise)	49
6. PE Generic Intruction Formats (Concise)	50
7. Design Estimates for Address Register Block and PE Status Block	53
8. Design Estimates of Address Modification ALU	57
9. Design Estimates for PE FIFO Queue Block	59
10. Design Estimates of Process Management Block	61
11. Design Estimates of the PE Data Transfer Block	63
12. Summary of MAU Pin Count	66

Chapter I

INTRODUCTION

Special purpose multiprocessor/multiprocess computing systems have gained wide popularity. This is part of a broader trend towards a "multiprocessor design philosophy" that resulted primarily from the advent of inexpensive and powerful (V)LSI processing devices [DURNI79] [FAGGI78]. However, the high degree of circuit integration that is the strength of these devices can often lead to complex and challenging design. Kung [KUNGH82] points out that the cost-effectiveness of special purpose systems is always in question; that is, their cost must be sufficiently low to offset their restricted applicability and (frequently) small production runs. Total cost is divided into design and parts costs. Design costs, usually the larger of the two, can be chiefly attributed to the cost of manpower. It is therefore desirable, if not necessary, to simplify the design of such systems in order to reduce design costs.

Recently, the movement towards simplified special purpose system design has gained momentum. This is evidenced by: (1) efforts to simplify hardware design through the use of modular hardware [KUNGH82], (2) special purpose system design methodologies [FATHI83a] [FATHI83b]

[BUHR84] [BUHR82] [BOWEN80], (3) special purpose realtime operating systems [INTEL80] [KAHN78] [LINDE80], and (4) powerful concurrent high level languages [BRINC75] [USDOD82] [WIRTH77] [LAMPS77]. The development of a multiprocessor architecture that can form the basis for simplifying special purpose multiprocessor/multiprocess system design is the motivating notion in the work that follows.

In this thesis, we describe the design and evaluation of a common bus memory access processor. In the design phase of this research we present: (1) the organization of the memory access processor-based multiprocessor architecture and (2) a detailed specification of the memory access processor. In the evaluation phase we show: (1) the realizability of the memory access processor through the use of a proposed single chip VLSI implementation and (2) that the memory access processor-based multiprocessor does not exhibit any loss of performance when compared (using signal processing applications) to an equivalent conventional common bus multiprocessor machine.

The memory access processor is an extended version of that found in the decoupled access/execute architecture [COHLE81] [SMITH82] [PLESZ83]. In a decoupled access/execute architecture, there is a partitioning of functions between a memory access processor and the execute processors, or "processing elements". The functional partitioning involves isolating all memory access and

program control responsibilities in the memory access processor and all information processing in the processing elements. The memory access processor maintains internal queues (such as operand address queues), address registers and address modification circuitry. Processing elements can be functionally heterogeneous, however, they share two common attributes: (1) local memory for data storage, and (2) generally executing high level macro-instructions (eg. "complex FFT").

In our first extension of the decoupled access/execute concept we assign individual processes to each of the processing elements. Previous work only indicated one process being served by one or more processing elements. This modification introduces the problem of multiple instruction stream management that we resolve by changes to the memory access processor's internal organization.

We further modify the decoupled access/execute architecture by implementing hardware multiprocess operating system (O/S) kernel features in the memory access processor. Two O/S features are included: hardware semaphore management, and the support of dynamic, transparent, process-to-processor allocation (ie. process management). Previous studies did not consider the memory access processor as a tool for multiprocess management.

In the remainder of the thesis, the memory access processor will be referred to as the Memory Access Unit, or MAU.

In the next chapter, we outline our motivating philosophy of simplified special purpose multiprocessor/multiprocess system design. It is expressed in terms of architectural characteristics that provide a simple design milieu. Both hardware and software design aspects are considered.

In chapter III, a high level description of the memory access unit (MAU)-based multiprocessor architecture is presented. This chapter will provide insight to the MAU by studying it from the perspective of its operating environment.

Chapter IV encompasses the detailed design of the MAU. It is in this chapter that the specific details of the extensions to the decoupled access/execute architecture's memory access processor are provided. The high level description, specialized execution features, control mechanisms and functional hardware organization are discussed.

Chapter V addresses a proposed single chip VLSI implementation of the MAU. The purpose of this chapter is twofold: (1) it demonstrates the realizability of the MAU and (2) it provides the MAU operation execution times required for the performance evaluation in chapter VI. The proposed implementation is presented in three sections: (1) a description of the instruction set to be used, (2) the design of circuits to support: (a) the instruction set, and (b) the high level MAU design requirements as specified

in chapter IV, and (3) an outline of the execution times of the various MAU operations.

In chapter VI, a performance analysis of the MAU-based multiprocessor machine using signal processing applications is presented. A comparison is made to an equivalent conventional common bus multiprocessor architecture. Execution time information is drawn from: (1) in the case of the MAU, the proposed single chip implementation, and (2) in the case of the processing elements, typical algorithmic execution times.

Chapter II

SPECIAL PURPOSE MULTIPROCESSOR/MULTIPROCESS ARCHITECTURAL CHARACTERISTICS FOR SIMPLIFIED DESIGN

In this chapter we present the motivating philosophy behind this thesis: the simplified design of special purpose multiprocessor/multiprocess machines. This concept will be expressed in terms of a set of architectural characteristics that lead to a simple design environment.

Simplified system design can result from following two paths: simplified hardware design, and simplified software design. The demarcation line between the two generally places processing components in hardware, and operating systems (O/S) and programming languages in software. Initially we shall present the ideas of simplified design from this point of view. However, we do not ultimately constrain ourselves to this historic dividing line.

2.1 SIMPLIFIED SOFTWARE DESIGN

The techniques of structured (a word almost synonymous with simplified) software design are widely documented [DAHL72] [KOPET79] [WIRTH71] [WIRTH74]. However, these techniques often describe the procedures of software development while assuming the presence of an O/S and a

suitable programming language. We approach the idea of simplified software design from the opposite direction; that is, given a class of typical applications, postulate a set of O/S and programming language features that enable applications software design to be simplified.

A number of fundamental features of special purpose multiprocessor/multiprocess O/S kernels are outlined in [BOWEN80]. General purpose high level programming languages that directly support structured design in a multiprocess, multiprocessor environment are well-known [USDOD82] [BRINC75] [WIRTH77] [LAMPS77]. Furthermore, special purpose languages for multiprocessor environments have also been investigated. In [PERRO79] and [HIBBA78] two such languages were proposed. In addition, the authors of these publications outlined desirable language characteristics for multiprocessor/multiprocess architectures.

In examination of the above literature, we summarize what we consider to be the essential O/S and programming language requirements to support simplified multiprocessor/multiprocess software design:

O/S:

- semaphore wait and signal primitives
- dynamic, transparent, process-to-processor allocation

Programming Language:

- block structures
- explicit declaration of concurrent activities

- explicit interprocess communications

It is highly desirable that many of these software features be implemented directly in hardware. This would increase speed, reduce the amount of system software, and enhance overall reliability. Integrated circuit manufacturers have already developed processors that include O/S kernel features implemented in hardware [WOLOC81]. The most notable processor in this class is the INTEL iAPX 432 [GUPTA83].

Hardware embedded programming languages, more commonly known as directly executable high level programming languages, are for the most part, in the development stages [CHU81]. However, some specialized machines have been built. For example, a processor that directly executes LISP exists [SYMB082]. One major criticism of directly executable programming languages is that they often limit the usefulness of the accompanying hardware system to the particular features of the language being used [LIPOV78].

2.2 SIMPLIFIED HARDWARE DESIGN

The major keyword in simplified hardware design is "modularity". Defining hardware components in terms of modules enables recursive and therefore simplified hardware design [KUNGH82] [KUNGS82]. Reduced development, debug, and repair times, and generally enhanced expandability, are the

usual advantages of hardware modularity. As a bonus, the allocation of specific software functions to particular hardware modules becomes appealing. Done with care, this enables the same software configuration to be used for different end-user systems [FATHI83a]. Furthermore, one could develop a group of library software modules that take on different characteristics depending on the hardware module used for execution. This is a run-time analogy to the Ada generic [USDOD82].

A given design can be partitioned in many very different (and often highly conflicting) ways, and still be considered modular. This stems from ambiguity in the implied meaning of modularity. The factors that characterize a design as modular are dependant on the context in which modularity is applied [BUHR84]. As evidenced by [KUNGH82] [KUNGS84] [FATHI83a] [FATHI83b], the context of "functional" modularity seems to be the most appropriate choice for simplified hardware design.

2.3 GENERAL REMARKS

In the remainder of this thesis, we present a multiprocessor architecture whose characteristics reflect our philosophy of simplified special purpose multiprocessor/multiprocess system design; that is, special purpose multiprocessor/multiprocess system design can be simplified if this architecture is used as a basis for

design. The particular features that we identify in this architecture are: (1) functional hardware modularity, (2) hardware embedded semaphore wait and signal primitives and (3) hardware embedded process management.

Chapter III

OVERVIEW OF THE MEMORY ACCESS UNIT (MAU) BASED MULTIPROCESSOR ARCHITECTURE

In this chapter we give a functional description of the MAU-based common bus multiprocessor architecture. The primary components of the architecture are: (1) a group of processing elements ("PEs"), (2) the memory access unit ("MAU"), (3) a data/program memory and (4) a common bus interconnecting the PEs and the MAU. The MAU has sole access to the memory.

3.1 DATA FLOW DESCRIPTION

Intermodule information (data/control) flow is illustrated in Figure 1. Data flow between the memory and the MAU consists of instructions and program data. Opcodes, operands, results and PE status are the data flows between the MAU and the PEs. The control flow between the PEs and the MAU is composed of three items: (1) PE commands that request service from the MAU, (2) PE identification codes (PE IDCs), and (3) a "sleep" command from the MAU to any of the PEs.

The instruction format is an opcode followed by addressing information. The opcode itself does not imply addressing modes. The opcodes can be divided into two groups:

Figure 1: MAU-based Multiprocessor Architecture: Data Flow Diagram

1. those governing program control and all facets of addressing, and,
2. those that perform information transformation (processing).

