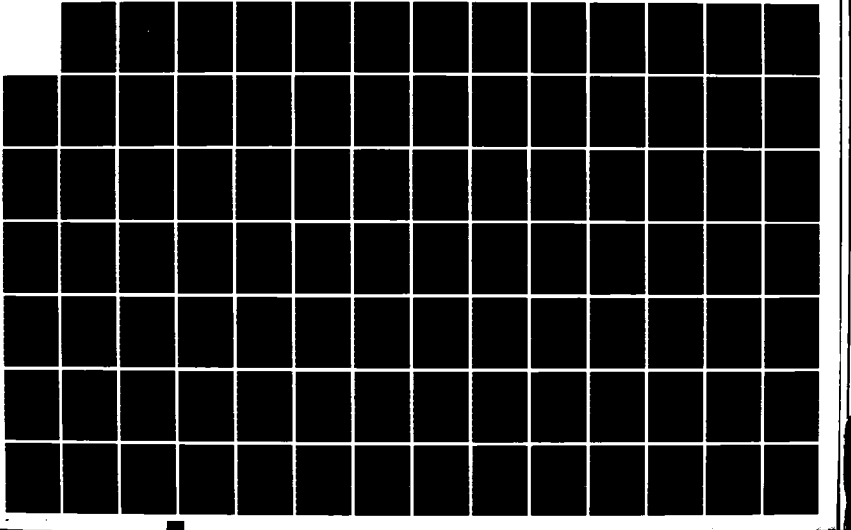
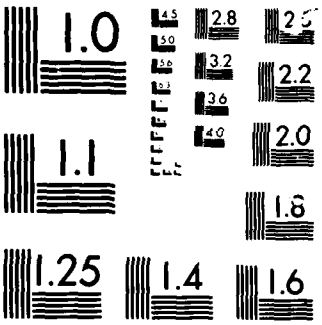
AD-A167 873 THE RISC (REDUCED INSTRUCTION SET COMPUTER) ARCHITECTURE AND COMPUTER PERFORMANCE EVALUATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M F BARROS 1/2

UNCLASSIFIED MAR 86 F/G 9/2 NL





(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE RISC ARCHITECTURE AND
COMPUTER PERFORMANCE EVALUATION

by

Manuel Filipe Pedrosa de Barros

March 1986

Thesis Advisor: Harriett B. Rigas

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 62	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
THE RISC ARCHITECTURE AND COMPUTER PERFORMANCE EVALUATION

12 PERSONAL AUTHOR(S)
Manuel Filipe Pedrosa de Barros

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Engineer's Thesis	FROM _____ TO _____	86 March	97

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	RISC Architecture; CISC Architecture; Computer Performance Evaluation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

A definition of Reduced Instruction Set Computers is developed.

A computer performance model which allows the evaluation of architectural alternatives is presented.

An example on the use of the model to compute the performance alternatives for a given application is presented to study the effect of the addition of an instruction to a processor instruction set.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Prof H. Rigas	22b TELEPHONE (Include Area Code) (408)646-2082
	22c OFFICE SYMBOL 62Rr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

The RISC Architecture and Computer Performance Evaluation

by

Manuel Filipe Pedrosa de Barros
Lieutenant, Portuguese Navy
B.S., Escola Naval, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
March 1986

Author: _____
Manuel Filipe Pedrosa de Barros

Approved by: _____
Harriett B. Rigas, Thesis Advisor

_____
Larry Abbott, Second Reader

_____
Harriett B. Rigas, Chairman, Department of Electrical and Computer Engineering

_____
John N. Dyer,
Dean of Science and Engineering

ABSTRACT

A definition of Reduced Instruction Set Computers is developed.

A computer performance model which allows the evaluation of architectural alternatives is presented.

An example on the use of the model to compute the performance alternatives for a given application is presented to study the effect of the addition of an instruction to a processor instruction set.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my gratitude to Prof. Rigas for her guidance in completing this project.

To my parents for all they taught me and finally and most important to my wife Carmo and my son Andre for their constant support and understanding, without which I would not have got here, I dedicate this work.

I. INTRODUCTION

The first Reduced Instruction Set Computer (RISC) appeared at the end of the 1970's and since then long and heated discussions have taken place in the computer architecture community. These discussions centered around the validity of the claims made by the RISC proponents regarding the performance achieved by the proposed machines when compared to traditional computers that are referred to as Complex Instruction Set Computers (CISC).

Due to a lack of an appropriate method to evaluate the performance effects of various architectural features, it is difficult to resolve the RISC/CISC controversy.

The interest in the ideas proposed by this philosophy has been growing, and presently many of the major computer companies are investing a great deal in this new type of computer architecture.

This thesis tries, first, to define the basic characteristics of a Reduced Instruction Set Computer, so that it is possible to focus on the specific architectural features peculiar to RISC machines.

The approach that in the author's opinion has to be followed, in order to evaluate computer performance, together with the author's disagreement on the approach taken on several published comparisons between RISC and CISC machines, are presented.

A model for computer performance evaluation is suggested. This model is composed of two parts. The first part deals with the timing analysis of the computer performance. The second part sets a criterion to determine the efficiency of a given computer control unit when used for a given application. Finally in order to evaluate the model, an example is given demonstrating the quantification of the

performance effects of an architectural enhancement to a system architecture.

The model suggested for computer performance evaluation constitutes a departure from the current computer performance evaluation methods, because the attention is centered on the computer architecture rather than on the measurements of throughput, response time and mean job turnaround time where the main emphasis of the evaluation process is put on the software.

The model is intended to provide a tool for computer architects to use, so that discussions regarding the performance achievements of certain architectural features might be quantified and rational conclusions may be reached.

II. WHAT IS A RISC ?

A. INTRODUCTION

In recent years a new type of computer architecture has received a great deal of attention.

This new architecture is mainly the result of an effort conducted in an academic environment. Profiting from the new possibilities that custom VLSI offers, the professors and students at the University of California at Berkeley, collaborating in several courses in this area, began projects on building single chip computers.

Due to limitations of the chip area, available tools and the available time for the completion of the project, several simplifications to contemporary architectures were made. For example, the instruction set was simplified by eliminating all instructions that might be called composite instructions. This type of instruction is equivalent, in the operation performed, to a sequence of other more elementary (atomized) instructions.

A claim has been made, that the obtainable performance of these machines was unexpectablly remarkable and this triggered a major discussion on the subject of the merits of RISC's.

Feeding the controversy is undoubtly the lack of an appropriate method or tool to measure computer architecture performance and the effects of a particular architecture modification on the computer performance.

From the very beginning the RISC machines were related to implementation issues in the use of VLSI technology.

Proponents called the approach "RISC", for Reduced Instruction Set Computers, as opposed to the traditional computers which they referred to as "CISC'S", for Complex Instruction Set Computers.

The "new architecture" proponents didn't present it as a proposal to enhance, in some way, the prevailing architecture, but as a complete departure from the previous work.

No precise definition has ever been given for the complete characteristics of a RISC machine, and because of that, there are now in existence several different machines all claiming to be RISC's. Although there are some common features there is no clear cut agreement on what comprises a reduced instruction set computer.

No doubt some very valid ideas were brought to the computer architecture environment by the "RISC philosophy proponents", but, nevertheless, it constitutes a sure risk to accept a new idea without an open, substantiative debate where the benefits are separated from the jargon.

The first step in understanding and identifying the RISC trade-off is a more precise definition of RISC.

As stated above, several implementations of RISC's are already in existence, and, of these, four have undoubtly enough importance to be mentioned.

They are:

1) The RISC I and II, developed at the University of California at Berkeley
2) The 801 Minicomputer, developed at the IBM Thomas S. Watson Research Center
3) The MIPS, developed at Stanford University.

In order to develop a definition of the "RISC" the existing "RISCs" should be studied.

B. THE RISC I AND II

The RISC I and II were both developed at the University of California at Berkeley where the acronym RISC originated. Since both were developed at U. C. Berkeley, they are very similar in their composition. In fact, RISC II is no more than an enhanced version of RISC I.

Both are single chip VLSI processors having the following characteristics:

1) They are 32-bit machines. That is, all registers and busses are 32 bit wide.

2) Instruction Set:

   2a) RISC I has 31 instructions
   RISC II has 39 instructions

   2b) Both have a load/store architecture. This means that all instructions except load and store are register-to-register. Load and store are the only memory-reference instructions.

   2c) All instructions except LOAD and STORE are single-cycle where a cycle is the time it takes to read and add two registers, and then store the result back into a register.

   2d) All instructions are the same size (32 bits). There are two different formats but the fields are at fixed locations.

   2e) Addressing Modes:
   There are two addressing modes; one for register-to-register instructions--Register Direct and the other for memory reference instructions--Index + Displacement.

3) Registers

   3a) Total number of on-chip registers
   RISC I --- 138
   RISC II --- 198

   3b) The processor is organized in multiple overlapping windows in order to facilitate parameter passing between procedures.
   The windows are organized in a circular buffer fashion. In the case that the nested procedure depth is greater than the number of windows minus one, the values in the window corresponding to the oldest procedure are stored in memory and this window is then free to be allocated to the current procedure. At any time 32 registers are visible constituting what is called the "current window". All windows have a fixed size and the composition shown in Figure 2.1.
   The global registers are common to all procedures, and therefore they are used to store global variables. Register R0 holds a fixed value of zero. The low registers are common to the current procedure and to the called procedure, although, in the called procedure, they will have a different number since there they constitute the high registers of the corresponding window. The high registers are common to the current procedure and to the calling procedure. The high and low registers along with the global registers constitute the overlapped part of each window and are used for parameter passing between procedures. The local registers are only visible in the current window.

4) The control unit is hardwired with most of its logic implemented using PLA's.

5) Pipeline Stages
   The RISC I has two pipeline stages, i.e., depending on the program sequence it can prefetch the next instruction while it executes the present

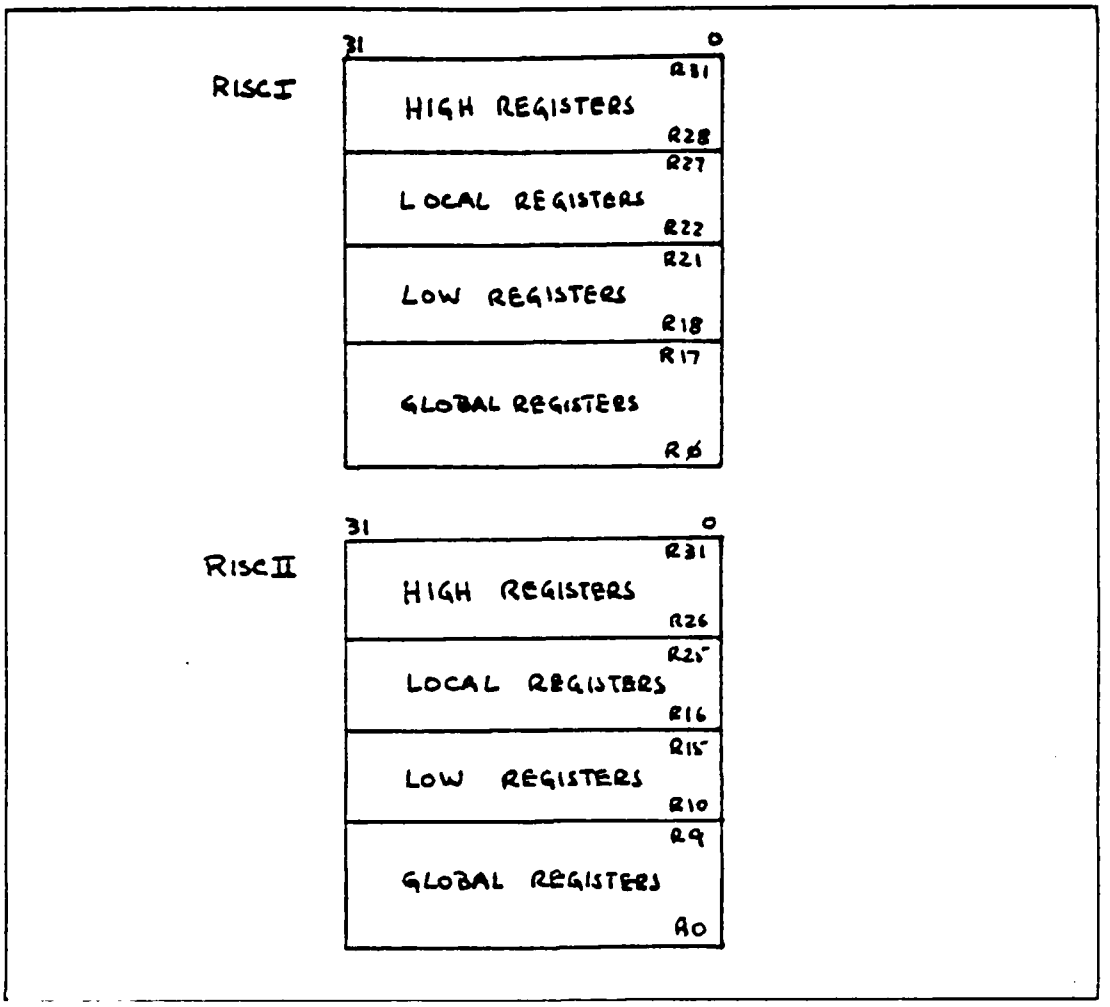


Figure 2.1 RISC Register Window.

instruction. The RISC II has three pipeline stages, i.e., depending on the program sequence it can prefetch the next instruction and store the final results of the previous instruction in a register, while it executes the present instruction.