The transformation opcodes are generic; that is, the same opcode may imply different internal operations on heterogeneous PEs. This instruction set can be described as orthogonal since it separates opcode and addressing information [GUTER83], and it uses different opcodes for addressing and processing operations [VIEHM83]. (Further specification of the instruction set is unimportant for the present discussion. The detailed description of the

instruction set is provided in chapter V as part of the proposed single chip MAU implementation.)

Instructions enter the processing system through the MAU. The MAU interprets instructions according to opcode group. Instructions that use opcodes from the addressing/program control group are executed in the MAU. Instructions used for information processing are pre-processed as follows: (1) the opcodes are queued for execution by a PE, and (2) addressing information is used to generate physical addresses of the operands and results; these addresses are also queued. The exception to this treatment of addressing information is PE register addressing. In this situation, the addressing information is appended to the opcode. In general, PE operands and results are pre-fetched and post-written, respectively. Each PE has its own instruction stream and data streams.

The PEs acquire their execution duties from the MAU. They request opcodes and operands, and return results and status, without concern for the details of program flow. The operands and results are either references to PE local registers, or data supplied by, or returned to, the MAU. PE registers may be of any complexity - from scalars to arrays. PE status information consists of a set of flags typical of those that are associated with processor (particularly arithmetic) functions.

There are four PE-to-MAU service requests: (1) a request for the next opcode, (2) a request for accompanying operands, (3) a request to return results from the previously executed opcode, and (4) a request to return the PE status that resulted from the execution of the previous opcode. PE identification codes (PE IDCs) are used to enable the MAU to distinguish individual PEs. The PE IDC accompanies each service request command. The MAU-to-PE "sleep" command is used to temporarily suspend the operation of a PE that cannot have its opcode request satisfied. When an opcode becomes available, the PE is "awakened" by an awaken signal from the MAU.

The MAU-based architecture partitions instruction processing in two ways. The first partitioning delineates processing and nonprocessing-oriented instructions. The second isolates phases of instruction execution (for PE bound instructions). This approach to information processing has been treated in a variety of ways [INTEL82] [COHLE81] [BOWRA76] [ANDER67] [SMITH82] [PLESZ83]. The simplest involves pipelining the phases of instruction execution [INTEL82]. The other extreme is completely decoupled "access/execute" [SMITH82] [PLESZ83]. The MAU-based multiprocessor architecture is an extension of the latter. The significant difference between our use of this partitioning and the prior work lies in the support of individual processes on each PE. Using the classification

criteria of Flynn [FLYNN66], the MAU-based machine is described as MIMD, as opposed to the SISD and SIMD machines indicated in the previous studies. Using a more recent architectural classification, the MAU-based multiprocessor machine would be described as a Multiple Task-Multiple Data (MTMD) machine [FATHI83a] (because the PE high level macro-instructions can be considered to be "jobs" or "tasks" that PEs are requested to perform).

The partitioning of instruction processing has two distinct characteristics that make it an attractive concept in the context of the notions outlined in chapter II. The first characteristic is (functional) hardware modularity. This results from partitioning instruction types and phases, and the ability to support easily, functionally heterogeneous PEs. The second characteristic is the complete centralization of all aspects of program control. This will facilitate the implementation of hardware multiprocess O/S kernel features.

3.2 HIGH LEVEL HARDWARE ORGANIZATION

A suitable hardware configuration of the MAU-based machine is illustrated in Figure 2. A structure graph showing module interactions and data flow is illustrated in Figure 3. The control lines in Figure 2 translate to module calls in Figure 3, and data lines in Figure 2 translate to data and control line in Figure 3. The system structure

graph is expressed in terms of Ada program structures using a graphical notation defined by Buhr [BUHR84]. A summary of the notation meanings is provided in the Appendix A. It is worthwhile to note that although the hardware structure has been expressed in terms of the Ada programming language, this has no bearing on the type of software system to be supported by the hardware. Rather, the Ada-based structure graph is being used as a vehicle for illustrating the interaction aspects of the hardware modules. How the Ada modules are translated into hardware is unimportant for the discussion. (The concept of using Ada program structures to describe and implement hardware components is gaining attention [ORGAN82]. As an example, one could imagine mapping an Ada task into a micro-code engine for a processor.) In the structure graph, the PEs, MAU and bus arbiter are represented as active packages; the memory is considered to be a passive package. The bus arbiter package handles request from PEs to gain access to the MAU.

PEs interact with the MAU in the following sequence: (1) request access to the MAU through the bus arbiter, (2) the bus arbiter indicates to the MAU that PE bus service is about to begin; this is done by calling "initiate service", (2)b, the bus arbiter grants service to the PE (by allowing it to "return from its call"), (3) PEs make service requests to the MAU through the service request entry, and (4) complete the use of the MAU by making an "end service" call on the MAU and a "bus free" call on the bus arbiter. A PE may make

Figure 2: MAU-based Multiprocessor Architecture: High Level Hardware Configuration

several requests to the MAU before it relinquishes the bus. A typical series of PE requests would be: (1) return result of previous operation, (2) return the status of the previous operation, (3) get the next opcode, and (4) get the next corresponding operand. If an opcode is not available from the MAU, then the MAU returns a sleep command after which the PE releases the bus and puts itself to "sleep" by waiting for an awaken call (from the MAU). The PE is later awakened (when an opcode becomes available) through a call by the MAU on the PE awaken procedure. All interrupts are

Figure 3: MAU-based Multiprocessor Architecture: Structure Diagram

made as calls to the MAU (since the MAU has sole jurisdiction over program control).

* * *

In the next chapter, the MAU is described in detail. The implementation of the extensions to the decoupled access/execute architecture are indicated. We do not delve further into the design of a PE since a PE is primarily an

arithmetic computing module that obeys a well-defined interaction protocol.

Chapter IV

MEMORY ACCESS UNIT (MAU) DESCRIPTION

In this chapter, a description of the organization and operation of the MAU is given. It is in the MAU that the extensions to the decoupled access/execute architecture are implemented; specifically these are: (1) the support of MIMD processing through multiple instruction stream management, (2) hardware semaphore management and (3) dynamic, transparent, process-to-processor assignment (process management).

The problem of managing multiple instruction streams is resolved through the use of: (1) replicated hardware units, (2) concurrently executing internal sequence controllers and (3) interleaving instruction stream processing under a fair-sharing algorithm. The replicated hardware eliminates the need to save and restore data when switching instruction stream processing target PEs. The use of concurrently executing internal sequence controllers will permit (1) maximum utilization of the MAU's internal resources and (2) provide maximum service to the PEs.

The hardware semaphore and process management are implemented through the use of specific hardware components and a control algorithm.

The details of all extensions will be provided in the appropriate sections of the MAU description.

The presentation proceeds in four steps: (1) a high level description of the internal structure of the MAU, (2) a discussion on specialized execution aspects, (3) an outline of control aspects and (4) a brief description of the functional hardware blocks.

The description of the MAU is expressed in terms of the high level requirements of a proposed single chip VLSI implementation (the detailed circuit designs are given in chapter V). The design objectives are: (1) support for 32-bit data words, (2) a 1 Mbyte physical address space, and (3) control support for eight PEs.

4.1 HIGH LEVEL DATA FLOW DESCRIPTION

A high level data flow diagram of the MAU internal architecture is shown in Figure 4. We identify six functional modules (f-modules) and three controller modules (c-modules). The c-modules can be considered "active" (or controlling) elements of the MAU and the f-modules can be regarded as "passive" (or controlled) elements. The six f-modules perform data processing under the direction of the c-modules. The f-modules are: the address modification ALU, the address register module, the PE FIFO queue module, the PE data transfer module, and the PE status module and the process management module. The c-modules represent three concurrent activities: instruction pre-fetching (INF

calculations, opcode-directed functions (eg. increment register) and logical address calculation (where applicable).

4.1.1.2 Address Register Module

The address register module consists of a group of registers used for memory address. These include: 6-general purpose address, 1-stack pointer, 1-program counter, 1-data segment base, and 1-program segment base. The 1 Mbyte physical address space is segmented, where the segment base is 16-bits, the segment offset is 16-bits, and the segment size varies from 16 bytes to 64 Kbytes.

4.1.1.3 PE FIFO Queue Module

The PE FIFO queue module contains the queueing structure that supports the decoupling of the execute and access phases of PE instructions. In the most general case of decoupled access/execute, six FIFO queues are required [SMITH82] [PLESZ83]. These queues are identified to be: the instruction queue, the PE opcode queue, the PE operand (physical) address queue, the PE result (physical) address queue, the PE operand queue, and the PE result queue. However, since we have considered signal processing as the target application, (where the majority of operands and results are vectors), we adopt a simpler version of the decoupled access/execute architecture. The simplification

involves eliminating operand pre-fetching and result post-writing. Instead, all fetching and storing is performed as soon as required. As a result, the operand and result queues are not necessary. The physical address queues are still required because although the data transfer can be delayed until required, the physical address computations must be performed as the PE instructions are pre-processed.

4.1.1.4 Process Management Module

The process management module supports process storage (in terms of a process control block or PCB) and process manipulation. Specifically, this involves ready-to-run and semaphore wait queue management, and interrupt handling.

4.1.1.5 PE Data Transfer Module

The PE data transfer module is a DMA-styled module for transferring PE operands and results between the PEs and the memory. The physical address of PE operands or results, the number of words to transfer, and the word size are passed to this module when data transfers are requested by PEs.

4.1.1.6 PE Status Module

The PE Status module is a set of 32-bit registers, one dedicated to each PE. Each register stores the most recent copy of a PE status word.

4.1.2 Control Modules (c-modules)

4.1.2.1 MAU Module

The MAU module performs the majority of the MAU coordination. It is responsible for process management, interrupts, awakening "sleeping" PEs, selecting target PEs for instruction stream processing, and instruction stream processing itself. Target PE instruction stream processing includes: decoding instructions, executing address/program control instructions, calculating and queueing PE operand and result physical addresses, and queueing PE opcodes. A psuedo-Ada execution flow description for the MAU module is given in Table 1. Note that the control modules are regarded as Ada tasks for the sake of this description (and the descriptions to follow in Tables 2 and 3). This mapping is further described in section 4.3 and Figure 6.

4.1.2.2 INF Module

The INF module pre-fetches instructions for the target PE selected by the MAU module. If the MAU module changes target PEs then it notifies the INF module. The INF module checks for a change in target PE at the beginning of every fetch cycle. Subsequent to receiving a notification of change in the target PE, the INF module acknowledges the MAU module's request. The psuedo-Ada execution flow description for the INF module is presented in Table 2.

4.1.2.3 PES Module

The PES module manages all interactions with the PES (with the exception of the PE awaken signal administered by the MAU module). These include: delivering opcodes and operands, storing results and PE status words, and signaling a PE to suspend itself pending an available opcode. A psuedo-Ada execution flow description for the PES module is given in Table 3.

TABLE 1

MAU C-Module Psuedo-Ada Execution Flow Description

```

loop
  CHECK FOR INTERRUPTS;
  if INTERRUPTED then
    SWITCH INSTRUCTION STREAM PROCESSING TO
    REQUIRED PE;
    if PROCESS RUNNING ON THAT PE then
      SAVE PROCESS AND PLACE IN RTR QUEUE;
    end if;
  else
    if A PE WITH A HIGHER PRIORITY then
      SWITCH INSTRUCTION STREAM PROCESSING TO
      THIS HIGHER PRIORITY PE;
      REQUEST INF MODULE TO SWITCH TARGET PE;
      CHECK (AND WAIT) FOR ACKNOWLEDGE FROM INF;
    else
      if NO PROCESS TO RUN then
        GET PROCESS;
      end if;
    end if;
  end if;
  GET NEXT OPCODE;
  if A PE OPCODE then
    QUEUE OPCODE;
    COMPUTE AND QUEUE PHYSICAL ADDRESSES OF OPERANDS
    AND RESULTS (AS APPROPRIATE);
  else
    if PE DEPENDENT PROGRAM CONTROL then
      STOP INSTRUCTION STREAM PROCESSING
      UNTIL PE IS SLEEPING;
      {the PE and MAU are thus at the same point
       in the instruction stream}
      EXECUTE INSTRUCTION;
    else
      EXECUTE INSTRUCTION;
    end if;
  end if;
end loop;

```


TABLE 2

INF C-Module Psuedo-Ada Execution Flow Description

```
loop
  CHECK FOR REQUEST TO SWITCH TARGET PE;
  if REQUESTED TO SWITCH TARGET PE then
    SWITCH TARGET PE;
    SEND ACKNOWLEDGE TO MAU C-MODULE;
  end if;
  if ROOM TO PUT MORE INSTRUCTIONS then
    FETCH INSTRUCTIONS;
    QUEUE INSTRUCTIONS IN INSTRUCTION QUEUE;
  end if;
end loop;
```

TABLE 3

PES C-Module Psuedo-Ada Execution Flow Description

```
loop
  accept INITIATE SERVICE do
    {set-up PE service}
  end accept;
  select
    accept SERVICE REQUEST do
      case SERVICE COMMAND of
        OP CODE REQUEST:
          if AVAILABLE then
            DEQUEUE OP CODE;
            RETURN OP CODE;
          else
            RETURN SLEEP COMMAND;
          end if;
        OPERAND REQUEST;
          DEQUEUE OPERAND ADDRESS;
          FETCH AND TRANSFER OPERANDS
            THROUGH PE DATA TRANSFER MODULE;
        RESULT RETURN REQUEST:
          DEQUEUE RESULT ADDRESS;
          TRANSFER AND STORE RESULTS
            THROUGH PE DATA TRANSFER MODULE;
        STATUS RETURN REQUEST:
          STORE STATUS;
      end case;
    end accept;
  or
    accept END SERVICE do
      {clean up}
    end accept;
  end select;
end loop;
```

4.2 SPECIALIZED EXECUTION FEATURES

Although most of the activities managed by the MAU are fairly conventional, there are several functions that require more detailed explanation in order to clarify the nature of their operations. Specifically, we identify process management (including semaphores), PE type switching, interrupt handling, multiple instruction stream interleaving and handling the procedural dependence of instructions.

4.2.1 Process Management

Dynamic, transparent, process-to-processor assignment is centred on the concept that a process is assigned to a PE on the basis of PE type (the identifier that distinguishes heterogeneous PEs) as opposed to a particular physical PE; that is, the application software references logical PEs to which the MAU assigns appropriate physical PEs. A processes' logical to physical PE assignment is fixed until it is removed from the physical PE (eg. by a wait on a semaphore). When the process is subsequently restored, it need only be assigned to a PE of the appropriate type, not necessarily to the same physical PE.

Process-to-processor assignment involves two phases of operation: (1) PE identification during MAU initialization and (2) run-time process suspension and allocation. These phases of operation will now be discussed.

7

Before a PE can be assigned a process, it must be both "hardware active" and "software identified" to the MAU. This is performed during MAU initialization. A PE is hardware active if its "activity line" is connected to the MAU activity inputs. The MAU senses these inputs to detect the presence of an active PE. Software identification is initiated once a PE is hardware active. During software identification, the MAU generates and delivers a "query" opcode to the active PE. After processing this opcode, the PE returns a "result" that identifies its PE type and the required size of the save area for the PE's internal state. The PE type uniquely identifies heterogeneous PEs. The size of the save area for a PE's internal state will indicate how many words are to be transferred between a PE and the memory during a complete save or restore of a PE's internal registers. Once all the active PEs have been identified, the MAU switches to the run-time phase of operation.

A state graph of run-time process management is shown in Figure 5. Lists of processes to be executed are maintained in the memory. There is a list for each PE type that is required by the applications program. The MAU searches these lists for processes to assign to PEs. After a process is assigned to a PE, it will remain assigned until it is suspended on a semaphore, terminated, interrupted, or explicitly switched to another PE of a different type. If a PE becomes available for any of these reasons, then the MAU

A time-sliced sharing of a single PE could be implemented by forcing (by external clock) a selection of processes from the process list and an internal rotation of processes through the PE RTR queues. The PE's internal state could be saved and restored as required by explicit requests from the MAU. However, the large amounts of PE internal data that may be necessary to save and restore, make time-slicing appear highly inefficient and undesirable.

Processes that are suspended on semaphores are queued on the appropriate semaphore wait queue. Semaphore suspension of a process does not result in an automatic save of a PE's internal state. This must be explicitly coded. Once the semaphore is signaled, the process at the head of the wait queue is moved to the tail of the RTR queue of the required PE type.

4.2.2 Switching PE Types

It is conceivable that it may be necessary to move a process from a PE of one type to a PE of another type. This can be accomplished by using a single explicit instruction. The operation involves capturing the process state in the Process Control Block RAM (PCB RAM), and then queueing the process in the RTR queue of the target PE type. This action is illustrated in Figure 5 by a double line.

4.2.3 Interrupt Handling

Interrupts and interrupt service routines (ISRs) are treated differently in the MAU-based architecture than they are in most conventional architectures. The major difference lies in the implementation of the ISRs. In the MAU-based machine, ISRs are considered to be complete processes, as opposed to the "simulated" procedure calls typical of most other systems. Each serviceable interrupt has a dedicated ISR process. During system initialization, each ISR process is started up on a PE. The ISR then "waits" on an interrupt dedicated semaphore immediately prior to the program statement that defines the beginning of interrupt processing. As a result, the ISR process PCB is stored in the PCB memory. When an interrupt occurs, the PCB of the process to be interrupted is queued at the head of the appropriate RTR queue. This action is illustrated in Figure 5 by a dot-dash line. A copy of the ISR process PCB is restored, thus allowing the interrupt to be serviced. The ISR process terminates at the end of interrupt processing, however, the original ISR process is still resident in the PCB RAM where it waits in anticipation of the next occurrence of the interrupt. The interrupted process' PCB is automatically restored since it is the first PCB on the RTR queue.

4.2.4 Multiple Instruction Stream Interleaving

Typically the MAU is managing instruction stream processing for several PEs simultaneously. However, it can only physically process them one-at-a-time. An analogous problem is multiprocessing on a single CPU. A frequent solution in this analogy is continually saving and restoring process states in a time-sliced sharing of the CPU. In the case of the MAU, we must save and restore the state of a PEs instruction stream processing and share instruction stream processing with other PEs in some equitable manner. We eliminate the need to save and restore the state of instruction stream processing by using replicated hardware.

In choosing a fair-sharing algorithm we consider optimal PE utilization as a desirable target. A suitable algorithm is described next:

- Each PE has a multilevel priority. A basic set of priority levels are: 0 - (highest) running process is an interrupt process, 1 - running process has PE idle, 2 - running process is processing first "j" instructions of it's processing "slice", PE is not idle and 3 - running process is processing more than first "j" instructions.
- Process instructions for a given PE until there is a PE of higher priority. In the case of multiple PEs at the same, higher priority, choose a PE on a round robin (circular priority) basis.

4.2.5 Handling the Procedural Dependence of Instructions

Instructions whose execution or precise actions depend on the outcome of a previous instruction, are called "procedurally dependent instructions" [TJADE70]. In the MAU-based architecture, examples of instructions that impose such dependencies are conditional and unconditional branches (including procedure calls), and the wait and signal semaphore primitives. In decoupled access/execute architectures, the dependencies between the access and execute phases of the instruction stream can cause significant problems; particularly since the phases are generally processed in different hardware units. Solutions that vary in complexity have been proposed [INTEL82] [COHLE81] [BOWRA76] [ANDER67] [SMITH82] [PLESZ83]. We adopt a simple approach: in the event of an access/execute dependency, the MAU ceases to process the instructions of the PE concerned until it and the PE have arrived at the same point in the instruction stream. This simplification is justified because many special purpose applications (eg. signal processing) have highly deterministic, execution independent instruction sequences.

4.3 CONTROL ASPECTS

The control organization of the MAU is illustrated in Figure 6 using an Ada structure graph. The three c-modules in Figure 4 map into three controller tasks. There are two types of control entities in the structure graph that do not appear in the original high level data flow diagram: (1) message mailbox buffer tasks and (2) resource arbiter active packages.