6) Use of Delayed Branch
In order to increase speed and not to discard the prefetch instruction, when a branch instruction is executed, the branch takes place only after the execution of the next sequential instruction. Typically the compiler arranges for the instruction following the branch to be part of the loop, see [Ref. 1].

8) Implementation
RISC I is implemented with 4 micron NMOS VLSI technology with a clock of 8 MHZ and a cycle of 500 NSEC. RISC II is implemented with 3 micron NMOS VLSI technology with a clock of 12 MHZ and a cycle of 330 NSEC.

9) Both RISC I and II have no floating-point support.

C. THE 801 MINICOMPUTER

Developed by IBM at the Yorktown Heigths Research Center from 1975 until 1979, it was the first machine to follow what later would be called "The RISC Approach to Computer Architecture".

Due to its proprietary nature, not much is known about it, but some of the ideas present in its design are known and have been, in a certain way, the basis for the development of RISC I and II at Berkeley and MIPS at Stanford.

As opposed to the RISCs and the MIPS, the 801 is not a single chip processor but a minicomputer.

The general approach is the basis for the design of an IBM NMOS VLSI single chip processor known as ROMP or 802.

The 801 machine is basically a 32 bit architecture with single-cycle four byte instructions and 32 registers. It has separate data and instruction cache memories. As in RISC I and II, the 801 also has a delayed branch scheme, that is the branch only takes place after the execution of the next instruction.

The 801 system is said to be compiler-based meaning that a greater demand is made on the compiler.

The 801 architecture was defined by George Radin in his article 'The 801 Minicomputer' [Ref. 2] as the set of run time operations which:

1) Could not be moved to compile time
2) Could not be more efficiently executed by object code produced by a compiler which understood the high-level intent of the program, or
3) Was to be implemented in random logic more effectively than the equivalent sequence of software instructions.

Both data and address busses are 32 bit wide. The addressing modes are few:

- base+index
- base+displacement

- register direct.

Also a highly-effective optimizing compiler was developed for the system.

D. THE MIPS

The MIPS computer was developed at Stanford University by John Hennessy and his students. Its acronym stands for Microprocessor without Interlocked Pipe Stages.

There are strong similarities with the RISC project at Berkeley. It has, however, some conceptual differences that have already been identified by its proponents in Ref. 3 as:

i) more complex user level instruction set.

ii) the main design goal is high performance of the hardware employed and not simplicity of the instruction set.

iii) much more complex compiler.

Specifically its characteristics are the following:

1) 32 bit machine.

2) Instruction Set

2a) 55 instructions

2b) Load/store architecture

2c) All instructions except LOAD and STORE are single-cycle

2d) Instructions may be 16 or 32 bit long. An optimizing compiler reorders the instructions so that all 16 bit instructions always come in pairs.

2e) Addressing Modes
- immediate
- base with offset
- indexed
- base shift

3) Registers
There are sixteen 32-bit general purpose registers,

4) Hardwired control with most of its logic implemented using PLA's

5) Use of Delayed Branch instructions

6) Five pipeline stages

7) No condition codes

8) Word-addressable machine

9) Separate data and instructions memory

10) No support for floating-point operations

11) Implemented with 4 micron NMOS VLSI technology with a clock rate of 8 MHZ.

E. TOWARD A DEFINITION OF A RISC MACHINE

Four machines have been described as examples of a new type of computer architecture defined as the RISC architecture, as opposed to the traditional architecture now referred to as CISC architecture.

Any definition of this architecture will have to encompass the characteristics common to the four previous examples.

To summarize, a RISC Machine will have the following characteristics:

1) Simple instruction set where the great majority of the instructions are single-cycle,
2) Load/store architecture, that is all instructions are register-to-register with the LOAD and STORE being the only memory-reference instructions,
3) Very few addressing modes,
4) Hardwired Control i.e. no microcode,
5) Instructions with one or two sizes and with fields at fixed locations,
6) Some degree of pipelining,
7) Demand on the compiler to increase performance.

III. MY APPROACH TO COMPUTER PERFORMANCE EVALUATION

A. INTRODUCTION

This thesis has been motivated by the rise of the new, RISC computer architecture trend, described in the previous chapter, and by the claims made by RISC proponents regarding the inherent superior performance of RISC when compared to traditional architectures.

Unfortunately, the claims made for these structures were not supported by any quantitative arguments. No specific attention was given to the effects of various factors introduced in the RISC architecture and to the influence that each factor had on the system performance.

Computer performance evaluation is different depending on the aspects of performance being evaluated. From the view point of a potential computer system buyer, there is a need to identify features in the system which will enhance the performance for a particular application. From the viewpoint of a computer architect, performance analysis is a way to evaluate specific enhancements from which trends in computer architecture design may follow.

B. EVALUATION AND MEASUREMENTS

In order to perform an evaluation of any kind, one must take measurements of the system under different conditions. One wants to take the measurements properly, or else the evaluation will be unvalid.

In order to guarantee that the evaluation will be based upon correct data, one has to know:

1) What the measurements are for

The buyer is not worried about any of the architectural details of the machine, but rather about the throughput of a system programmed in a high-level language.

In contrast, the computer architect must be concerned with the internal characteristics and the behavior of the system, even when he is testing a system using programs written in high-level languages.

Considering the RISC family of machines the correct point of view is undoubtly the latter one.

2) What is measured

Typically one wants to test how each enhancement to the computer architecture affects the system performance. In order to get a realistic comparison of features, only one feature at a time may differ. If more than one feature is different, it is difficult to measure the individual effect of each architectural feature on the system performance.

3) How is the evaluation performed

Because it is not feasible to build a new system each time one of the architectural features is altered, a model is required.

Because it is through the use of a model that the performance effects of any architectural feature will be determined, this model has to be able to quantify, in a precise manner, the effects of any change in the architecture.

4) For which application are the measurements valid

The application for which the system is being used has an effect on the system performance. No system will show the same performance in two different environments. For example, in one application the user might be doing only word-processing, and, in the second, the system might be floating-point intensive.

There are, nevertheless, systems that present a balanced performance throughout a diversified number of applications. They are the so called "General Purpose Computers". But even for these, the performance fluctuates, indicating that general purpose computers have a better performance for some applications than for others.

Due to these reasons, the system performance evaluation must pay attention to the rigorous definition of the application for which the system performance is being evaluated.

This requirement for a precise definition of the application, will clarify the validity of the conclusions.

5) Which factors interact with the measurements

In the second question, the need to make just one change at a time when making the evaluation is emphasized, otherwise it would be impossible to determine the individual effect of an enhancement on the system performance.

Specifically if the evaluator has already made measurements for several changes in the architecture and has also quantified the effect of each of those changes on the system performance, it is possible to compare two systems, that differ by all those changes plus an extra one, not yet considered. As a result of the analysis, the effect of this last change on the system performance can be quantified.

C. THE RISC/CISC CONTROVERSY

Because the problem being discussed is related to computer architecture, there is a need for a concise statement defining Computer Architecture as it is commonly understood.

The adopted definition is the IEEE standard 729-1983 stating Computer Architecture as:

" The process of defining a collection of hardware and software components and their interfaces to establish a framework for the development of a computer system. "

In the published papers on RISC, several comparisons of CISC and RISC examples were made.

The way these comparisons were done did not give any insight, to the answers to the questions presented in the previous section, or other similar questions.

The result is that now, no one knows for example, if the performance of the RISC II is due primarily to its register

organization scheme, as some claim, or to the simplicity of its instruction set, as others do.

Specifically,

1) If one wants to evaluate the effects of reducing the instruction set, one might pick a CISC machine e.g. the VAX-11 and consider the improvements due to all the instructions whose execution is equivalent to a sequence of simpler instructions. For each of these more complex instructions one could determine if the execution is faster than the equivalent sequence. If that is not the case, the instruction should be discarded. If an improvement is seen, then consider the cost of adding the instruction to the instruction set.

2) If one wants to evaluate the effects of reducing the number of addressing modes, one should consider:

   - Why are they needed ?
   - With which data types are they used ?
   - What its the benefit brought by its addition.

3) If one wants to evaluate the effects of overlapped register windows, one should test implementation of overlapped windows on several systems and measure, as a cost/benefit ratio, the effect of overlapped windows on the system performance.

4) One cannot change more than one feature at a time and hope to get an idea of what the effect of each feature is on the system performance.

5) If one wants to do an evaluation using programs written in a high-level language, one should state that as a limiting factor. Since different compilers generate different code, some compilers are better than others and therefore make different contributions to the system performance. Furthermore, in the case of compiler generated code, the frequency of execution of each instruction in the system instruction set will be different for different high-level languages. Besides, two different systems with distinct instruction sets do not necessarily have the same best compiler.

6) If one wants to make some conclusive statement about the advantages and disadvantages of the RISC architecture, one must separate the effects of features that are orthogonal to the RISC philosophy.

The fact is that in the papers published on RISC's, almost all the comparisons made, involved systems with different instruction sets, different addressing modes and a different number of registers and registers organization schemes. Furthermore compiler generated code was used without considering the performance effects. These are the reasons why no one can say whether the RISC architecture is or is not better by itself.

In this situation, while the RISC proponents are bringing some jargon to the architectural environment, those against RISC are losing track of the possible benefits present in the RISC philosophy.

D. AN EXAMPLE

As an example, let us pick a common CISC processor, the MC68000 and consider its addressing modes.

The MC68000 has six basic types of addressing modes, namely:

1) REGISTER DIRECT - The effective address is the register designation field in the instruction.

$$EA = Rn$$

2) ABSOLUTE - The effective address is that given in the instruction field itself and it is used directly without modification

$$EA = \text{INSTRUCTION FIELD}$$

3) REGISTER INDIRECT - The effective address is the contents of the designated register

$$EA = (Rn)$$

4) IMMEDIATE - The operand is part of the instruction itself and no further addressing is needed

5) PROGRAM COUNTER RELATIVE - The effective address is computed by taking the value in the program counter register and adding or subtracting an offset value

$$EA = PC + OFFSET$$

or

$$EA = PC - OFFSET$$

6) IMPLIED - The operand is in a register designated by the mnemonic of the instruction.

The uses of each addressing mode depends on the programmer.

Until now, the philosophy present in the design process was to give the maximum versatility possible to the programmer, so that he or she could choose the address mode better suited to his or her needs. The rise of the RISC architecture brings some questions regarding the correctness of this philosophy.

In order to answer these questions, there is a need to have a correct method for the evaluation of a system performance. Together with the evaluation method there are some points that have to be considered when deciding how many addressing modes to include in the system instruction set and how long each addressing mode should be.

The considerations are to:

1) reduce the storage requirements per program

2) reduce the number of bits that must be moved between processor and memory to execute a program, i.e., reduce the bandwidth requirements on the bus

3) reduce the average length of an instruction, i.e., reduce the required width of the instruction bus.

There is a trade-off between the number of instructions needed for the system to execute a program and the average instruction size.

The decision regarding the number of addressing modes to include is also very much dependent on the application, on the data types, on the operations involved, on the use of nested procedures, and how the parameter passing operation is accomplished between procedures.

Although not considered here, the addressing problem is also very much related to schemes of memory protection where one wants to forbid the regular user program from accessing some part of memory.

Besides how each one of the addressing modes is used, it is also important to consider the frequency with which each addressing mode is used.

Not much material is available regarding the usage of addressing modes. As an example, consider again the addressing modes of the MC68000.

1) REGISTER DIRECT

Since the operand is, in this case, in a register, no memory accesses are involved. This provides some speed advantages when used for operating on frequently-accessed variables. For infrequently-accessed variables it would not

be used because the number of registers available on-chip is usually very small.

2) ABSOLUTE

A memory access cycle is involved in absolute addressing, because the operand is in memory. For this reason it is not as fast as the previous mode.

Absolute addressing does not have much versatility because the instruction address field is constant and the operand must reference a fixed location in memory. Nevertheless, it is simple. Because no alteration on the address field of the instruction is performed, absolute addressing is an efficient mode to use when the operand is within the range of the instruction.

3) REGISTER INDIRECT

In the register indirect mode, one register access plus one memory access cycle are involved because the register holds the operand address and not the operand itself.

The register indirect approach is used when the address of the operand has just been calculated. It provides address-range extension, and in fact this extension increases with the difference between the size of the instruction address field and the size of the specified register.

4) IMMEDIATE

Immediate addressing is the fastest way of addressing, although it is limited by the instruction size. No additional memory accesses are needed since the operand is within the instruction itself. Since programs are not self-modifying it is used only for predefined values---constants.

5) PROGRAM COUNTER RELATIVE

The major advantage of relative addressing is that it allows the generation of position independent code because the location referenced is always fixed relative to the

program counter. The importance of this fact is very much dependent on the memory management scheme adopted in the system.

In addition to the regular memory access, an addition or subtraction must also be executed. It is used in relative jump instructions e.g., to set up loops or to set up parameters to be passed to a subroutine.

6) IMPLIED

Implied addressing is equivalent to the register direct addressing. However, implied addressing restricts the opcode to the predetermined register specified by the design of the opcode and the design of the processor.

E. SUGGESTED APPROACH

It is not feasible to build a new system each time a single architectural feature is changed, in order to evaluate its effects on system performance.

As a result, there is then need for a model.

This model should be clear, complete, and able to reflect the interrelations that exist between the different components. The model should also be applicable to any computer system, i.e., the model should be general.