Buffer tasks have been introduced to provide a message mailbox facility for the MAU and INF tasks. This implies that the MAU and INF tasks will perform checks of events (eg. interrupts) at their convenience rather than by waiting for them to occur (as is required when "accepting a call" from an event originating task).

We add resource arbiter active packages to arbitrate controller access to six shared resources: the memory, the address registers/ALU, the instruction queues, the opcode queues, the operand address queues, and the result address queues. (The PE status registers are only accessed in a mutually exclusive way and thus do not need arbitration.) The resource arbiters are designed to support a two-level access priority. The MAU task has non-pre-emptive high priority access to all resources. All other tasks have equal, lower priority. The resource requests are organized to eliminate any possibility of deadlock; that is, no two tasks ever attempt to reserve the same set of resources

simultaneously, (where the number of resources in a set is at least 2). A resource request graph for the three tasks and the six shared resources is illustrated in Figure 7.

Figure 6: MAU Structure Graph

Figure 7: MAU Resource Request Graph

4.4 FUNCTIONAL BLOCK DESCRIPTION

The functional block diagram of the MAU is illustrated in Figure 8. The six functional modules from Figure 4 map directly into the six functional blocks. The connecting buses provide the paths required by the c-modules to move information between f-modules.

The specific implementation details of the functional blocks are summarized below.

1. Address Register Bank - eight sets of 10, 16-bit registers (one set for each PE).

Figure 8: MAU Functional Block Diagram

2. PE FIFO Queue Block - Instruction queue: 16-bit words, 6 deep; this corresponds to an average of three instructions, Other queues: 32-bit words, 6 deep. A complete set of FIFO Queues is included for each PE. This eliminates the need for saving queue data when the PE targeted for instruction stream processing is changed. A further detailed view of the PE FIFO queue block is shown in Figure 9.
3. Process Management Block - The process management block is shown in further detail in Figure 10. The process management block consists of four major

Notes: (1) control not indicated
 (2) "B" denotes tristate buffer

Figure 9: PE FIFO Queue Block: Detailed Functional Block Diagram

components: (1) the process control block (PCB) storage RAM, (2) the semaphore array, (3) the PCB pointer record array, and (4) the PCB ready-to-run (RTR) and semaphore wait queue pointers. A PCB is composed of the contents of the 10 address registers and the 32-bit PE status word of a given process. The PCB RAM is divided into 128 PCB storage areas (24 bytes each). The semaphore array contains 32 user-accessible semaphores: 16 for general purpose use and 16 for use with interrupts. The PCB pointer

record array is a collection of records that are used to implement the linked list structures of the RTR and semaphore wait queues. Each record contains a successor record pointer, a PCB pointer, and a PE type code. A stack of pointers to available PCB pointer records is maintained in order to provide the dynamic allocation (and de-allocation) of PCB pointer records. The RTR and semaphore queue pointers are used to access queue head and tail PCB pointer records. There are 16 wait pointers, one for each of the general purpose semaphores; the interrupt semaphores only permit one interrupt process (to be explained later) per semaphore and thus do not require a supporting queue structure. Instead, the interrupts have an interrupt pointer block for pointers to interrupt processes (to be detailed in section 4.4.3). There are eight RTR queue pointers, one for each of (potentially) eight heterogeneous PEs; for example, if 8 identical PEs were used then only one RTR queue pointer would be required.

* * *

In the next chapter, a discussion of a proposed single chip VLSI implementation of the MAU is given. The discussion proceeds in essentially three sections: (1) a presentation of the instruction set to be implemented, (2)

Figure 10: Process Management Block: Detailed Functional Block Diagram

detailed circuit designs of the MAU functional blocks and
(3) estimates of the execution times of MAU operations.

Chapter V

PROPOSED SINGLE CHIP (VLSI) IMPLEMENTATION OF THE MEMORY ACCESS UNIT (MAU)

In this chapter, a proposed single chip VLSI implementation is presented. The motivation behind this effort is twofold: (1) it demonstrates the realizability of the MAU and (2) it provides some insight into the execution times for MAU operations. To achieve these goals we proceed as follows: (1) define the instruction set to be implemented, (2) define the circuits that correspond to the functional blocks outlined in chapter IV, (3) determine the chip pin count and (4) based on the instruction set from (1) and the designs in (2), derive the execution time and control store estimates of the MAU operations.

5.1 INSTRUCTION SET DEFINITION

5.1.1 General Organization

The instruction set of the MAU-based multiprocessor machine is based on the "orthogonal instruction set" concept. Although a firm definition of this idea is still lacking, there are some generally accepted notions:

1. operation information (opcode) and addressing information are not encoded together [GUTER83] and,

2. identical operations for information processing and address processing have different opcodes [VIEHM83]. This concept is very attractive for the MAU-based architecture because of the architectures requirements to: (1) isolate addressing information from operation information and (2) execute addressing operations in a different processor than that used for information processing.

The general format of an instruction is illustrated in Figure 11. Each instruction consists of a one byte opcode plus zero, one or more bytes of addressing information (addressing codes and numerical values). This organization is derived from that used in the U.S. Department of Defense NEBULA machine instruction set [USDOD81].

Figure 11: MAU-based Multiprocessor: General Instruction Format

The opcodes can be categorized into two groups (as discussed in chapter III): (1) those corresponding to addressing and program control and (2) generic information processing opcodes. This partitioning effectively divides the instruction set into two respective groups: (1) MAU

instructions and (2) PE generic instructions. The instructions in each of these groups will be outlined in a later section.

5.1.2 Addressing Modes

The addressing modes of all data are determined by addressing codes (adcodes). Some addressing modes require literal values to completely specify the address. For example, the indexed addressing mode requires 3 bytes of address description: (1) the index register mode code (1 byte) and (2) the base address (a 2 byte literal). There must be an addressing description for each data to be accessed. The coding schemes for addressing modes are shown in Table 4. An example demonstrating the address coding of a multi-operand instruction is shown in Figure 12. This particular instruction is an MAU ADD instruction.

The physical memory addressing is performed using a segmented address space (as mentioned in chapter III). In segmented address mapping, a physical address is determined by adding the logical address (the segment offset) to the segment base. The physical address size is a function of the number of bits representing the segment base, the number of bits representing the logical address and the minimum segment size. For this particular application, we choose a 16-bit logical address, a 16-bit segment base, and a 16 byte minimum segment size. The physical address space is thus 1

TABLE 4

MAU-based Multiprocessor Instruction Set Addressing Modes

ADDRESSING MODE	CODING
Register Direct	0010RRRR
Register Indirect	11WWRRRR
Literal	001111WW [16-bit constant]
Absolute	10WW0000 [16-bit constant]
Scaled Indexed	01WW0RRR [16-bit constant]
Unscaled Indexed	11WW0RRR [16-bit constant]
Vector	000WWRRR [16-bit constant]

Notes: "R" - Register encoding bit
"W" - Wordsize encoding bit

Figure 12: Multi-Operand Instruction Example: "MAU ADD"

Mbyte. The logical-to-physical address transformation is illustrated in Figure 13. Two segment registers are used for address mapping: (1) a program segment register for all program references and (2) a data segment register for all data references. Isolating data and program areas is useful for preventing data writes from accidentally damaging

TABLE 5

MAU Instruction Descriptions (Concise)

<u>ONE OPERAND</u>		<u>SYSTEM/PROCESS CONTROL</u>	
INCR	Register	WAIT	Semaphore
DECR	Register	SIGNAL	Semaphore
CLR	Register	DEFNSEM	Semaphore, n
TEST	Register	DEFNINTR	Interrupt, Ptype
<u>TWO OPERAND</u>		SWINTR	Interrupt
		ENINTR	
		DISINTR	
ADD	Reg, AnyOperand	DEFNPROC	Ptype
SUB	Reg, AnyOperand	TERMPROC	
CMP	Reg, AnyOperand	SWITCHPE	Ptype
<u>MOVE</u>		<u>PROGRAM CONTROL</u>	
LOAD	Reg, AnyOperand	IBLT	Reg1, Reg2, Address
STORE	Reg, AnyOperand	DBGT	Reg1, Reg2, Address
PUSH	Register	JMP	Address
POP	Register	BBC	n, Address
		JSR	Address
		RTS	

5.1.4 PE Instructions

PE generic instructions provide a consistent software interface between heterogeneous PEs and the MAU. Each PE opcode can be interpreted uniquely by heterogeneous PEs. For example, a PE of type "A" may interpret the opcode 05H as a complex vector multiply, whereas a PE of type "B" may interpret the same opcode as a real scalar addition to a vector.

In order to simplify the PE FIFO queue operations, PE instructions are restricted in the following ways:

1. only one non-PE register reference per instruction and,
2. in two-operand instructions, one of the operands must be an implied result (where appropriate).

We can categorize the PE generic instructions into the same groups as the MAU instructions except that program control instructions do not exist. A summary of the PE generic instruction formats is presented in Table 6. Detailed descriptions of the PE instruction formats are found in Appendix B.

TABLE 6
PE Generic Instruction Formats (Concise)

<u>ONE OPERAND</u>	<u>SYSTEM CONTROL</u>
ALU1 Register	SYSTEM1 Register SYSTEM2
<u>TWO OPERAND</u>	<u>MOVE</u>
ALU2 Reg,AnyOperand	LOADPE Reg,AnyOperand STOREPE Reg,AnyOperand MOVEPE Reg1,Reg2 SAVEPE RESTORPE

5.2 DESIGN OF FUNCTIONAL BLOCKS

In this section the design of the functional blocks outlined in chapter IV is presented. The circuit designs are based on the use of NMOS transistors and a two phase clock. The designs are somewhat superficial because complete specification is beyond the scope of this thesis. However, there is enough detail to provide estimates of design magnitude and execution speed.

5.2.1 Address Registers and PE Status Registers

The one-cell dynamic R/W register can be designed using 6 NMOS transistors [MEAD80]. This requires a load signal to indicate read or write (load). The basic register cell and register controller are illustrated in Figure 14. Based on this cell configuration and the PE status and address register cell requirements in chapter IV, we estimate the total number of transistors and total number of control lines for these two blocks. The results are presented in Table 7.