The model should reflect the performance effects of any computer architectural feature such as:

- Bus Width
- Addressing Modes
- Pipelining
- Instruction Queue
- Instruction Prefetching

In the method suggested for computer performance evaluation, a comparison is made between a reference system and the same system with some change. The reference system is the computer system for which it is desired to determine the impact of each architectural enhancement. The result of this comparison will then constitute a measure of the

performance effects of the particular change. The conceptual view of the system used in the model is illustrated in Figure 3.1.

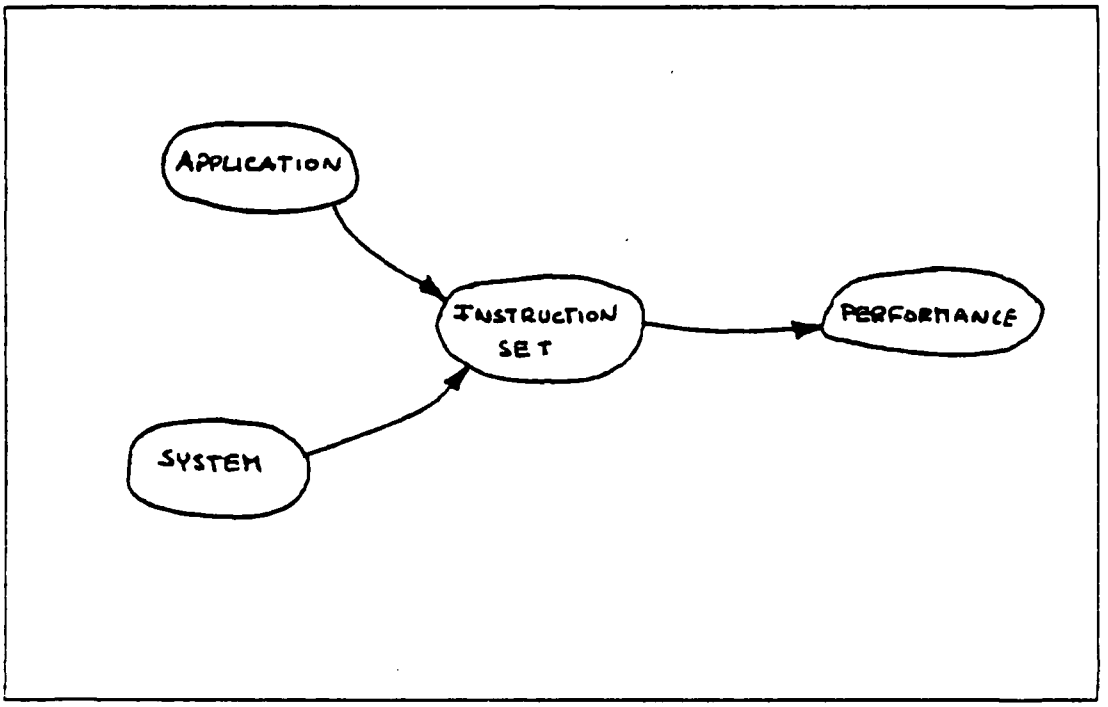


Figure 3.1 Conceptual View.

Four entities are considered:

1) The Application, any evaluation will only be valid for a certain application, not for any application
2) The System being considered
3) The System Instruction Set
4) The Performance, as the object of the evaluation process.

The instruction set constitutes the central point of the conceptual view. The application uses it. The system supports it. The best match will necessarily give the best performance.

The application is characterized by a set of tasks that must be performed. Each task is performed with a different frequency. For each task a program must be written, so that

one task is mapped into one program. Each one of these programs executes in a different time.

The weight of each task or its representation in the application is then the product of the frequency of its execution and the corresponding program execution time.

The effects of the application on the system performance are the frequency of execution of each instruction in the system instruction set. This together with the average execution time of the programs of interest will ultimately lead to a " typical " program of the application.

The system supports an instruction set in two ways: one by the execution time of each instruction and the other by the complexity of the control unit necessary to implement the instruction set.

An instruction set is desired that allows for the writing of programs with a minimum execution time, but also minimizes the amount of support that has to be given by the system.

IV. TIMING ANALYSIS

A. INTRODUCTION

In this chapter a detailed analysis of the model for computer performance evaluation is introduced. As described in the previous chapter the model is divided into two parts. In the first part, the model considers a timing analysis. In this analysis the application determines the dynamic frequency of execution of each instruction present in the system instruction set and finally the system architectural characteristics determine the execution time of each instruction.

In the second part of the model, which follows in the next chapter, the model considers the relation between the application and the control unit necessary to implement the system instruction set. From this relation a performance figure is obtained.

Any architectural feature will have consequences both in the execution time of each instruction and in the complexity of the control.

As has already been mentioned the first part of the model is a timing measure. It will consider the execution time of the specified application's " typical " program.

Several factors contribute to the execution time of a program and not all of them are part of the computer architecture. Some have depend on the implementation of the system.

The implementation is very much related to the technology chosen. The technology will determine, for example, the maximum clock rate obtainable and the number of computer components to be placed on chip.

Two factors have a great impact on the system performance, they are the clock rate and the average memory access

time. Also the number of components on chip is an important factor, since one of the most time consuming operations is to transmit data from one place to another. For example by being able to have more registers on chip, one might be able to reduce the average operand access time and therefore speed up the computer operation. If one considers the storage registers as part of the system memory then one can see that the average memory access time is reduced.

In the suggested approach to computer performance evaluation, the main concern is architectural features and not implementation restrictions due to technology limitations. The reason for this is that a method to evaluate computer performance should be general and therefore be able to survive constant technological change.

B. THE COMPUTER SYSTEM

Any computer system architecture is made of hardware and software tools. In the area of software, an important factor is the operating system.

For the sake of simplicity, and since in fact the operating system is also a program that has to be run on the system, it can be considered as part of the application in the computer performance evaluation process.

If the operating system is not considered as part of the application software there would be a need to track all calls to the operating system, measure the time the system takes to execute the correspondent subroutines and subtract this from the program execution time.

In the hardware, the major components are:

i) the processor
ii) the memory
iii) the busses
iv) the I/O interfaces
v) glue circuits

The processor consists of the portions of the computer made up of the control unit, the arithmetic logic unit, the general purpose registers and the busses that connect all of these.

The memory consists of all the parts of a computer used for either temporary or permanent storage, for instructions or for data. The busses are a collection of signal lines with multiple sources and multiple sinks. They provide for the intercommunication capability among the other computer components. The I/O interfaces are the parts of the computer through which the system communicates with the outside world.

In order for the overall system to have a good performance, it is desired to balance the average work done by each component per unit of time. Since each computer component has a different function, the work done by each is different from the others. It is this work that has to be characterized, so that an understanding of how to maximize it, is possible.

One requirement is that the idle time for each component should be as low as possible. For example the processor should be in an idle state for a data element stored in memory as little as possible.

1. Memory and I/O Interface

Both memory and I/O interface can be considered together, since both are communication media. Memory performs a communication between two instants in time. I/O interfaces perform a communication between the computer system and the outside world.

For both memory and I/O the work is characterized by how long it takes to correctly receive a unit of information from the bus and how long it takes to correctly place the same unit of information on the bus. This unit of information will be the same in the case of instructions and data. This unit of information is then one bit.

For both memory and I/O, the measure of their performance is the number of bits that are received or transmited per unit of time. This is in fact no more than a bandwidth in units of bits per second.

For example, a memory unit with a word size of sixteen bits and an access time of two microseconds performs the same work as another memory with a word size of thirty two bits and an access time of four microseconds.

$$\text{MEMORY BANDWIDTH} = \frac{\text{MEMORY WORD SIZE}}{\text{MEMORY ACCESS TIME}} \quad (\text{BIT/SEC}) \qquad (4.1)$$

2. The Busses

The function of a bus is to pass information from a computer component acting as a source to other components acting as sinks. The memory and I/O interfaces are also communication media that treat data and instructions in the same way.

The nature of these signals has no influence on the characterization of the bus work or the efficiency with which the bus preforms its work.

The bus work is characterized by:

i) the number of active sources at a time, here assumed to be one

ii) the number of active sinks

iii) the number of signal lines, i.e., the bus width

iv) the bus cycle time

As its function is to be a communication medium, the bus work is measured by a bandwidth in units of bits per second.

The particular bus bandwidth will be given by:

$$\text{BUS BANDWIDTH} = SI \times \frac{WI}{BCT} \quad (\text{BIT/SEC}) \qquad (4.2)$$

where

SI - is the number of active sinks

WI - is the bus width

BCT - is the bus cycle time

3. The Processor

After receiving data and/or instructions from the bus, the processor alters this data according to the sequence of instructions and then delivers the final results back to the bus.

While the previous computer components treat data and instructions in the same manner, this is not true for the processor case. In this case, instructions specify the operations that have to be performed, and the data constitutes the object on which the operations are performed.

The structure of the processor, i.e., the specific configuration of each element is dependent on the instruction set and on the data types involved. The instruction set configuration makes requirements on the processor, because the instruction set is intimately related to the processor control unit and the datapath.

The data types involved in an application should be supported by the processor. If, for example, a lot of array manipulation is done, then it is to be expected that the system considers some parallel operation capability.

In addition to the data types, the instruction set is also dependent on the application. Therefore the processor structure is also dependent on the application.

C. THE APPLICATION

An application is characterized in the same way independent of the computer system being evaluated. It is characterized by a certain number of tasks that have to be done. Each task is executed with a certain frequency. For each task and for each system there will correspond a program written with that system instruction set.

The frequency of execution of each task is given by the number of times (n), that this task is executed in a sample

of N tasks. So the frequency of execution of each task is nothing more than the probability of this task being in execution at any given time.

$$F_i = \frac{n}{N} \tag{4.3}$$

where

F_i - is the frequency of execution of task i

n - number of times the task i was executed in a big sample

N - total number of tasks that were executed in that sample

For each task there is a corresponding computer program. This program will take some time to execute.

The weight of each task or its representation in the application will be given by the product of its execution frequency and its program execution time in the system under study.

$$W_i = F_i \times T_i \tag{4.4}$$

where

W_i - weight of the task i in the particular application and for the system in study

T_i - execution time of the correspondent program

By this it is seen that the weight of the task is both dependent on the application choice and on the system choice.

A program is a sequence of instructions. Its execution time can be divided into smaller pieces where only one instruction is executed. In this way the program execution time is given by a sum of products. Each element of the sum will be referred to a single instruction, and consists of

the product of the instruction execution time and the number of times each instruction is executed.

Therefore each element of the sum will be given by:

$$S_j = N_j * IXT_j \qquad (4.5)$$

where

N_j - is the number of times that the instruction j is executed for the particular program

IXT_j - execution time of instruction j

The program execution time will be given by:

$$T_i = \sum_{j=1}^{J} S_j \qquad (4.6)$$

where

S_j - the weight of instruction j in the system instruction set and for the particular task

J - the total number of instructions in the system instruction set

Finally, the weight of the application for the system under study will be given by the weighted sum of its tasks.

So,

$$W_a = \sum_{i=1}^{I} W_i \qquad (4.7)$$

but since

$$W_i = F_i * T_i \qquad (4.4)$$

then

$$W_a = \sum_{i=1}^{I} F_i * T_i \qquad (4.8)$$

But

$$T_i = \sum_{j=1}^{J} S_j \qquad (4.6)$$

and

$$S_j = N_j * I \times T_j \qquad (4.5)$$

So,

$$W_a = \sum_{i=1}^{I} F_i \sum_{j=1}^{J} S_j = \sum_{i=1}^{I} F_i \sum_{j=1}^{J} N_j \, I \times T_j \qquad (4.9)$$

D. THE PERFORMANCE

A comparison is made between the weights that an application has in two different systems. In this chapter, where a timing analysis is done, the weight of an application involves the execution time of each instruction and the dynamic frequency of execution of the same instructions.

The performance will be given by the ratio of these two weights.

$$Perf = \frac{W_a}{W'_a} \qquad (4.10)$$

where

W_a - is the weight of the particular application for the reference system

W_a' - is the weight of the same application for the system being considered

Note that the two systems either have two different instruction sets or the time of execution of each instruction is different or both.

So,

$$W_a' = \sum_{i=1}^{I} F_i \sum_{k=1}^{K} N_k \; IxT_k \qquad (4.11)$$

Therefore

$$Perf = \frac{\sum_{i=1}^{H} F_i \sum_{j=1}^{J} N_j \; IxT_j}{\sum_{i=1}^{I} F_i \sum_{k=1}^{K} N_k \; IxT_k} \qquad (4.12)$$

where

I - is the total number of tasks in the particular application. It is the same as the number of programs.

J - is the total number of instructions in the reference system instruction set

K - is the total number of instructions in the system in study instruction set

Considered in this way the measure of the performance for a system is better the larger the ratio.

E. A SPECIAL CASE AND THE RISC

If the application involves only one task and therefore only one program, the performance would be given by,

$$Perf = \frac{\sum_{j=1}^{J} N_j \, IxT_j}{\sum_{k=1}^{K} N_k \, IxT_k} \qquad (4.13)$$

Let us now consider the RISC philosophy. For this case the value of J is fixed.

The RISC proponents advocate that by reducing the total number of instructions in the instruction set i.e., by reducing the value of K, the performance of the system inceases. They also advocate that the instruction execution time for each instruction is reduced by having a simpler, more straightforward machine with better performance.

Their argument is that the value of the denominator is reduced because the two previous factors compensate for the necessary increase in the number of times each instruction is executed. By reducing the denominator the system will have a better performance.

F. THE SYSTEM ARCHITECTURE AND TIMING

As has just been seen, the particular choice of application determines the dynamic frequency of execution of each instruction in the instruction set. To continue the study, there is now a need to analyze how the system architectural characteristics influence the system performance.

The system structure and its instruction set are necessarily related. For every instruction, the system has to have the necessary support in terms of the control unit and the datapath. Also, any new enhancement to the system architecture will affect the execution time of one or more instructions. Therefore it will always affect the average instruction execution time.

The model under discussion considers that each instruction has a certain associated weight, this weight being dependent on the application and on the system architecture. The application determines the number of times each instruction is executed, i.e., the dynamic frequency of execution of the instruction. The system architecture determines the execution time of each instruction. It is this execution time that will now be studied.

We define the Life Cycle of an instruction (LC) as the time period beginning at the instant the instruction is first fetched from memory and ending at the instant the final results produced by the operation are stored back in memory.

The instruction execution time will then be some portion of its time life cycle. This portion will be dependent on the system architectural characteristics such as pipelining, parallel processing, instruction prefetching, instruction queue, etc.

The main phases through which an instruction has to pass in its life cycle are:

i) Fetching

ii) Execution

The time the system takes to fetch an instruction is dependent on the instruction bus width, the instruction length and the bus cycle time in the following way:

$$t_f = \frac{\text{INSTRUCTION LENGTH}}{\text{BUS WIDTH}} \times (\text{BUS CYCLETIME}) \quad (4.14)$$

This value for the fetch time will be an average, more or less rigorous, depending on:

i) instruction size (fixed or variable)

ii) the availability of the instruction queue

Not all the instructions have the same structure, but nevertheless, all of the instructions accomplish some transformation on some data. The data might be one or more operands and the final result in the case of an arithmetic instruction, or the data might be the contents of the program counter in the case of a branch.

In order for the system to be able to accomplish the transformation required by the instruction, it has to:

1) decode the instruction

2) locate the data (e.g., addressing modes)

3) place the data in a convenient location to be transformed, if it is not there already

4) perform the transformation asked for by the instruction

5) relocate the data in a convenient location.

Whether these phases are performed in a sequential fashion or in parallel depends on the system architecture. For example, suppose that the instructions followed a fixed format with separate and predefined fields for OPCODE and ADDRESSSING. Then it would be possible to decode the instruction and the address field simultaneously.

In order for the system to process the addressing mode and depending on the particular address mode, it may have to do one or more of the following:

- preform data transfers either register-to-register or memory-to-register;
- preform some addition e.g., in the case of base addressing, index addressing or branch addressing;
- preform some multiplication e.g., in the case of the VAX-11 index mode.

For the sake of simplicity one could consider all the data transfers that have to be done while the system

executes a program and determine an average time for data transfer.

Typically if the system has on-chip registers, cache memory and main memory, the value for the average data transfer time will be:

$$t_{DT} = \frac{R}{T}(RAT) + \frac{C}{T}(CAT) + \frac{M}{T}(MAT) \qquad (4.15)$$

where

R - number of register accesses
C - number of cache accesses
M - number of main memory accesses
T - total number of data transfers
RAT - register access time
CAT - cache access time
MAT - memory access time

and

$$T = R + C + M \qquad (4.16)$$

In summary, in the instruction life cycle one has:

TF - fetching time
TDEC - decoding time
TLOC - locating data (address mode)
TDATA - access data
TOP - perform the operation
TW - write the final results

If the system performs all of these time phases in a sequential fashion so that there is no overlap, then the instruction time life cycle will just be the summation of all the time phases:

$$LCno = TF+TDEC+TLOC+TDATA+TOP+TW \quad \text{(no overlap)} \quad (4.17)$$

If some overlap among the phases is present, then the instruction time life cycle will be some portion of the previous value (no overlap case).

$$LCo = y * LCno \quad \text{(overlap case)} \quad (4.18)$$

where

y - is a coefficient that measures the efficiency of the architectural scheme that accounts for the overlap possibility. Its value will be always between zero and one.

Some of the architectural characteristics that might influence the value of " y "are:

- separate or common memories for data and instructions,
- instruction format
- instruction type
- bus width
- dual port memories

The architectural characteristics will also determine the amount of overlap execution among different instructions. The efficiency of this overlap will then determine what portion of the instruction time life cycle value will be the instruction execution time (IXT).

$$IXT = w * LCo \quad (4.19)$$

where

IXT - instruction execution time

w - efficiency of the overlap among the time life cycles of different instructions. Values ranging from zero to one.

The value of w, that is the amount of overlap will be determined by several architectural characteristics such as:

- pipelining
- prefetching
- instruction queue
- parallel processing
- instruction length
- bus width
- datapath

V. CONTROL ANALYSIS

A. INTRODUCTION

In the previous chapters a timing analysis of the system operation was presented. In it a study was made first of the application effects on performance through the dynamic frequency of execution of each instruction, and second of the system architecture effects on performance through the execution time of each instruction.

Finally to complete the model being suggested, one has to consider the requirements that the instruction set poses on the system in terms of the required control complexity.

These requirements will also be dependent on the application.

This is also important since no matter what technology is used in the system implementation, the number of resources available on-chip will always be limited.

Typically the control unit is implemented using either microcode or is hardwired e.g., using programmable logic arrays. Some of the factors that impact the choice are:

- instruction set complexity
- required control unit size
- possibility of future changes in the instruction set
- speed

The size of the control unit (i.e., the number of gates needed to implement the control unit) will determine the space available on-chip for other components. In the case of the RISC I and II the smaller control unit and therefore the smaller power consumption, allowed the designers to add more registers to the processor chip. With the choice of additional hardware for the processor , the designers in fact reduce the average memory access time if one considers the registers as also part of the system memory.

B. THE CONTROL UNIT AS A FINITE STATE MACHINE

The control unit of a computer system can be viewed as a finite state machine, and therefore can be analyzed as such. If analyzed in that way, the control unit operation can be described by a state diagram. In its most simple and most general case, the state diagram will typically have only two states, see Figure 5.1.

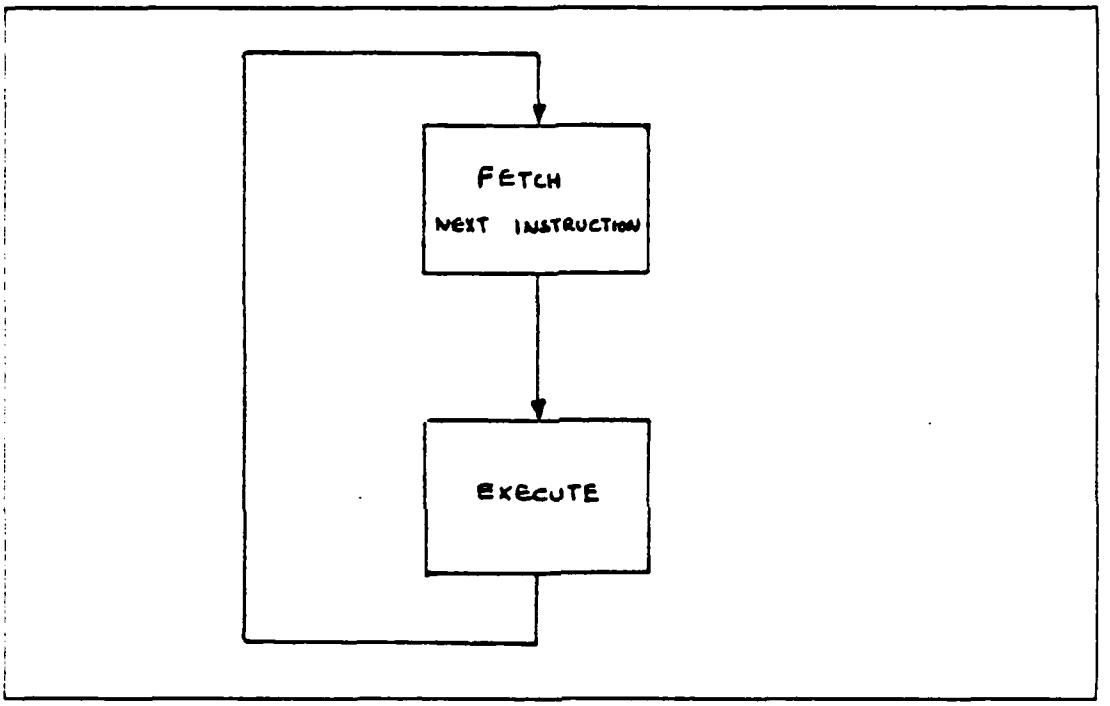


Figure 5.1 Simple Control Unit State Diagram.

In a more detailed analysis, the control unit state diagram will have a tree like format where any vertical path will correspond to the execution of an instruction, see Figure 5.2.

In this case, each and every instruction is identified and each state although, still belonging to one of the two major phases fetch and execute, will now correspond to a microstep in the control unit output sequence while the system is executing a program.

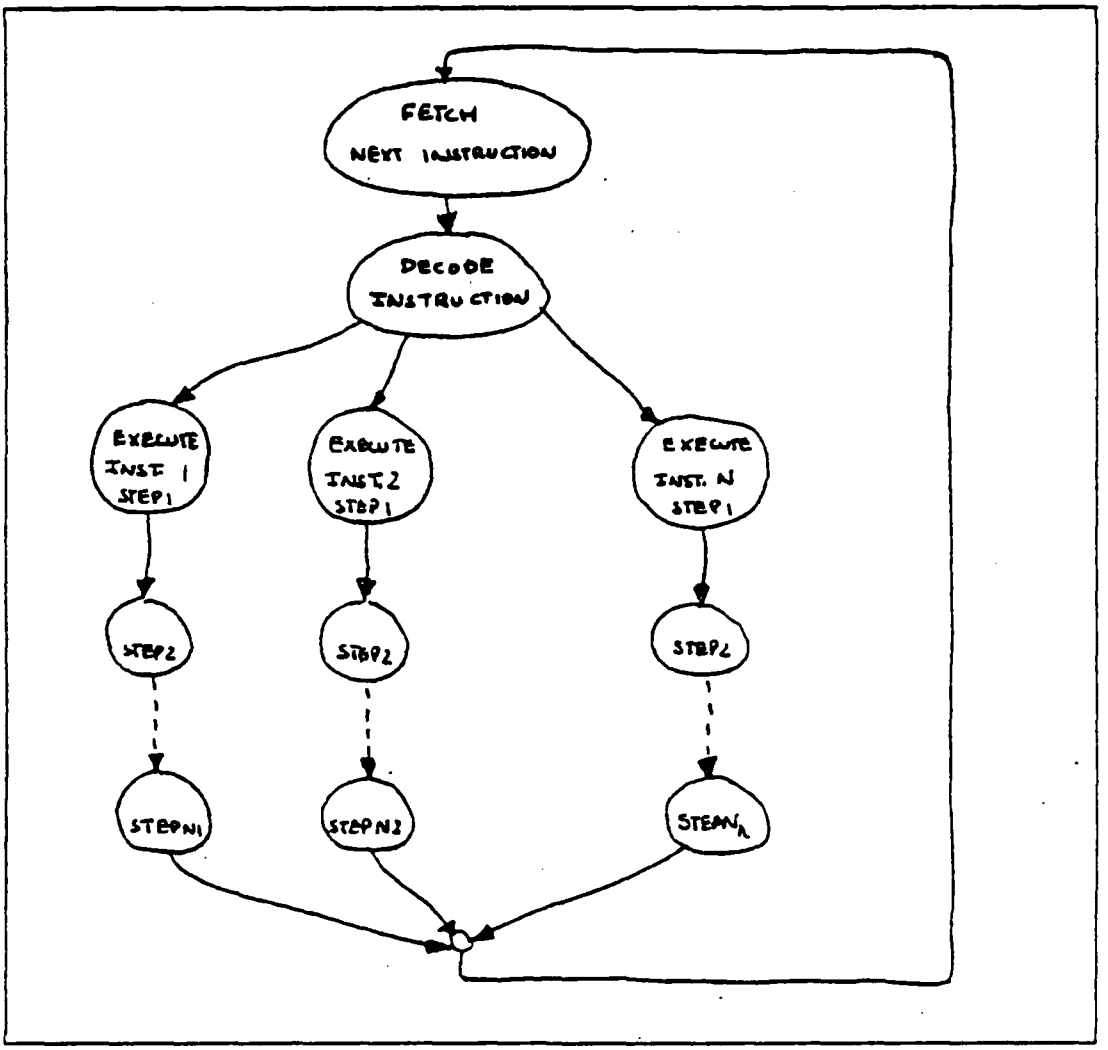


Figure 5.2 More Detailed Control Unit State Diagram.

Of course this is complicated if the system is able to deal with more than one instruction at a time. Nevertheless the complexity of the controller can always be associated with the number of states.

C. THE CONTROL UNIT COMPLEXITY

Not all the states will count in the same fashion since there are states that will be common to more than one instruction or vertical path.

The number of these shared states will depend both on the processor instruction set itself and on the implementation choices made by the processor designer. For example, in this last case the processor designer could make use of microcode subroutines to be shared or called by more than one instruction.