Figure 14: Basic Register Cell and Controller

TABLE 7

Design Estimates for Address Register Block and PE Status Block

ADDRESS REGISTERS

Transistor:

7680 = 8 banks x 10 reg/bank x 16 bit/reg x 6tr/bit

1120 = 8 banks x 10 reg/bank x 14 tr/reg

8800 transistors

Control Lines:

1 - read

2 - write

4 - register select

3 - bank select

1 - clock

10 lines

PE STATUS REGISTERS

Transistors:

1536 = 8 banks x 1 reg/bank x 32 bits/reg x 6 tr/bit

112 = 8 banks x 1 reg/bank x 14 tr/reg

1648 transistors

Control lines:

1 - read

1 - write

3 - bank select

1 - clock

6 lines

5.2.2 Address Modification ALU

The address modification ALU is capable of performing six arithmetic-logic operations plus logical-to-physical address translation. The arithmetic-logic operations are:

addition, subtraction, comparison, increment, decrement and bit shift (for scaled indexed mode logical address calculations). The increment/decrement facility is ± 1 , ± 2 , and ± 4 . A high level diagram of the address modification ALU is shown in Figure 15. The selectors and shift register designs are based on those found in [MEAD80] and are not indicated. A 2-bit slice of the arithmetic-logic subunit is shown in Figure 16. The two bits shown, bits 15 and 16, isolate two cell types. Cell type 1 is a two input arithmetic-logic cell with carry/borrow (bit 15). Cell type 2 is a one input arithmetic-logic cell with carry/borrow (bit 16). The upper 4 bits accommodate the 20-bit arithmetic required for logical-to-physical address translation. A summary of control line and transistor estimates are presented in Table 8.

Figure 15: Address Modification ALU High Level Organization

Figure 16: Two Bit Slice of Arithmetic-Logic Subunit (bits 15 and 16)

TABLE 8

Design Estimates of Address Modification ALU

Transistors:

REGISTERS AND SELECTORS

140 = 20 bits x 7 tr/bit {input B}
 160 = 16 bits x 10 tr/bit {input A}
 120 = 20 bits x 6 tr/bit {output}

ARITHMETIC-LOGIC SUBUNIT

640 = 16 bits x 40 tr/celltype1
 80 = 4 bits x 20 tr/celltype2
 100 (approximately, for control)
 1240 transistors

Controls lines:

11 - control
 1 - zero flag
 12 lines

5.2.3 PE FIFO Queue Block

The basic FIFO queue is based on a floating input stage shift register stack; that is, the data input stage is determined by the location of the last full stage. As the stack is shifted (pushed or popped), the input stage changes accordingly. Diagrams of a 1-bit 3 level FIFO stack and stack controller are illustrated in Figure 17. Based on the FIFO queue requirements outlined in chapter IV and the circuits presented in Figure 17, we estimate the number of transistors and control lines in this block. These results are presented in Table 9.

FIFO Queue Controller

One Bit, 3-Level FIFO Queue

Figure 17: FIFO Queue Controller and 1-Bit, 3-Level FIFO Queue Example

TABLE 9

Design Estimates for PE FIFO Queue Block

Transistors:

{per PE}

INSTRUCTION QUEUE

$$780 = 6 \text{ words} \times (16 \text{ bit/words} \times 7 \text{ tr/bits} \\ + 1 \text{ float input ckt/word} \times \\ 18 \text{ tr/floating input ckt})$$

OPCODE QUEUE

$$1452 = 6 \text{ words} \times (32 \text{ bit/word} \times 7 \text{ tr/bit} \\ + 1 \text{ float input ckt/word} \times \\ 18 \text{ tr/floating input ckt})$$

OPERAND ADDRESS QUEUE

$$1452 = 6 \text{ words} \times (32 \text{ bit/word} \times 7 \text{ tr/bit} \\ + 1 \text{ float input ckt/word} \times \\ 18 \text{ tr/floating input ckt})$$

RESULT ADDRESS QUEUE

$$1452 = 6 \text{ words} \times (32 \text{ bit/word} \times 7 \text{ tr/bit} \\ + 1 \text{ float input ckt/word} \times \\ 18 \text{ tr/floating input ckt})$$

CONTROLLERS

$$\begin{array}{r} 128 = 32 \text{ tr/queue} \times 4 \text{ queues} \\ 5264 \text{ tr/PE} \\ \times 8 \text{ PEs} \\ \hline 42112 \text{ transistors} \end{array}$$

Control lines:

$$\begin{array}{r} 3 - \text{select/queue} \\ 1 - \text{pop/queue} \\ 1 - \text{push/queue} \\ 5 \text{ lines/queue} \\ \times 4 \text{ queues} \\ \hline 20 \text{ lines} \end{array}$$

5.2.4 Process Management Block

The process management block consists of registers, LIFO stacks and dynamic RAM. The design of such structures is discussed in [MEAD80]. We obtain estimates of the number of transistors and control lines based on the requirements outlined in chapter IV and the designs in [MEAD80]. These estimates are presented in Table 10.

TABLE 10

Design Estimates of Process Management Block

Transistors:

SUSPEND RECORD ARRAY

{4-bit PE type, 11-bit PCB addresses (ROM),
7-bit successor pointer per record}

8448 = 128 records x 11 bits x 6 tr/bit (RAM)

1408 = 128 records x 11 bits x 1 tr/bit (ROM)

SUSPEND RECORD PUSHDOWN STACK

7168 = 128 words x 7 bit/word x 8tr/bit

INTERRUPT POINTERS

{4-bit PE type, 7-bit pointer per record}

1056 = 16 interrupt/record x 11 bit/record x 6 tr/bit

PCB RAM

24576 = 128 processes x 12 word/process x
16 bit/word x 1 tr/bit

8000 = RAM overhead

QUEUE POINTERS

1008 = 24 pointers x 7 bits/pointer x 6 tr/bit

MISCELLANEOUS

2000 = temporary latches, controllers etc.
51000 transistors (approx.)

Control lines:

4 - wait queue select
3 - RTR queue select
3 - interrupt pointer select
1 - interrupt load
1 - stack push/pop
2 - suspend record load
3 - misc. latches
2 - PCB RAM read/write
5 - enables
24 lines

TABLE 11

Design Estimates of the PE Data Transfer Block

Transistors:

BI-DIRECTIONAL DATA LATCHES

640 = 2 latches x 32 bit/latch x 10 tr/bit
 100 = controllers

COUNTER, ADDRESS LATCHES

360 = 2 counter latches x 10 bit/latch x 6 tr/bit
 + 2 address latches x 20 bit/latch x 6 tr/bit

BUFFER

32 = 32 tr
 1672 transistors

Control lines:

6 - to MAU pointer
 4 - internal control
 {r/w, program PDT, clock}

5.3 DESIGN OF RESOURCE ARBITERS

In this section we consider the design of the resource arbiters. Although it might be expected that the design of the controller micro-code sequencers is also appropriate at this point, they will not be provided. Any specifications of the micro-code sequencers, beyond the control store estimates to appear in a later section, are outside of the scope of this thesis.

The resource arbiters can be implemented using a two-input asynchronous arbiter with master request/acknowledge specified in [PLUMM72]. Using this arbiter we provide request/acknowledge line-pairs to the PES and INF controllers and a master request/acknowledge to the MAU controller. The master request/acknowledge feature implies that all requests for resource access are propagated to the MAU controller for approval. Approval is granted by the MAU controller if it doesn't require the resource on it's next micro-code step. A diagram of this arbiter organization is shown in Figure 19. This arbiter requires approximately 150 transistors and no control lines other than the clock. The total number of transistors in all six arbiters would then be 900.

Figure 19: Two-Input, Master Request/Acknowledge Asynchronous Arbiter Organization

5.4 CHIP PINOUTS

The MAU chip pin count calculations can be derived using the data/control flows and control signals outlined in chapter III. Additional information is acquired from the PE data transfer block design (section 5.2.5) and the Process Management description (section 4.2.1).

A summary of the chip pin count is given in Table 12. In this table, the pinouts are categorized in three groups: (1) PE bus pins, (2) memory bus pins and (3) other pins (ie. other connections to the MAU). The total number of pins is 128.

TABLE 12
Summary of MAU Pin Count

PE BUS

- 32 - data
- 1 - sleep
- 1 - service request
- 1 - service acknowledge
- 1 - service complete
- 3 - PE service command
- 3 - PE identification code
- 1 - PE ready (for data transfer)
- 1 - PDT ready
- 1 - PDT request/acknowledge
- 1 - PDT request/acknowledge (for data transfer)
- 1 - PDT complete

MEMORY BUS

- 1 - read
- 1 - write
- 32 - data
- 20 - address

MAU to Bus Arbiter and PEs

- 3 - PE awaken select (encoded)
- 1 - awaken signal
- 1 - initiate service (from bus arbiter)
- 8 - PE activity lines
- 4 - interrupts
- 10 - miscellaneous

Total number of MAU pins = 128

5.5 MAU OPERATIONS: EXECUTION TIME AND CONTROL STORE ESTIMATES

5.5.1 Execution Time Estimates

MAU operations can be either control operations or instruction operations. Control operations are those that are concerned with executing MAU, PES and IFP controller functions. Instruction operations are those that handle the execution of MAU instructions and the pre-processing of PE instructions. (Although instruction operations are managed by the MAU controller, we regard them as separate operations in order to isolate the control overhead from the instruction execution time.)

Estimates of the execution times of these operations can be determined using the algorithm below.

1. List the sequential steps required to execute the operation.
2. In each step indicate the control line information needed to execute the step.
3. Specify concurrent control actions wherever possible.
4. For each step determine the number of unit clock cycles to execute it.
5. Sum the clock cycles for the operation.

In many cases the execution time is dependent on dynamic conditions and/or addressing modes. In these instances the only appropriate estimate is an execution time range.

The execution delays introduced by asynchronous events are not included in the execution time calculations.

Examples of such events are arbitrated resource requests and PE-MAU interactions.

An example of an execution time estimate is presented in Appendix C. The execution time estimates for all MAU operations are tabulated in Appendix D.

5.5.2 Control Store Estimates

The estimates of the control store required for MAU operations are derived from the MAU operation sequence steps (specified for the execution time estimates). An estimate is computed using the algorithm below.