If states are shared among instructions, then there will always be some trade-off between the total number of states of the control unit and its speed. This tradeoff is due to the fact that when states are shared among different instructions, the control unit has to have some feedback capability. The specific value of the feedback will force the next state of the control unit, when the vertical paths corresponding to the instructions will ultimately separate themselves.

No matter what this feedback will be, it will always have some cost related to it. The cost is the extra time it takes for the values of the feedback signals to be valid. Since the cost is time, it will be reflected in the average instruction execution time, and so affect the performance of the system in the portion the model described in the previous chapter.

In this part of the model we focus on the comparisons of two control units.

The complexity of a particular instruction will then be dependent both on the number of states it has and on the number of states which are shared by more than one instruction.

The cost of adding a new instruction to a certain processor instruction set is the number of new states that have to be added to the control unit state diagram. The addition of this instruction will have a cost on the system performance that can be minimized by maximizing the number of states necessary to its execution that are already in existence in the control unit state diagram.

Returning to the control unit the number of states is then dependent on:

i) the number of instructions

ii) the number of states that are common to more than one vertical path (or instruction)

iii) the average height of each instruction

Where the height of one instruction is defined as the number of states in its vertical path.

D. THE APPLICATION AND THE CONTROL UNIT

In the previous chapter the instruction set and the dynamic frequency of execution of each instruction together with the instruction execution time were considered. Now one wants to know how effective, the control unit is for the application where the processor is being used.

It has already been seen that the complexity of the control unit is related to the number of states. One knows that a smaller and simpler control unit has an effect on the processor performance, because more space would be available on-chip for other resources. One choice might be to add new registers to the processor chip and thus try to decrease the average memory access time.

One also wants to minimize the number of instructions that are needed in order to perform a certain task, so one has to go back to the application. An application is characterized by a certain number of tasks that have to be done. Each task is performed with a certain frequency. For each task a program will have to be written using the instruction set available. Each program corresponds to a sequence of instructions used to perform the corresponding task.

Directly from the program it should be possible to compute the static frequency of each instruction. But that is not the only frequency that is of interest to the performance evaluation process. The dynamic frequency of execution is more important.

The two frequencies will be different for each instruction depending on:

i) program sequence

ii) conditional branches and the most frequent values of the variables on condition.

The execution of a program is then a sequence of several instructions execution.

Since a single instruction corresponds to a vertical path in the processor control unit state diagram, the execution of a program will then be an up and down walk on the state diagram.

When comparing two control units, the one that would have to execute fewer instructions, supposing that the average height of an instruction would be the same for both control units, will be the best. The height of an instruction is in fact a measure of what the RISC proponents call the instruction complexity. Because it would be natural that two different processors have instruction sets with different values for the average height of an instruction, the bottom line is that the comparison of two control units' complexity cannot be done through the counting of instructions executed, but through the counting of the number of states through which each control unit has to pass when the system executes a typical application program.

It is to be expected that if one wants to add an instruction to a processor instruction set, the control unit will suffer by an expansion. For a hardwired implementation e.g., using PLA's these will have to grow; for a microcode implementation typically there will be a need to increase the size of the microcode memory. The amount of the control unit expansion will be dependent on the implementation, on the instruction itself, and on the designer's choice regarding the number of states that will be shared with existing instructions. There is a relation between the number of gates used in order to implement a controller and

the number of states present on the controller state diagram.

Because there is a direct and individual relation between the control unit states and the gates that compose the control unit, and because one wishes to use each and every one of these gates a similar number of times in order to increase the overall efficiency, then for better efficiency it is desirable that all states are used in a balanced way. With some similarity one might say that the efficiency of the use of an instruction set increases when all the instructions in that instruction set tend to be used an equal number of times.

An application has an indirect relation to the number of states through which the control unit has to pass in order for the system to execute the corresponding programs.

In the optimum case the control unit will have the following characteristics:

i) minimum number of gates

ii) for the specific application all states will be used in a balanced number of times

iii) no state exists that will never be used.

E. THE MODEL

Assume that a control unit has a total number of states T. Associated with each state there will be a certain number of gates. This number will be dependent on the implementation choice, either microcode or hardwired logic. Of these T states, an application uses S states, and of these S states some states will be used more than others.

The weight of the application is related to the number of states through which the control unit has to pass in order to execute the corresponding programs.

Each state has some weight associated with it. This weight will be dependent on:

i) the number of times the state is used

ii) the number of instructions that share the state

iii) the number of gates needed for implementing each state.

The complexity of an instruction will be related to its height, that is the number of states in the corresponding vertical path in the control unit state diagram.

So,

$$C_j = \sum_{h=1}^{H} W_h \qquad (5.1)$$

where

C_j - complexity of the instruction j

W_h - weight of state h

H - height of the instruction j

and

$$W_h = \frac{G}{U_h} \qquad (5.2)$$

G - number of gates per state (implementation)

U_h - number of instructions to which the state is common

The weight of an instruction will be the product of the number of times the instruction is executed for a given program times the instruction complexity.

That is

$$W_j = N_j \times C_j \qquad (5.3)$$

where

N_j - number of times the instruction j is executed

As in the previous chapter, the weights of the task and the application will be:

$$W_i = F_i \sum_{j=1}^{I} N_j \times C_j \qquad (5.4)$$

where

W_i - weight of task i

F_i - frequency of task i for a certain application

J - number of instructions in the instruction set

For an application its weight will be:

$$W_a = \sum_{i=1}^{I} F_i \sum_{j=1}^{J} N_j \times C_j \qquad (5.5)$$

or

$$W_a = \sum_{i=1}^{I} F_i \sum_{j=1}^{J} N_j \sum_{k=1}^{H} \frac{G}{U_k} \qquad (5.6)$$

where

W_a - weight of the application

I - number of tasks in the application of interest

J - number of instructions in the processor instruction set

H - height of each instruction

Similar to the timing analysis in the previous chapter, the performance of the system under study will be given by:

$$Perf = \frac{W_a}{W'_a} \qquad (5.7)$$

where

W_a - weight of the application for the reference system

W'_a - weight of the same application for the system being considered

So,

$$Perf = \frac{\sum_{i=1}^{I} F_i \sum_{j=1}^{J} N_j \sum_{h=1}^{H} \frac{G0}{U_h}}{\sum_{i=1}^{I} F_i \sum_{k=1}^{K} N_k \sum_{l=1}^{L} \frac{G1}{U_l}} \qquad (5.8)$$

where

- I - number of tasks (programs) in the application
- J - number of instructions in the reference system instruction set
- K - number of instructions in the system under study instruction set
- H - height of instruction j in the reference system instruction set
- L - height of instruction k in the system under study instruction set
- N_j - number of times instruction j is executed while the reference system executes the typical application program
- N_k - number of times the instruction k is executed while the system under study executes the same program
- $G0$ - number of gates per state in the reference system control unit
- $G1$ - number of gates per state in the system under study control unit
- U_h - number of instructions that share state h in the reference system control unit state diagram

U_l - number of instructions that share state l in the system under study control unit state diagram.

VI. CASE ANALYSIS

A. INTRODUCTION

As an example we will analyze the change in performance of a particular application program when some floating point capability is added to a processor which currently performs fixed point arithmetic.

In this case study, the performance effects of the program code sequence will not be considered. These effects are mostly due to any capability of the processor related to:

- pipelining
- parallel processing

Specifically, the case consists in the possible addition of a floating point multiply instruction to a processor instruction set. The processor that was chosen was the Motorola MC68000. The application for this evaluation is the computation of a Fast Fourier Transform.

B. THE ADDITION OF AN INSTRUCTION

The addition of an instruction to the original instruction set has several consequences.

First of all if a hardwired controller is used the processor's control unit must be expanded so that the instruction is incorporated. The amount of the control unit expansion is dependent on the number of new states that the instruction under consideration will add to the control unit state diagram and also on the control unit implementation.

In fact, one of the reasons to use microcode in the implementation of an instruction set is due to the flexibility it gives in any future changes of the instruction set.

Second and depending on the operation performed by the instruction, some hardware will have to be added to the processor. The amount of hardware that will have to be added to the processor is dependent both on the hardware that already exists on-chip, that the instruction might use and is dependent also on how fast one wants the instruction to operate.

The addition of more hardware to the processor will cause a rise in the power consumed by the processor. Due to a limited power dissipation capability, the net effect of the increase in the number of gates that constitute the control unit and the datapath will be a reduction in the size of existing processor components or a migration of some off-chip, so that the power consumed by the processor stays constant.

One choice might be to replace some of the registers available on-chip by the hardware necessary for the new instruction. By reducing the number of registers on-chip, there will be a decrease in the ratio of register accesses to the number of main memory accesses.

In the case of a Load/Store architecture such as the RISC architecture, a reduction in the number of registers will cause an increase in the dynamic frequency of execution of LOAD and STORE instructions relative to the other instructions.

In a traditional architecture, where the LOAD and STORE instructions are not the only memory reference instructions, the effect of reducing the number of on-chip registers is an increase in the average instruction execution time because the proportion of memory accesses to register accesses will increase.

This increase in average instruction execution time will cause an increase in the typical application's program execution time. It is this increase in execution time, that

will have to be overcome by the addition of the new instruction to the processor instruction set, so that in fact the program execution time might suffer a reduction rather than an increase.

C. THE COST/GAIN TRADEOFF

The floating point multiply instruction after being added to the processor instruction set, will replace the sequence of instructions that the processor had to execute every time a multiplication of two floating point numbers was called for.

In order for the addition of the floating point multiply instruction to be considered, the instruction has to pass several tests. The first test requires the instruction execution time to be smaller than the correspondent instruction sequence execution time.

If that is not the case, then there is no point in adding the instruction to the processor instruction set.

So, consider:

lni - execution time of the new instruction

lseq - execution time of the corresponding sequence of instructions

For the addition of the new instruction to be considered:

$$lni < lseq \qquad (6.1)$$

Assume then that in fact the above condition is true, then

$$lseq = lni + lgain \qquad (6.2)$$

or

$$lni \;/\; lseq = c \qquad (6.3)$$

where $c < 1$

For the sake of simplicity, consider that the application of interest is composed of only one task. That is to say that the effects on the processor performance will be considered only within the context of a program.

The model suggested for computer performance evaluation has two parts, a timing analysis and a control unit complexity analysis. These two parts of the model will give rise to two distinct criteria to which the addition of the instruction will have to comply. So that the gain in the processor performance that is obtained, will surpass the reduction or cost in the processor performance due to the requirements brought by the same instruction to the processor architecture.

1. Timing Criterion

The timing model says that the effects of the addition of one instruction to the system instruction set, on the system performance will be measured by:

$$Perf = \frac{\sum_{j=1}^{J} N_j L_j}{\sum_{j=1}^{J} Na_j La_j + Nnew\ Lnew} \qquad (6.4)$$

where

J - is the number of instructions on the original system instruction set

N_j - number of times that the instruction j is executed before the addition of the new instruction to the processor instruction set

L_j - execution time of the same instruction j on the original system

Na_j - number of times that the instruction j is executed after the addition of the new instruction

La_j - execution time of the instruction after the addition of the new instruction

N_{new} - number of times the new instruction is executed

L_{new} - execution time of the new instruction

The numerator is a measure of the execution time of the application program before the addition of the instruction under consideration. The denominator is a measure of the execution time of the application program after the addition of the new instruction.

The sequence of instructions in the original instruction set that implements the operation performed by the new instruction is executed a number of times. This number will be equal to Nnew.

The sequence execution time will consist of the execution time of several instructions.

Therefore

$$L_{seq} = \sum_{j=1}^{J} N_{seq_j} L_j \qquad (6.5)$$

where

N_{seq_j} - number of times that the instruction j of the original instruction set is executed during the sequence of instructions execution.

then

$$Perf = \frac{\sum_{j=1}^{J} No_j L_j + Nnew \sum_{j=1}^{J} Nseq_j L_j}{\sum_{j=1}^{J} Na_j La_j + Nnew\ Lnew} \qquad (6.6)$$

and

$$N_j = No_j + Nnew\ Nseq_j \qquad (6.7)$$

where

No_j - number of times the instruction j of the original instruction set is executed outside the sequence.

For improvement in performance:

$$Perf > 1 \qquad (6.8)$$

This indicates that it is worthwhile to add the new instruction to the original instruction set for this application.

Then, one wants

$$\sum_{j=1}^{I} No_j L_j + L new \sum_{j=1}^{J} Nseq_j L_j > \sum_{j=1}^{J} Na_j La_j + Nnew\ Lnew \qquad (6.9)$$

but

$$l_{seq} = \sum_{j=1}^{J} N_{seq_j} L_j \qquad (6.5)$$

so

$$\sum_{j=1}^{J} N_{o_j} L_j + N_{new} l_{seq} > \sum_{j=1}^{J} N_{a_j} L_{a_j} + N_{new} L_{new} \qquad (6.10)$$

$$N_{new} (l_{seq} - L_{new}) > \sum_{j=1}^{J} N_{a_j} L_{a_j} - \sum_{j=1}^{J} N_{o_j} L_j \qquad (6.11)$$

The right term of the inequality corresponds to the increase in the application program execution time, that was caused by the suppression of some hardware components of the processor e.g., some registers.

This increase, caused by an increase in the number of instructions that have to be performed--case of the LOAD and STORE instructions in a Load/Store architecture, or caused by an increase on the average instruction execution time--case of a traditional architecture.

Therefore

$$\sum_{j=1}^{J} N_{a_j} L_{a_j} - \sum_{j=1}^{J} N_{o_j} L_j = \text{TIMING COSTS} = T_{cost} \qquad (6.12)$$

On the left term of equation 6.7,

Lseq - Lnew

represents the gain in execution time that was obtained by substituting the sequence of original instructions by the new instruction, each time the operation was performed.

So,

$$Lseq - Lnew = Timing\ Gains = Tgain \qquad (6.13)$$

Then,

$$Nnew\ Tgain > Tcost \qquad (6.14)$$

or

$$Nnew > Tcost / Tgain \qquad (6.15)$$

Based on an timing analysis, it is only advantageous to add the new instruction if:

1) $Lseq > Lnew$ (6.16)

and

2) $Nnew > Tcost / Tgain$ (6.17)

To put it in another way, the addition of an instruction to a processor instruction set will only increase performance if that instruction is executed a

sufficient number of times during the application programs execution. The exact number of times the instruction must be executed is given by the above criterion.

2. Control Unit Complexity Criterion

Concerning the analysis of the control unit complexity one has:

$$Perf = \frac{\sum_{j=1}^{J} N_j \sum_{h=1}^{H} \frac{Go}{U_h}}{\sum_{j=1}^{J} Na_j \sum_{h=1}^{H} \frac{Go}{U_h} + Nnew \sum_{h=1}^{Hnew} \frac{Go}{U_h}} \qquad (6.18)$$

Since the implementation of the control unit will be the same and the implementation determines the value of GO, the equation simplifies to,

$$Perf = \frac{\sum_{j=1}^{J} N_j \sum_{h=1}^{H} \frac{1}{U_h}}{\sum_{j=1}^{J} Na_j \sum_{h=1}^{H} \frac{1}{U_h} + Nnew \sum_{h=1}^{Hnew} \frac{1}{U_h}} \qquad (6.19)$$

As in the timing analysis one wants:

$$Perf > 1 \qquad (6.20)$$

That is

$$\sum_{j=1}^{J} N_j \sum_{h=1}^{H} \frac{1}{U_h} > \sum_{j=1}^{J} Na_j \sum_{h=1}^{H} \frac{1}{U_h} + N_{new} \sum_{h=1}^{H_{new}} \frac{1}{U_h} \qquad (6.21)$$

As before, the execution of the sequence will consist on the execution of several instructions, then

$$\sum_{j=1}^{J} N_j \sum_{h=1}^{H} \frac{1}{U_h} = \sum_{j=1}^{J} No_j \sum_{h=1}^{H} \frac{1}{U_h} + N_{new} \sum_{j=1}^{J} Nseq_j \sum_{h=1}^{H} \frac{1}{U_h} \qquad (6.22)$$

Then

$$\sum_{j=1}^{J} No_j \sum_{h=1}^{H} \frac{1}{U_h} + N_{new} \sum_{j=1}^{J} Nseq_j \sum_{h=1}^{H} \frac{1}{U_h} >$$

$$> \sum_{j=1}^{J} Na_j \sum_{h=1}^{H} \frac{1}{U_h} + N_{new} \sum_{h=1}^{H_{new}} \frac{1}{U_h} \qquad (6.23)$$

or

$$N_{new} \underbrace{\left\{ \sum_{j=1}^{J} Nseq_j \sum_{h=1}^{H} \frac{1}{U_h} - \sum_{h=1}^{H_{new}} \frac{1}{U_h} \right\}}_{L_s} > \underbrace{\sum_{j=1}^{J} (Na_j - No_j) \sum_{h=1}^{H} \frac{1}{U_h}}_{E_s} \qquad (6.24)$$

where

Ls - represents the gain in the number of states, obtained each time the operation performed by the instruction and/or the sequence is executed.

Es - represents the cost in the number of states due to the addition of the new instruction

Then

$$Nnew * Ls > Es \qquad (6.25)$$

or

$$Nnew > Es / Ls \qquad (6.26)$$

D. AN ILLUSTRATIVE EXAMPLE

An example is now presented to clarify the use of the model suggested through the present and previous chapters.

The example quantizes the effects of adding a floating point multiply instruction to an existing processor instruction set.

As has been previously stated, the values determined for the increase or decrease on the system performance will only be valid for a given application.

1. The Processor

The Motorola MC68000 is selected for this example. The MC68000 is a widely known microprocessor that has a simple instruction set offering no floating point support.

The MC68000 has a 16-bit data bus and a 32-bit address bus. In addition to the Program Counter and Status Registers, the MC68000 has seventeen 32-bit registers. These registers are divided into two groups. The first group,

composed of eight registers are general purpose data registers. The second group, composed of the remaining nine registers is used mostly for handling addresses.

In total, there are fourteen addressing modes on the MC68000, although they can be studied in six basic types. These addressing modes are already described in chapter two of this thesis.

The instruction set of the MC68000 consists of 56 basic instructions, having from zero to two addresses. Each instruction can use several addressing modes. This fact determines that the MC68000 does not follow a Load/Store architecture.

The instruction set of the MC68000 supports five basic types of data:

- bits
- bytes (8 bits)
- words (16 bits)
- longwords (32 bits)
- Packed binary-coded decimal (BCD) with two digits per byte

The input/output on the MC68000 is memory-mapped, i.e., all I/O interfaces share the address space with memory.

Considering the implementation of the MC68000, it is a single-chip VLSI HMOS processor with a typical clock rate between 4 and 12 MHZ and with a typical memory access of 4 clock cycles.

2. The Application

For the application we choose a program that computes a Fast Fourier Transform. This program was obtained from ' The Fast Fourier Transform' by E. Oran Brigham [Ref. 4]. The program is written in Fortran. The flowchart of the computation done by this program is on page 161 of the above reference. The program itself appears on page 164 of the same book.

From the reading of the program, one can immediately verify that some of the operations that are called for could not be directly implemented with the MC68000 instruction set.

For these operations it was necessary to use either subroutines present in ' Microprocessor Systems, a 16-Bit Approach' by William J. Eccles [Ref. 5] or newly written subroutines. The subroutines to handle floating point numbers in the MC68000 came from Ref. 5.

The subroutines that were written are shown on appendixes C and D, these subroutines compute the sine and the cosine of an angle, according to an algorithm presented in the ' Software Manual of the Elementary Functions' by William J. Cody, J.R. and William Waite [Ref. 6:pp. 125-143].

The translated program for the Fast Fourier Transform computation is shown on Appendixes A and B.

3. The Floating Point Representation

The floating point representation that was chosen is the IEEE proposed standard for single precision. This standard determines a 32-bit long representation of a floating point number, shown in Figure 6.1.

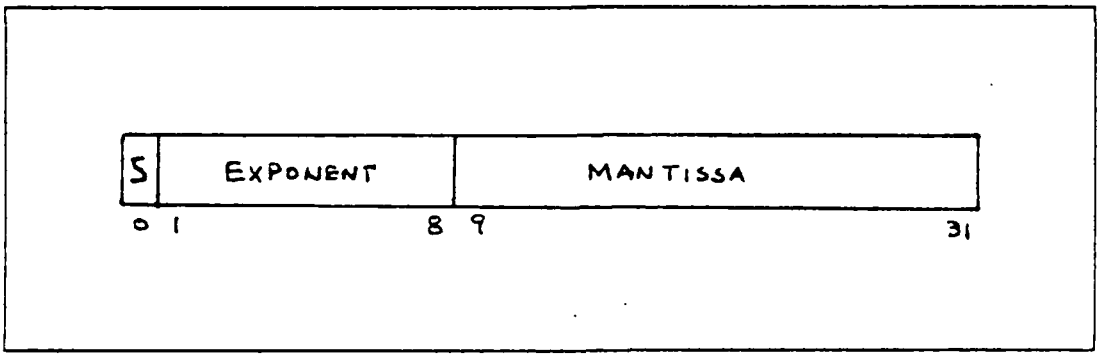


Figure 6.1 Floating Point Representation.

This standard has the following characteristics:

i) 32 bits are used

ii) radix of two

iii) the radix point before the first digit with assumed one to the left

iv) mantissa

iv.a) sign position - 0

iv.b) value position - 9-31

iv.c) representation - normalized, sign/magnitude

v) exponent

v.a) sign position - no sign

v.b) value position - 1-8

v.c) representation - biased exponent, bias = 127(dec)

v.d) range of exponent - -126 to 127

vi) range of floating point number - +- 5.9*10**-39 to +-1.7*10**38

All the subroutines that handle the floating point data and that were used obey to this standard, so does the hardware necessary to implement the floating point multiply.

4. The Hardware Involved

The general structure of the hardware required for the implementation of an additional floating point multiply instruction in the MC68000 instruction set was obtained from the 'Introduction to Computer Architecture' [Ref. 7:p. 80] and is shown on Figure 6.2.

The hardware consists of:

i) three 32-bit registers, these can be some of the already existing data registers on the MC68000,

ii) an 8-bit adder used for the exponent addition, that could just be the adder already existing on the MC68000,

iii) a multiplier used for the mantissa multiplication,

iv) an exclusive-or gate for the product sign calculation,

v) a normalizer and converter

With the hardware structure that was chosen it is possible to perform in parallel the determination of the sign of the result, the addition of the two exponents, and the multiplication of the two mantissas.

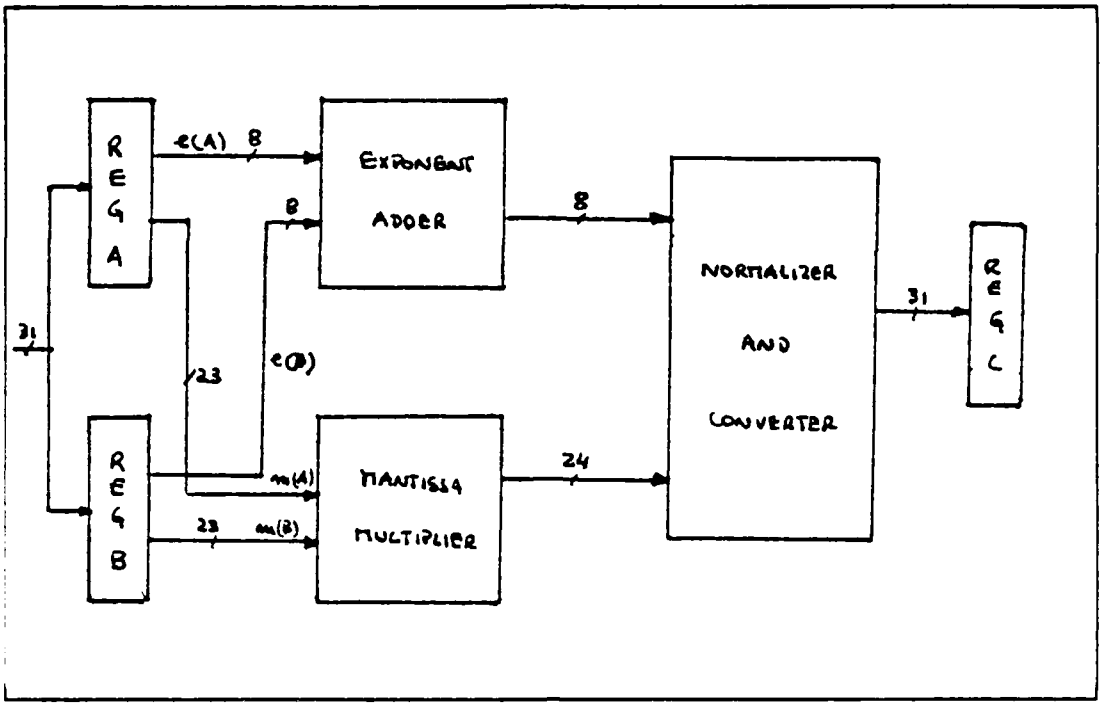


Figure 6.2 General Hardware Structure for the Floating Point Multiply Instruction.

The execution time of the floating point multiplication instruction will then be determined by the slowest of these three distinct and parallel operations.

The sign computation involves just one exclusive-or gate gate and therefore takes a maximum of one clock cycle.

The addition of the two exponents involves in fact the addition of the two exponents, followed by the subtraction of the bias since this has also to be performed concurrently with the determination of exponent overflow or underflow.

From [Ref. 7] the addition of the contents of two registers using the MC68000, takes 4 clock cycles to complete. After this addition an extra clock cycle will be taken for the determination of exponent overflow and underflow together with the subtraction of the extra bias.

Therefore it is concluded that the addition of the two exponents will take a maximum of 5 clock cycles.

For the mantissas multiplication, a multiplier will have to be added to the processor hardware. According to "Digital Systems: Hardware Organization and Design by Frederick J. Hill and Gerald R. Peterson ' [Ref. 8] the multiplier structure that gives the best cost/performance tradeoff in terms of the hardware involved and the time it takes to perform a multiplication is a multiplier that uses a carry-save adder. There a carry save adder type multiplier was chosen.

Also, according to [Ref. 8:p. 361] the time that a carry-save adder takes to perform an N-bit multiplication using a adder for which each addition/shift cycle takes two clock cycles is given by:

$$Tmult = (N+1)Tc \qquad (6.27)$$

where

Tc - is the clock cycle time

In the case being discussed the multiplication involves two operands - the mantissas. Each mantissa is 24-bits long. Therefore according to the formula shown above, the multiplication of the two mantissas will take 25 clock cycles. This makes the the multiplication the longest operation involved.

Note that, the detection of a zero product can be done concurrently with the multiplication, since a zero product will happen only in the case where one of the operands is zero.