1. Compute the number of control line-steps for each MAU resource required by the operation; eg. if the Address Registers/ALU resource (28 control lines) is needed for 5 steps of an operation, then the number of control line-steps is 140.
2. Sum the number of control line-steps for all resources.
3. Compute the control store on the basis of one bit per control line-step.

Many of the MAU operations involve only slight variations in the control requirements. For instance, the only difference between INCR R and DECR R is the ALU function select. For these cases the same micro-code can be shared by both operations where a parameter is used to indicate the appropriate processing for individual

operations. In fact, numerous operations are subdivided into collections of micro-code procedures. These procedures are then shared by several operations. For example, the procedure "save process control block (PCB)" is called by both WAIT and DEFNINTR.

An example control store estimate computation is presented in Appendix C. The control store estimates for the MAU operations are tabulated in Appendix D. The total number of transistors required for control store (from Appendix D) is 30,000.

5.6 SUMMARY OF DESIGN ESTIMATES

The design estimates for the proposed single chip VLSI implementation of the MAU are summarized as follows:

Number of transistors = 140,000

Control Store (bits) = 21,000

Pinouts = 128

MAU instructions = 27

PE generic instruction formats = 8

Since these design estimates are acceptable when contrasted with comparable devices, such as 32-bit microprocessors [GUPTA83], it can be concluded that the MAU is practically realizable.

* * *

In the next chapter, a performance comparison between the MAU-based multiprocessor machine and an equivalent conventional common bus multiprocessor architecture is presented. Signal processing is the application environment.

Chapter VI

PERFORMANCE EVALUATION OF THE MAU-BASED MULTIPROCESSOR ARCHITECTURE

In this chapter we present a comparative performance evaluation of the MAU-based multiprocessor architecture. A typical common bus multiprocessor architecture is chosen to set the performance standard since it closely resembles the physical configuration of the MAU-based machine. The forum for comparison is signal processing applications. The two hardware configurations are illustrated in Figure 20. We make the assumptions that:

1. the processors in the common bus architecture (P_i) have the same capabilities as the PEs in the MAU-based architecture, plus the facilities to handle data addressing and program control,
2. PEs and P_i 's are respectively homogeneous, and,
3. all PEs and P_i 's are executing identical processes.

The performance evaluation is presented in three steps: (1) a description of the multiprocessor performance models, (2) the development of the signal processing application model and (3) a complex FFT application performance comparison.

(a) MAU-based Multiprocessor

(b) Conventional Common Bus Multiprocessor

Notes: (1) all PE_i and P_i have
local data memory
(2) Bus is one data word in
width (for data transfer)

Figure 20: Performance Comparison Hardware Configurations

6.1 MULTIPROCESSOR PERFORMANCE MODELS

For the following discussion, we identify three types of instructions: data transformation, data transfer, and control. Data transformation instructions only invoke information processing within a PE or P_i ; they do not include fetching operands from global memory, or returning results to global memory. (Memory that is used for local data storage by a processor is considered to be "register" storage.) Data transfer instructions are used to move data between the processors and the global memory. Control instructions are all those instructions that affect

addressing information and program control. We define an "operation" to be a group of instructions that are required to accomplish some specific information transformation. Typically, an operation would include two data transfer instructions (get operands and store results), one or more transformation instructions, and several control instructions.

In a simple comparison of the typical common bus and MAU-based architectures, we find that the fundamental difference between them lies in the division of responsibilities for instruction execution. Each P_i in the common bus architecture is required to execute all three types of instructions. Conversely, in the MAU-based architecture, each PE executes transformation and transfer instructions, and the MAU executes control instructions on the behalf of all PEs. (Although the mechanism of instruction fetching is a notable difference between the two architectures, particularly if the processors in the common bus machine maintain local copies of programs, we shall disregard it within the scope of this comparison in view of the intended applications). By exploiting this difference, we can define the uncomplicated and yet representative models shown in Figure 21.

Bus throughput is chosen as the performance characteristic by which the two architectures are compared. The bases for this choice are as follows:

(a) Conventional Multiprocessor Model

(b) MAU-based Multiprocessor Model

T^c - mean control time/operation (seconds)

T_{tr}^d - mean data transform time/operation (seconds)

T_t^d - mean memory access time/operation (seconds)

t_w - mean data transfer time/word (seconds)

n_w - mean no. of words transferred/operation (words)

C - operation cycle time (seconds)

λ_b - bus throughput (words/second)

Figure 21: Performance Comparison Models

1. a performance bottleneck resulting from limitations imposed by bus throughput is expected in a shared bus architecture;
2. furthermore, because the MAU is physically positioned between the PE bus and the global memory, the bus throughput may be further reduced because of a congested MAU.

The bus throughput for the two architectures can be derived by using the models shown in Figure 21. From Figure 21 we have the definitions that follow.

1. The mean memory access time per operation (T_{td}) is the product of the transfer time per operation (t_w) and the mean number of words transferred per operation (n_w);
2. The mean data transform time per operation (T_{trd}) is the time a processor spends on "useful" information processing;
3. The apparent time spent on control per operation (T_c), is the time spent executing control functions (such as address calculations). In the typical common bus architecture, $T_c = K$, the mean control time per operation. In the case of the MAU-based architecture, T_c is regarded as the period of time a PE is delayed because the MAU is unable to deliver the next opcode. This delay results when the MAU is pre-occupied with executing control instructions for PEs.

This implies $T_c = \gamma_c(K, N, T_{trd}, T_{td})$, where γ_c is the apparent control time function and N is the number of PEs.

The total cycle time for an operation (C) is constrained as:

$$(Eq. 1) \quad C \leq T_{tr}^d + T_t^d + T^c$$

We do not consider the bus arbitration time since it is typically very small in comparison to all other quantities. From Little's rule [LITTL61], the bus throughput is computed as follows:

$$(Eq. 2) \quad \lambda_b = \frac{N n_w}{C}$$

In a Bottleneck Model formulation [WOODS81] of bus throughput, a set of constraints that define the limits of bus throughput are postulated. The active constraint is that which is the actual limiting factor under given conditions. In our models the constraints are:

1. (Eq. 1), the wait time for an operation ≥ 0 ,

2. (Eq. 2), Little's rule,

3. (Eq. 3), $\frac{\lambda_b T_t^d}{n_w} \leq 1$ bus is less than 100% utilized, and,

4. (Eq. 4), $\frac{\lambda_b K}{n_w} \leq 1$ MAU is less than 100% utilized.

The active constraints on bus throughput are (for each architecture):

1. Typical Common Bus -

a)

$$(Eq. 5) \quad \lambda_b \leq \frac{N n_w}{T_{tr}^d + T_t^d + K}$$

b)

$$(Eq. 6) \quad \lambda_b \leq \frac{n_w}{T_t^d}$$

2. MAU-based -

a)

$$(Eq. 7) \quad \lambda_b \leq \frac{N n_w}{T_{tr}^d \cdot T_t^d \cdot \gamma_c(T_{tr}^d, T_t^d, K, N)}$$

b)

$$(Eq. 8) \quad \lambda_b \leq \min\left(\frac{n_w}{T_t^d}, \frac{n_w}{K}\right)$$

Given the equation parameters and the γ_c function, we can produce the bus throughput constraint graphs required for the performance comparison.

6.2 SIGNAL PROCESSING APPLICATION MODEL

Signal processing applications are generally highly deterministic during execution. For example, on the MAU-based architecture or the typical common bus machine, a fast convolution [BOWEN82] would be computed as follows:

(instruction type:)	(program instruction)
	INITIALIZATION
(control:)	LOOP I TIMES
(transfer:)	LOAD PROCESSOR WITH INPUT ARRAY
(transform:)	COMPLEX FFT
(transform:)	COMPLEX ARRAY MULTIPLY
(transform:)	COMPLEX INVERSE FFT
(transfer:)	STORE SCALAR RESULT
	END LOOP

This sequence can be re-stated in terms of the types of instructions used:

control
transfer
transform(s)
transfer

where all of the transforms have been grouped as one. Furthermore, in a continuously executing process, the sequence would be repeated endlessly. This instruction progression is quite typical of signal processing computations. By allowing this sequence to represent an "operation", we can derive an approximate expression for γ_c .

6.2.1 The Derivation of the Apparent Control Time Function (γ_c)

The MAU, within the scope of this discussion, can be regarded as a processor that executes control instructions concurrently with the data transformation and data transfer instructions of the PEs. Therefore, in the case of an MAU-based machine containing only a single PE, we have the following example execution time lines:

1. underutilized MAU -

2. fully utilized MAU -

In the first example, where $K \leq T_{trd} + T_{td}$, we find that γ_c is approximately zero; for the second example, where $K > T_{trd} + T_{td}$, we get $\gamma_c = K - (T_{trd} + T_{td})$. If we extend this idea to N PEs in general, we derive the following expression for the average apparent control time function:

$$(Eq. 9) \quad \gamma_c \geq \begin{pmatrix} \approx 0 & ; K \leq \frac{(T_{tr}^d + T_t^d)}{N} \\ N \left[K - \frac{(T_{tr}^d + T_t^d)}{N} \right] & ; \text{otherwise} \end{pmatrix}$$

The intuitive meaning of (Eq. 9) is that: if the mean control time is sufficiently short, then it is "hidden" by the MAU, and if the mean control time is too long, then a PE experiences an average delay equal to the control time residues of all N PEs. This result was verified in a simulation of an MAU-based multiprocessor system through the use of a multitasking programming language [BUHR78].

6.2.2 Simulation of an MAU-based Multiprocessor Machine

In the simulation of an MAU-based multiprocessor machine, we map the "active elements" of the architecture to concurrent tasks, and the "passive elements" to procedures. The exception are the resource arbiters. Resource arbiters are implemented as "monitors" [HOARE74] [BUHR78], more specifically as two-level non-preemptive priority gladiator monitors. This monitor type is a variation on the original gladiator monitor [BOWEN80], and was developed specifically

for this simulation. Intertask communications is performed using monitor-based message mailboxes. The structure graph for the simulated system is shown in Figure 22. Note that the PE service task also carries out the bus arbitration duties. This is possible because of a selective receive capacity in the message mailboxes.