The normalization must still be done sequentially. The normalization involves at most one left shift of the mantissa product and a decrement of the product exponent.

There is only at most one shift, since the mantissas of both operands are in normalized form and therefore their values are between 0.5 and 1. In the worst case, the two mantissas are both 0.1 (binary) and so their product will be 0.01 (binary). In this case only one left shift is necessary in order to normalize the mantissa of the product.

The normalization requirement that the standard makes on the mantissa, also dictates that any overflow or underflow of the exponent product does not have a possible recovery.

In conclusion, the floating point multiply instruction with this hardware will take approximately 26 clocks to complete.

The hardware that would have to be added to the MC68000 would only consist of the 24 bit carry-save adder, the exclusive-or gate and some logic to determine overflow or underflow of the exponent and a zero product.

All this hardware will be more or less equivalent to two of the 32-bit registers existing on the MC68000. Say then, that due to power dissipation limitations on the MC68000 two of the 32-bit data registers would then be removed from the MC68000, in order to add the additional hardware necessary to implement the floating point multiply instruction.

5. The Model

As stated previously, the addition of the instruction will have some costs. One of these costs has been referred in the previous subsection, it is the removal of two of the data registers.

As one might expect the removal of some of the registers from the MC68000 will have an effect on the system performance by reducing the number of registers accesses and increasing the number of main memory accesses.

In the specific case of the application that is being considered, this is not true because, at most, six of the eight data registers are used at one time. Therefore, for this specific case, the timing costs involved due to the addition of the floating point multiply instruction will be zero.

For each and every subroutine involved in this application, the execution time of the subroutine was computed following a worst case and a best case criteria. The difference between the two execution time values for each subroutine arises due to data dependencies on the number of times each instruction is executed.

The execution times of each subroutine were then combined, best with best and worst with worst, in order to define two boundary lines for the final execution time of the whole program.

For the specific case of the floating point multiply subroutine, the smallest execution time corresponds to a multiplication of two floating point numbers where one of them is zero. The longest execution time for the same subroutine corresponds to the multiplication of two numbers where an exponent underflow occurred after the normalization step. Here, for the same reason as before, the normalization requires at most one left shift.

Specifically, the values obtained for the execution times of each subroutine are shown in Table I in terms of clock cycles.

For the whole program the execution time will be dependent on the values of the data and on the number of entry points (N) to the Fast Fourier Transform computation. The values obtained in terms of clock cycles and number of required floating point multiplies are shown in Table II.

The best case and the worst case execution of a floating point multiply subroutine takes respectively 203

TABLE I

EXECUTION TIME OF EACH SUBROUTINE
IN FAST FOURIER TRANSFORM PROGRAM

	BEST CASE	WORST CASE
GETFP	162	162
STFP	180	253
NORM	126	1524
ADDFP	178	1929
MULTFP	203	604
SINE	2681+3MULTFP	14459+9MULTFP
COSINE	3904+3MULTFP	20756+9MULTFP

TABLE II

FAST FOURIER TRANSFORM
APPLICATION PROGRAM EXECUTION TIME

N	BEST CASE	WORST CASE
16	572482+352MULTFP	1899074+736MULTFP
32	1418194+880MULTFP	4734674+1840MULTFP
64	3484658+2112MULTFP	11444210+4416MULTFP
128	8198594+4928MULTFP	26770882+10304MULTFP
256	18901458+11264MULTFP	61352402+23552MULTFP
512	42902562+25344MULTFP	138417186+52992MULTFP
1024	96186226+56320MULTFP	308440946+117760MULTFP
2048	213497794+123904MULTFP	680458178+259072MULTFP
4096	469394450+270336MULTFP	1488217106+565248MULTFP

and 640 clock cycles to execute. For a clock rate of 10 MHZ, the program execution time before the addition of the new instruction will be is in Table III.

TABLE III

FFT PROGRAM EXECUTION TIME BEFORE THE ADDITION OF THE FLOATING POINT MULTIPLY INSTRUCTION

N	BEST EXECUTION TIME (SEC)	WORST EXECUTION TIME (SEC)
16	0.064	0.234
32	0.160	0.584
64	0.391	1.411
128	0.920	3.299
256	2.119	7.558
512	4.805	17.042
1024	10.762	37.957
2048	23.865	83.694
4096	52.427	182.963

For the same clock rate, the program execution time after the addition of the floating point multiply instruction is shown in Table IV.

The best case is the one where the implementation of the floating point multiply offers less gain.

For the best case

Tgain = 203 - 26 = 177 clock cycles

For the worst case

Tgain = 604 - 26 = 578 clock cycles

As already explained, for both cases Tcost is zero. This is due to the fact that in the particular application program two of the general purpose data registers are never used. In the case that all general purpose data registers were used in the application program this would not be true. If this happened then there would be an increase in the ratio of the number of register accesses to the number of

TABLE IV

FFT PROGRAM EXECUTION TIME AFTER THE ADDITION OF THE FLOATING POINT MULTIPLY INSTRUCTION

N	BEST EXECUTION TIME (SEC)	WORST EXECUTION TIME (SEC)
16	0.058	0.192
32	0.144	0.478
64	0.354	1.156
128	0.833	2.704
256	1.919	6.196
512	4.356	13.979
1024	9.765	31.150
2048	21.672	68.719
4096	47.642	150.291

main memory accesses, causing an increase on the average operand access time and an increase on the average instruction execution time.

Using the formula for the model regarding the timing analysis the performance effects of the addition of the floating point multiply instruction come as shown in Table V.

From these results one can see that the improvement on the MC68000 performance due to the addition of the floating point multiply instruction for this specific application varies between ten and twenty percent and is independent of the number of data points to the Fast Fourier Transform computation.

TABLE V

PERFORMANCE EFFECTS OF THE ADDITION OF THE FLOATING POINT MULTIPLY INSTRUCTION

N	BEST CASE Perf	WORST CASE Perf
16	1.11	1.22
32	1.11	1.22
64	1.11	1.22
128	1.11	1.22
256	1.10	1.22
512	1.10	1.22
1024	1.10	1.22
2048	1.10	1.22
4096	1.10	1.22

VII. CONCLUSIONS

This thesis began by making an identification and characterization of a new and controversial type of computer architecture called RISC for Reduced Instruction Set Computers. The rise of this new computer architecture and the discussions that followed regarding its performance, when RISC machines are compared with CISC machines, has shown the need for an appropriate tool to evaluate computer performance from an architectural point of view.

This thesis suggests a model to be used by computer architects to determine the performance effects of an enhancement to a computer architecture. The computer evaluation process is important, since it generates have a quantified perception of the influences that each enhancement to the system architecture will have on the system performance. The availability of a model to do computer performance evaluation is therefore essential in the decision-making process for determining which architectural features a system should have to optimize its performance for a certain application.

To develop this model for the evaluation of computer performance, a conceptual view of what determines the system performance was formed. It is the author's opinion that the performance of a system results from the quality of the match between a particular application requirement and the architectural characteristics of the system. This match is done through the customization of the system instruction set.

The model that is suggested is divided into two parts. The first part makes a quantification of the effects that an architectural enhancement to the system has in the execution time of a "typical" application program. The second part of the model compares the efficiency of the design of two

systems control units. In both parts the model considers that the application determines the number of times each instruction of the system instruction set is executed.

For the first part, the system architecture determines the execution time of each instruction. For the second part, the system architecture determines the number of states through which the system control unit will have to pass during the execution of the application program(s).

Finally, an example on how to use the model, in order to determine what are the costs and benefits of adding an instruction to a processor instruction set for a particular application, is given.

The program that was used to apply the model is a bit misleading in the quantification of the cost/benefit ratio of the enhancement. This is due to the fact that in opposition to what should be expected, the program does not use all the system architectural resources and so, even before the addition of the new, instruction does not optimize the system performance. If that were not the case and the program was an optimal one for the application of interest and for the processor chosen, then, surely, the enhancement to the system architecture would have some costs.

In any event and even considering that the example is a bit misleading, the author arrived at two criteria, each one derived from one of the parts of the model, for which the addition of an instruction to a system instruction set has to obey so that the performance of the system for the particular application is increased.

These two criteria will be applied if the new instruction execution time is smaller than the execution time of the sequence of instructions that implemented the function before the addition of the new instruction to the system.

For the first part of the model the criterion for the addition of the new instruction, states that:

$$Nnew > Tcost / Tgain$$

where

Nnew - is the number of times the new instruction is executed for the particular application

Tgain - is the difference in the execution times of the sequence of instructions that had to be executed by the system every time the operation was performed before the addition of the new instruction and the execution time of the new instruction.

Tcost - is the increase in the application program execution time that was caused by the suppression of some hardware components of the processor

For the second part of the model, the criterion for the addition of the new instruction, states that:

$$Nnew > Es / Ls$$

where

Ls - represents the gain in the number of control unit states, obtained each time the operation performed by the the instruction and/or the sequence is executed.

Es - represents the cost in the number of states due to the addition of the new instruction to the system instruction set.

The two parts of the model need to be thoroughly checked and confirmed with measured values, so that their validity is determined.