Figure 22: Structure Diagram of MAU simulation (Tasks and Monitors only)

The MAU-based multiprocessor simulation is described below.

1. A continuous 5 instruction sequence of control, control, transfer, transform, and transfer is used to represent each signal processing application process.
2. The prioritized fair-sharing algorithm for the MAU is:

- a) service each PE instruction stream for "j" instructions, or,
 - b) if while processing a PE's instruction stream another PE becomes idle (sleeping), then the MAU switches the target PE to the idle PE, (note that this does not apply if the present target PE already idle); multiple idle PEs are chosen using round robin selection.
3. The user specifies K, Ttrd, Ttd, and N. The simulation reports γ_c . All time is expressed in unit clock cycles.
 4. The basic unit cycle is represented by "i" loops of a PASCAL "FOR" statement, where "i" is specified by the user. This action enables isolation of the MAU characteristics from the underlying run-time operating systems characteristics. The proper "i" value can be calibrated using a single PE machine. (In the case of a single PE machine, $\gamma_c = K - (Ttrd + Ttd)$ such that $\gamma_c \geq 0$. Thus γ_c can be intuitively predicted for this simple case.)

Approximately twenty samples of γ_c were taken for each given set of input values. The number of processors was varied from 1 to 5, Td (=Ttrd + Ttd) was varied from 1024 to 4096 clock cycles, and K from 200 to 2000 clock cycles. (Varying proportions of Ttrd and Ttd gave less than 20% variation in γ_c thus allowing them to be considered as one value, as was predicted by the γ_c model).

Graphs showing a comparison of the theoretical and simulated γ_c values are found in Figure 23. In each graph, γ_c versus K is plotted for a family of curves varying in N , with a fixed T_d . Although there is some discrepancy between the theoretical predictions and the simulated results, the discrepancy is sufficiently minor thus allowing the γ_c relation to be considered an acceptable model of the MAU-based multiprocessor machine.

Figure 23: Graphs of Y_c versus K Comparing Theoretical and Simulation Y_c values (T_d fixed, family in N)

6.3 COMPLEX FFT PERFORMANCE COMPARISON

For performance comparison, we use the continuously executing 5 instruction sequence described in the previous subsection and consider the transform instruction to be a complex FFT. Bus throughput constraint graphs for 64-point, 128-point, 512-point and 1024-point (64 bits/point) complex FFTs are presented in Figure 24(a)-(d). The mean control time, K , is varied to demonstrate the effect of the MAU on bus throughput. The execution times of the complex FFT are typical of those found using a commercially available add-on array processor, the Data General Eclipse AP/130 [DATAG78].

As evidenced by the graphs in Figure 24, the MAU can become an active constraint on bus throughput when K is equal to T_{td} . However, since this circumstance is not likely to occur in practice, (particularly with large values of T_{td}), it can be concluded that, in general, the presence of the MAU does not adversely affect the bus throughput.

Figure 24: Complex FFT Performance Comparison Bus Constraint Graphs

2 2
OF / DE

In the second phase of MAU evaluation, we have shown that the MAU-based multiprocessor machine favourably compares in performance with an equivalent, conventional common bus multiprocessor machine, when considering signal processing applications. We chose the constraint on bus throughput as the performance characteristic to be used as the basis for comparison. Analytic models of bus throughput constraints were derived for both architectures. The MAU model was validated by a simulation of the MAU-based multiprocessor machine. A model of a signal processing application was developed in order to standardize the comparison environment. Within this environment, we demonstrated that the two architectures produced essentially the same performance under normal conditions.

In retrospection of the motivating philosophy of this research, we feel that the MAU-based multiprocessor architecture does provide the designer with a viable framework for developing simple, modular, special purpose multiprocessor/multiprocess system designs. This has been attained by striving for functional hardware modularity and hardware embedded multiprocess O/S kernel features. However, the benefits gained here are not without side-effects. A trade-off for general applicability has been made in order to provide the desired architectural features. As was indicated in chapter VI, the MAU can become a performance bottleneck in situations where control

In the second phase of MAU evaluation, we have shown that the MAU-based multiprocessor machine favourably compares in performance with an equivalent, conventional common bus multiprocessor machine, when considering signal processing applications. We chose the constraint on bus throughput as the performance characteristic to be used as the basis for comparison. Analytic models of bus throughput constraints were derived for both architectures. The MAU model was validated by a simulation of the MAU-based multiprocessor machine. A model of a signal processing application was developed in order to standardize the comparison environment. Within this environment, we demonstrated that the two architectures produced essentially the same performance under normal conditions.

In retrospection of the motivating philosophy of this research, we feel that the MAU-based multiprocessor architecture does provide the designer with a viable framework for developing simple, modular, special purpose multiprocessor/multiprocess system designs. This has been attained by striving for functional hardware modularity and hardware embedded multiprocess O/S kernel features. However, the benefits gained here are not without side-effects. A trade-off for general applicability has been made in order to provide the desired architectural features. As was indicated in chapter VI, the MAU can become a performance bottleneck in situations where control

time is an appreciable proportion of data transfer time. This is often the case in many non-signal processing applications (eg. in communications processing). Thus the MAU-based multiprocessor architecture would not be suitable outside of applications that exhibit a low control time to data transfer time ratio.

There is a further question: how many PEs can one MAU effectively control? The physical requirements for multiple instruction stream management seem to indicate that the MAU-based architecture is not suited for large numbers of processors. Furthermore, estimates of performance for large N cannot be obtained from the models used in chapter VI because the overhead that is neglected in the analysis of small numbers of PEs may become quite significant with large numbers of PEs. Thus, expecting the same level of performance for large N as one can predict for small N is not necessarily a valid extrapolation.

Natural extensions to this research seem to be: (1) the specification of a signal processing chip set centred around the MAU and (2) the development of an appropriate design methodology, incorporating the MAU-based multiprocessor architecture, that will provide a logical, structured path from design concepts to final implementation.

REFERENCES

- [ANDER67] Anderson, D.W., Sparacio, F.J., Tomasulo, F.M.: "The IBM/360 Model 91: Machine Philosophy and Instruction Handling", IBM Journal of Research and Development, Vol. 11, Jan. 1967, pp. 8-24.
- [BOWEN80] Bowen, B.A., Buhr, R.J.A.: The Logical Design of Multiple Microprocessor Systems, Prentice-Hall, Englewood Cliffs, N.J., 1980.
- [BOWEN82] Bowen, B.A., Brown, W.R.: VLSI Systems Design for Digital Signal Processing, Volume I: Signal Processing and Signal Processors, Prentice-Hall, Englewood Cliffs, N.J., 1982, pp. 162-166.
- [BUHR78] Buhr, R.J.A., Cavers, J.R., Hui, M.H.: "A Small Multiprogramming Kernel for PDP-11 Computers", Proceedings of DECUS Symposium, Ottawa, Canada, Feb. 1978.
- [BUHR82] Buhr, R.J.A., Mitchell, S.: "Object-Oriented Structured Design of Layered Protocol Systems", Third Intl. Conf. on Distributed Computing Systems, Ft. Lauderdale, Fla., October, 1982.
- [BUHR84] Buhr, R.J.A.: System Design with Ada, Prentice-Hall, Englewood Hills, N.J., 1984.
- [BOWRA76] Bowra, J.W., Torng, H.C.: "The Modelling and Design of Multiple Function-Unit Processors". IEEE Transactions on Computers, Vol. C-25 no. 3, March 1976, pp. 210-220.
- [BRINC75] Brinch Hansen, P.: "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. SE-1 no. 2, June 1975, pp. 199-207.
- [CHU81] Chu, Y., Abrams, M.: "Programming Languages and Direct Execution Architecture", Computer, Vol. 14 no. 7, July 1981, pp. 22-32.
- [COHLE81] Cohler, E.U., Storer, J.E.: "Functionally Parallel Architecture for Array Processor", Computer, Vol. 14 no. 9, September 1981, pp. 28-36.

- [DAHL72] Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming, Academic Press 1972.
- [DATAG78] Data General Eclipse AP/130 Array Processor Programmer's Reference, Data General Corporation, 1978.
- [DURNI79] Durniak, A.: "VLSI Shakes the Foundations of Computer Architecture", Electronics, May 24, 1979, pp. 111-133.
- [FAGGI78] Faggin, P.: "How VLSI Impacts Computer Architecture", IEEE Spectrum, May 1978, pp. 28-31.
- [FATHI83a] Fathi, E.T., Krieger, M.: "Multiple Microprocessor Systems: What, Why, and When", Computer, Vol. 16 no. 3, March 1983, pp. 23-32.
- [FATHI83b] Fathi, E.T., Krieger, M.: "An Executive for Task-driven Multimicrocomputer Systems", IEEE Micro, Vol. 3 no. 5, October 1983, pp. 32-41.
- [FLYNN66] Flynn, M.J.: "Very High Speed Computing Systems". Proceedings of the IEEE, Vol. 54 no.12, December 1966, pp. 1901-1909.
- [GUPTA83] Gupta, A., Toong, H.D.: "An Architectural Comparison of 32-bit Microprocessors", IEEE Micro, Vol. 3 no. 1, Feb. 1983, pp. 9-22.
- [GUTER83] Guterl, F.: "Chip Architecture: A Revolution Brewing", IEEE Spectrum, July 1983, pp. 30-37.
- [HIBBA78] Hibbard, P. et al.: "A Language Implementation Design for a Multiprocessor Computer System", Proceedings of 5th Annual Symposium on Computer Architecture, 1978, pp. 66-72.
- [HOARE74] Hoare, C.A.R.: "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17 no. 10, October 1974, pp. 549-557.
- [INTEL80] Introduction to the iRMX-86 Operating System, INTEL Corporation, Santa Clara, Calif., 1980.
- [INTEL82] INTEL iAPX 286 Advance Information, INTEL Corporation, January 1982.
- [KAHN78] Kahn, K.C.: "A Small-Scale Operating System Foundation for Microprocessor Applications", Proceedings of the IEEE, Vol. 66 no. 2, Feb. 1978, pp. 209-216.

- [KOPET79] Kopetz, H.: Software Reliability, Springer-Verlag, New York, N.Y., 1979.
- [KUNGH82] Kung, H.T.: "Why Systolic Architectures?", Computer, Vol. 15 no.1, January 1982, pp. 37-46.
- [KUNGS82] Kung, S.-Y. et al.: "Wave Front Array Processor: Languages, Architectures and Applications", IEEE Transactions on Computers, Vol. C-32 no.11, November 1982, pp. 1054-1066.
- [LAMPS77] Lampson, B.W. et al.: "Report on the Programming Language Euclid", SIGPLAN NOTICES, Vol. 12 no. 2, February 1977.
- [LINDE80] Linden, F.V.D., Wilson, J.: "Real-time Executive for Microprocessors", Microprocessors and Microsystems, Vol. 4 no. 6, July/August 1980, pp. 211-218.
- [LIPOV78] Lipovski, G.J., Doty, K.L.: "Developments and Directions in Computer Architecture", Computer, Vol. 11 no. 8, August 1978, pp. 54-67.
- [LITTL61] Little, J.D.C.: "A Proof of the Queueing Formula $L = \lambda W$ ", Operations Research, Vol. 9, 1961, pp. 383-387.
- [MEAD80] Mead, C., Conway, L.: Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980.
- [MORGA82] Morgan, C.L., Waite, M.: 8086/8088 16-bit Microprocessor Primer, McGraw-Hill, 1982.
- [ORGAN82] Organick, E.I., Lindstrom, G.: "Mapping High-Order Language Program Units into VLSI Structures", Proceedings of IEEE COMPCON'82, Feb. 1982, pp. 15-18.
- [PERRO79] Perrot, R.H.: "A Language for Array and Vector Processors", ACM Transactions on Programming Languages and Systems, Vol. 1 no. 2, October 1979, pp. 175-195.
- [PLESZ83] Pleszkun, H.R., Davidson, E.S.: "Structured Memory Access Architecture", Twelfth International Conference on Parallel Processing, August 1983, pp. 461-471.
- [PLUMM72] Plummer, W.W.: "Asynchronous Arbiters", IEEE Transactions on Computers, Vol C-21 no. 1, January 1972, pp. 37-42.

- [SMITH82] Smith, J.E.: "Decoupled Access/Execute Computer Architectures", Ninth Annual Symposium on Computer Architecture, April 1982, pp. 112-119.
- [SYMB082] Symbolics LISP Reference Manual, Symbolics Inc., 1982.
- [TJADE70] Tjaden, G.S., Flynn, M.J.: "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computers, Vol. C-19 no. 10, October 1970, pp. 889-895.
- [USDOD81] NEBULA Instruction Set Architecture, MIL-STD-1862A, U.S. Department of Defense, November 1981.
- [USDOD82] Reference Manual for the Ada Programming Language Draft Revised MIL Standard 1815, U.S. Department of Defence, July 1982.
- [VIEHM83] Viehman, M.J.: Spectrum Forum Letter, IEEE Spectrum, November 1983, pp. 8.
- [WIRTH71] Wirth, N.: "On the Composition of Well Structured Programs", Computing Surveys, Vol. 6 no. 4, December 1974, pp. 247-259.
- [WIRTH77] Wirth, N.: "MODULA - A Language for Multiprogramming", Software - Practice and Experience, Vol. 7, 1977, pp. 3-35.
- [WOLOC81] Wolochow, P.I.: "Microcomputer Operating System Trends", Digital Design, Vol. 11 no. 12, December 1981, pp. 51-56.
- [WOODS81] Woodside, C.M.: "Performance Prediction as Part of Design", Internal Report, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Canada, October 1981.

Appendix A

GRAPHICAL DESIGN NOTATION BY BUHR [BUHR84]

Ada-Based Graphical Design Symbols

buffer tasks are active FIFO message mailboxes

Note: tasks or packages enclosed by other modules are local to those modules

Non-Ada Graphical Design Symbol Extensions

Appendix B
DETAILED INSTRUCTION DESCRIPTIONS

CODE KEY

R = register
A = operand by any addressing mode
RD = register direct mode
M = any mode
O = operand
C = 16-bit constant (literal value)
UI = unscaled indexed
L = literal
V = vector
n = literal value
I = interrupt number
S = semaphore number
P = PE type literal value

* - all M except V
** - all M except V,L
*** - all M except L,RD

MAU INSTRUCTIONSONE OPERAND

MNEMONIC		DESCRIPTION	FORMAT
INCR	R	$R \leftarrow R + 1$	O, RD
DECR	R	$R \leftarrow R - 1$	O, RD
CLR	R	$R \leftarrow 0$	O, RD
TEST	R	$R = 0?$ set zero flag	O, RD

TWO OPERAND

MNEMONIC		DESCRIPTION	FORMAT
ADD	R, A	$R \leftarrow R + A$	O, R, M, <C> *
SUB	R, A	$R \leftarrow R - A$	O, R, M, <C> *
CMP	R, A	$R = A?$ set zero flag	O, R, M, <C> *

MOVE

MNEMONIC		DESCRIPTION	FORMAT
LOAD	R, A	$R \leftarrow A$ zero extended	O, RD, M, <C> *
STORE	R, A	$A \leftarrow R$	O, RD, M, <C> **
PUSH	R	$SP \leftarrow SP + 2$ $(SP) \leftarrow R$	O, RD
POP	R	$R \leftarrow (SP)$ $SP \leftarrow SP - 2$	O, RD

PROGRAM CONTROL

MNEMONIC		DESCRIPTION	FORMAT
IBLT	R1, R2 A	$R1 \leftarrow R1 + 1$ if $R1 \neq R2$ then $PC \leftarrow A$	O, RD, RD, UI, C index register = PC
DBGT	R1, R2, A	$R1 \leftarrow R1 - 1$ if $R1 \neq R2$ then $PC \leftarrow A$	O, RD, RD, UI, L index register = PC
JMP	A	$PC \leftarrow A$	O, M, <C> **
BCC	n, A	if PE c.c. = n then $PC \leftarrow A$	O, L, C, UI, C index register = PC
JSR	A	$SP \leftarrow SP + 2$ $(SP) \leftarrow PC$ $PC \leftarrow A$	O, M, <C> **
RTS		$PC \leftarrow (SP)$ $SP \leftarrow SP - 2$	O

SYSTEM/PROGRAM CONTROL

MNEMONIC	DESCRIPTION	FORMAT
WAIT S	S ← S - 1 IF S < 0 THEN suspend process	O, RD
SIGNAL S	IF S < 0 THEN move suspended process to RTR S ← S + 1	O, RD
DEFNSEM S, n DEFNINTR I, P	S ← n create PCB for interrupt I using PE type P	O, RD, L, C O, RD, L, C
DEFNPROC P	define process using PE type P	O, L, C
TERMPROC SWINTR I	terminate process cause interrupt I to occur	O O, RD
ENINTR	enable interrupt	O
DISINTR	disable interrupt	O
SWITCHPE P	switch to PE of type P	O, L, C

PE GENERIC INSTRUCTION FORMATS

MNEMONIC		DESCRIPTION	FORMAT
ALU1	R	R ← monadic R	O, RD
ALU2	R, A	R ← R dyadic A	O, RD, M, <C> *
LOADPE	R, A	R ← A	O, RD, M, <C> ***
STOREPE	R, A	A ← R	O, RD, M, <C> ***
MOVEPE	R1, R2	R1 ← R2	O, RD, RD
SYSTEM1	R	any system operation based on R; R unchanged	O, RD
SYSTEM2		any system operation not requiring operands	O
SAVEPE		save all internal registers in stack area	O
RESTORPE		restore all internal registers from stack area	O

Appendix C

EXAMPLE MAU OPERATION EXECUTION TIME AND CONTROL STORE ESTIMATE CALCULATION

MAU Operation = INCR R instruction

1 cycle - select ALU input A source; select register R;
 select increment operation
1 cycle - latch ALU input a
1 cycle - compute
1 cycle - latch ALU output
1 cycle - write output to register R

Total time = 5 cycles

Total number of sequence steps = 5

Total number of control lines = 22 for ALU and registers
 + 6 for MAU shared resource
 acknowledge lines
 = 28 lines

Total control store = 5 steps x 28 control lines x
 1 bit/control line-step = 140 bits

Appendix D

MAU OPERATIONS: EXECUTION TIME AND CONTROL STORE ESTIMATES

<u>OPERATION</u>		<u>EXECUTION TIME</u>	<u>CONTROL STORE</u>
		(unit clock cycles)	(bits)
MAU	control	5 -> 61	4859
MAU	bootstrap	268	2000
PES	control	2 -> 14	1590
IFP	control	2 -> 21	460
PDT	control	8 cycles/32 bit word	570
INCR	R	5	140
DECR	R	5	(shared)
CLR	R	2	50
TEST	R	2	(shared)
ADD	R,A	10 -> 26	1428
SUB	R,A	10 -> 26	(shared)
CMP	R,A	10 -> 26	(shared)
LOAD	R,A	9 -> 21	1122
STORE	R,A	9 -> 21	(shared)
POP	R	10	510
PUSH	R	11	561
IBLT	R1,R2,A	15 -> 20	940
DBGT	R1,R2,A	15 -> 20	(shared)
JMP	A	5 -> 13	752
BCC	n,A	12 -> 18	880
JSR	A	24	1539
RTS		10	570
WAIT	S	5 -> 62	189
SIGNAL	S	6 -> 16	162
DEFNSEM	S,n	5	76
DEFNINTR	I,P	29	338
SWINTR	I	2	53
ENINTR		1	32
DISINTR		1	32
DEFNPROC	P	5	60
TERMPROC		5	46
SWITCHPE	P	15	528
ALU2	R,A	11 -> 19	1210
LOADPE	R,A	11 -> 19	(shared)
STOREPE	R,A	11 -> 19	(shared)
ALU1	R	3	63
MOVEPE	R1,R2	5	105
SYSTEM1	R	3	63

SYSTEM2	2	42
SAVEPE	2	42
RESTORPE	2	42

Total control store = 21055 bits

Total number of transistors for control store
(including 50% overhead) = 30000 (approximately)
= approx. 30000

END

112-06-186

FIN