APPENDIX A

FAST FOURIER TRANSFORM

```
FFT     MOVE.W    N,N2          ;N2=N/2
        ASR.W     N2            ;
        MOVE.W    NU,NU1        ;NU1=NU-1
        SUBI.W    #1,NU1        ;
        CLR.W     K             ;K=0
        MOVE.W    NU,D0         ;DO  100 L=1,NU
LOOP1   BEQ.S      100          ;
102     MOVE.W     N2,D1        ;DO  101 I=1,N2
LOOP2   BEQ.S      101          ;
        MOVE.W    NU1,D2        ;P=IBITR(K/2**NU1,NU)
        MOVE.W    K,D3          ;
LOOP3   BEQ.S      200          ;
        ASR.W     #1,D3         ;
        SUBI.W    #1,D2         ;
        BRA       LOOP3         ;
200     MOVE.W     D3,J         ;
        JSR       IBITR         ;
        MOVE.L    RX,P          ;
        MOVE.W    N,D3          ;ARG = 6.283185*P/FLOAT(N)
                                ;convert N to float. point
        MOVEQ.L   #159,D4       ;
300     ASL        #1,D3        ;
        SUBI.L    #1,D4         ;
        BCC       300           ;
        MOVE.B    #9,D5         ;
        LSR.L     D5,D3         ;
        ROR.L     D5,D4         ;
        ANDI.L    mask,D4       ;clear D4 except exponent
        OR.L      D4,D3         ;D3 <-- FLOAT(N)
        MOVE.L    D3,FPN        ;store FPN
```

```
        MOVE.L    P,D3          ;convert P to float. point
        MOVEQ.L   #159,D4       ;
400     ASL        #1,D3;
        SUBI.L    #1,D4         ;
        BCC       400           ;
        MOVE.B    #9,D5         ;
        LSR.L     D5,D3         ;
        ROR.L     D5,D4         ;
        ANDI.L    mask,D4       ;clear D4 except exponent
        OR.L      D4,D3         ;D4 <-- FLOAT(P)
        MOVE.L    D3,FPP        ;store FPP
        LEA       FPWR,A2       ;A2 points to Floating Point
                                ;Working Register
        LEA       FPACC,A1      ;A1 points to Floating Point
                                ;Accumulator
        LEA       FPP,A0        ;FPWR <-- FPP
        JSR       GETFP         ;
        MOVE.L    #2PI,(A1)     ;FPACC <-- 2PI
        MOVE.B    #2PI,2(A1)    ;
        JSR       MULTFP        ;FPACC <-- 2PI
        LEA       FPN,A0        ;FPWR <-- FPN
        JSR       GETFP         ;
        JSR       DIVFP         ;FPACC <-- 2PI/FPN
        LEA       ARG,A0        ;store ARG
        JSR       STFP          ;
        MOVE.L    ARG,X         ;C=COS(ARG)
        JSR       COSINE        ;
        MOVE.L    RESULT,C      ;store C
        JSR       SINE          ;S=SIN(ARG)
        MOVE.L    RESULT,S      ;store S
        MOVE.W    K,K1          ;K1=K+1
        ADDI.W    #1.K1         ;
        MOVE.W    K1,D3         ;K1N2=K1+N2
        ADD.W     N2,D3         ;
        MOVE.W    D3,K1N2       ;
```

```
LEA       XREAL,A3      ;TREAL=XREAL(K1N2)*C+
                        ;          +XIMAG(K1N2)*S
LEA       XIMAG,A4      ;
ASL.W     #1,D3         ;D3 <-- 2*K1N2
SUBI.W    #2,D3         ;D3 <-- 2*K1N2-2
ADDA.W    D3,A3         ;
ADDA.W    D3,A4         ;
MOVEA.L   A3,A0         ;FPWR <-- XREAL(K1N2)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
LEA       C,A0          ;FPWR <-- c
JSR       GETFP         ;
JSR       MULTFP        ;FPACC <-- XREAL(K1N2)*C
LEA       TREAL,A0      ;store partial result
JSR       STFP          ;
MOVEA.L   A4,A0         ;FPWR <-- XIMAG(K1N2)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
LEA       S,A0          ;FPWR <-- S
JSR       GETFP         ;
JSR       MULTFP        ;FPACC <-- XIMAG(K1N2)*S
LEA       TREAL,A0      ;FPWR <-- partial TREAL
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- TREAL
JSR       STFP          ;store TREAL
                        ;TIMAG=XIMAG(K1N2)*C-
                        ;          -XREAL(K1N2)*S
MOVEA.L   A3,A0         ;FPWR <-- XREAL(K1N2)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
LEA       S,A0          ;FPWR <-- S
JSR       GETFP         ;
```

```
JSR       MULTFP        ;FPACC <-- XREAL(K1N2)*S
LEA       TIMAG,A0      ;store partial result
JSR       STFP          ;
EORI.L    mask,(A0)     ;change sign of TIMAG
MOVEA.L   A4,A0         ;FPWR <-- XIMAG(K1N2)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
LEA       C,A0          ;FPWR <-- C
JSR       GETFP         ;
JSR       MULTFP        ;FPACC <-- XIMAG(K1N2)*C
LEA       TIMAG,A0      ;FPWR <-- partial TIMAG
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- TIMAG
JSR       STFP          ;store TIMAG
                        ;XREAL(K1N2)=XREAL(K1)-TREAL
EORI      mask,TREAL    ;change sign of TREAL
MOVE.L    TREAL,(A3)    ;XREAL(K1N2) <-- TREAL
LEA       XREAL,A5      ;
MOVE.L    K1,D3         ;
ASL       #1,D3         ;
SUBI.L    #2,D3         ;
ADDA      D3,A5         ;
MOVEA.L   A5,A0         ;FPWR <-- XREAL(K1)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
MOVEA.L   A3,A0         ;FPWR <-- XREAL(K1N2)
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- XREAL(K1)-TREAL
JSR       STFP          ;store
                        ;XIMAG(K1N2)=XIMAG(K1)-
                        ;            -TIMAG
EORI      mask,TIMAG    ;change sign of TIMAG
MOVE.L    TIMAG,(A4)    ;XIMAG(K1N2) <-- -TIMAG
```

```
LEA       XIMAG,A6      ;
ADDA.L    D3,A6         ;A6 --> XIMAG(K1)
MOVEA.L   A6,A0         ;FPWR <-- XIMAG(K1)
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
MOVEA.L   A4,A0         ;FPWR <-- XIMAG(K1N2)
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- XIMAG(K1N2)
JSR       STFP          ;store
                        ;XREAL(K1)=XREAL(K1)+
                        ;              +TREAL
EORI      mask,TREAL    ;change sign of -TREAL
LEA       TREAL,A0      ;FPWR <-- TREAL
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
MOVEA.L   A5,A0         ;FPWR <-- XREAL(K1)
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- final XREAL(K1)
JSR       STFP          ;store
                        ;XIMAG(K1)=XIMAG(K1)+
                        ;                +TIMAG
EORI      mask,TIMAG    ;change sign of -TIMAG
LEA       TIMAG,A0      ;FPWR <-- TIMAG
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- FPWR
MOVE.B    2(A2),2(A1)   ;
MOVEA.L   A6,A0         ;FPWR <-- partial XIMAG(K1)
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- final XIMAG(K1)
JSR       STFP          ;store
ADDI.W    #1,K          ;K=K+1
SUBQ.W    #1,D1         ;
BRA       LOOP2         ;
```

```
101     MOVE.W    N2,D1          ;K=K+N2
        ADD.W     K,D1           ;
        MOVE.W    D1,K           ;
        CMP.W     N,D1           ;IF (K.LT.N) GO TO 102
        BMI       102            ;
        CLR.W     K              ;K=0
        SUBI.W    #1,NU1         ;NU1=NU1-1
        ASR.W     N2             ;N2=N2/2
        SUBQ.W    #1,D0          ;
        BRA       LOOP1          ;
100     MOVE.W    N,D0           ;
        MOVE.W    #1,D1          ;DO  103 K=1,N
LOOP4   BEQ.S     103            ;
        MOVE.W    D1,J           ;I=IBITR(K-1,NU)+1
        SUBI.W    #1,J           ;
        JSR       IBITR          ;
        MOVE.W    RX,I           ;
        ADDI.W    #1,I           ;
        CMP.W     I,D1           ;IF (I.LE.K) GO TO 103
        BPL       1003           ;
        LEA       XREAL,A3       ;TREAL=XREAL(K)
        LEA       XIMAG,A4       ;
        MOVE.W    D1,D2          ;
        ASR       #1,D2          ;
        SUBI.W    #2,D2          ;
        MOVEA.L   A3,A5          ;
        MOVEA.L   A4,A6          ;
        MOVE.W    I,D3           ;
        ASR       #1,D3          ;
        SUBI      #2,D3          ;
        ADDA.L    D1,A3          ;A3 --> XREAL(K)
        ADDA.L    D1,A5          ;A5 --> XIMAG(K)
        ADDA.L    D2,A4          ;A4 --> XREAL(I)
        ADDA.L    D2,A6          ;A6 --> XIMAG(I)
        MOVE.L    (A3),TREAL     ;
```

```
        MOVE.L    (A5),TIMAG    ;TIMAG=XIMAG(K)
        MOVE.L    (A4),(A3)     ;XREAL(K)=XREAL(I)
        MOVE.L    (A6),(A5)     ;XIMAG(K)=XIMAG(I)
        MOVE.L    TREAL,(A4)    ;XREAL(I)=TREAL
        MOVE.L    TIMAG,(A6)    ;XIMAG(I)=TIMAG
1003    ADDQ.W    #1,D1         ;
        SUBQ.W    #1,D0         ;
        BRA       LOOP4         ;
103     RTS                     ;RETURN
```

APPENDIX B

IBITR FUNCTION

```
IBITR   MOVEM.L   D0-D3,-(A7)   ;save registers
        MOVE.W    J,J1          ;J1=J
        CLR.W     IBIT          ;IBITR=0
        MOVE.W    NU,D0         ;DO  200 I=1,NU
LOOP    BEQ.S     2000          ;
        MOVE.W    J1,D1         ;J2=J1/2
        ASR.W     #1,D1         ;
        MOVE.W    D1,D2         ;D2 <-- J2
                                ;IBITR=IBITR*2+(J1-2*J2)
        ASL.W     #1,D2         ;
        MOVE.W    J1,D3         ;
        SUB.W     D2,D3         ;D2 <-- (J1-2*J2)
        ASL       IBIT          ;
        ADD.W     D3,IBIT       ;
        MOVE.W    D1,J1         ;J1=J2
        SUBI      #1,D0         ;
        BRA       LOOP          ;
2000    MOVEM.L   (A7)+,D0-D3   ;restore registers
        RTS                     ;RETURN
```

APPENDIX C

SINE FUNCTION

```
SINE    MOVEM.L   D0-D4,-(A7)   ;save registers
        MOVE.L    X,D0          ;
        BTST.L    #bit,X        ;test sign of X
        BNE       100           ;
        MOVE.B    #-1,SGN       ;SGN <-- -1
        BCHG      #bit,D0       ;D0 <-- -D0
        BRA       200           ;
100     MOVE.B    #1,SGN        ;SGN <-- 1
        MOVE.L    D0,Y          ;Y <-- D0
200     CMP.L     YMAX,D0       ;YMAX - D0
        BPL       300           ;
        error message
300     MOVEA.L   Y,A0          ;A0 --> Y
        JSR       GETFP         ;FPWR <-- Y
        MOVE.L    1/PI,(A1)     ;FPACC <-- inverse of pi
        MOVE.B    1/PI,2(A1)    ;
        JSR       MULTFP        ;FPACC <-- Y/PI
        MOVEA.L   Y/PI,A0       ;A0 --> Y/PI
        JSR       STFP          ;store Y/PI
        MOVE.L    Y/PI,D1       ;D1 <-- Y/PI
        MOVE.L    D1,D2         ;
        ANDI.L    mask,D1       ;D1 <-- mantissa
        BSET      #bit,D1       ;insert hidden bit
        LSR       #7,D2         ;hi D2 has exponent
        SWAP      D2            ;lo D2 has exponent
        SUBI.B    #127,D2       ;extract bias
        BPL       400           ;if positive go to 400
        MOVE.W    #0,N          ;clear N
        BRA       500           ;
400     BNE       600           ;if zero go to 600
```

```
        MOVE.W    #1,N          ;N <-- 1
        BRA       500           ;
600     ASL.L     D2,D1         ;shift left mantissa by
                                ;exponent value, max = 8
        ANDI      mask,D1       ;leave only integer part
        ASR.L     #7,D1         ;
        SWAP      D1            ;mantissa in lo D1
        MOVE.W    D1,N          ;N <-- integer of mantissa
500     MOVE.L    Y/PI,XN       ;XN <-- FLOAT(N)
        BTST.B    #0,N          ;N even ?
        BEQ       700           ;if even do nothing
                                ;otherwise
        BCHG      #7,SGN        ;change sign of SGN
700     MOVE.L    X,|X|         ;determine F
        ANDI      mask,|X|      ;clear sign bit
        MOVEA.L   XN,A0         ;FPWR <-- XN
        JSR       GETFP         ;
        MOVE.L    -C1,(A1)      ;FPACC <-- C1
        MOVE.B    -C1,2(A1)     ;
        JSR       MULTFP        ;FPACC <-- -(XN*C1)
        MOVEA.L   |X|,A0        ;FPWR <-- |X|
        JSR       GETFP         ;
        JSR       ADDFP         ;FPACC <-- |X|-(XN*C1)
        MOVEA.L   TEMP,A0       ;store FPACC
        JSR       STFP          ;
        MOVEA.L   XN,A0         ;FPWR <-- XN
        JSR       GETFP         ;
        MOVE.L    -C2,(A1)      ;FPACC <-- -C2
        MOVE.B    -C2,2(A1)     ;
        JSR       MULTFP        ;FPACC <-- -(XN*C2)
        MOVEA.L   TEMP,A0       ;FPWR <-- |X|-(XN*C1)
        JSR       GETFP         ;
        JSR       ADDFP         ;FPACC <-- F
        MOVEA.L   F,A0          ;store F
        JSR       STFP          ;
```

```
MOVE.L    F,|F|          ;|F| <-- F
ANDI.L    mask,|F|       ;clear sign bit
CMPI.L    |F|,#eps       ;|F| - eps
BMI       800            ;branch if |f| < eps
                         ;otherwise
                         ;determine R(g)
MOVEA.L   F,A0           ;FPWR <-- F
JSR       GETFP          ;
MOVE.L    (A2),(A1)      ;FPACC <-- F
MOVE.B    2(A2),2(A1)    ;
JSR       MULTFP         ;FPACC <-- F*F
                         ;G = F*F
MOVE.L    (A1),(A2)      ;FPWR <-- G
MOVE.B    2(A1),2(A2)    ;
MOVE.L    R4,(A1)        ;FPACC <-- r4
MOVE.B    R4,2(A1)       ;
JSR       MULTFP         ;FPACC <-- r4*G
MOVEA.L   G,A0           ;store G
JSR       STFP           ;
MOVE.L    R3,(A2)        ;FPWR <-- r3
MOVE.B    R3,2(A2)       ;
JSR       ADDFP          ;FPACC <-- r4*G+r3
MOVEA.L   G,A0           ;FPWR <-- G
JSR       GETFP          ;
JSR       MULTFP         ;FPACC <-- (r4*G+r3)*G
MOVE.L    R2,(A2)        ;FPWR <-- r2
MOVE.B    R2,2(A2)       ;
JSR       ADDFP          ;FPACC <-- (r4*G+r3)*G+r2
MOVEA.L   G,A0           ;FPWR <-- G
JSR       GETFP          ;
JSR       MULTFP         ;FPACC <-- ((   )*G+r2)*G
MOVE.L    R1,(A2)        ;FPWR <-- r1
MOVE.B    R1,2(A2)       ;
JSR       ADDFP          ;FPACC <-- (   )*G+r1
MOVEA.L   G,A0           ;FPWR <-- G
```

```
        JSR       GETFP          ;
        JSR       MULTFP         ;FPACC <-- R(g)
        MOVEA.L   F,A0           ;FPWR <-- F
        JSR       GETFP          ;
        JSR       MULTFP         ;FPACC <-- F*R(g)
        JSR       ADDFP          ;FPACC <-- F*R(g)+F
        MOVEA.L   RESULT,A0      ;store result
        JSR       STFP           ;
        BRA       900            ;
800     MOVE.L    F,RESULT       ;result <-- F
900     MOVE.B    SGN,D3         ;test value of SGN
        BPL       DONE           ;if positive do nothing
                                 ;otherwise
                                 ;change sign of result
        MOVE.L    RESULT,D4      ;
        BCHG      #31,D4         ;
        MOVE.L    D4,RESULT      ;
DONE    MOVEM.L   (A7)+,D0-D4    ;restore registers
        RTS                      ;return to main program
```

APPENDIX D

COSINE FUNCTION

```
COSINE  MOVEM.L   D0-D4,-(A7)  ;save registers
        MOVE.B    #1,SGN       ;SGN <-- 1
        MOVE.L    X,|X|        ;|X| <-- x
        ANDI      mask,|X|     ;clear sign bit
        MOVEA.L   |X|,A0       ;FPWR <-- |X|
        JSR       GETFP        ;
        MOVE.L    PI/2,(A1)    ;FPACC <-- PI/2
        MOVE.B    PI/2,2(A1)   ;
        JSR       ADDFP        ;FPACC <-- |X|+PI/2
        MOVEA.L   Y,A0         ;store Y
        JSR       STFP         ;
        MOVE.L    Y,D0         ;D0 <-- Y
        CMP.L     YMAX,D0      ;YMAX - D0
        BPL       100          ;
        error message
100     MOVEA.L   Y,A0         ;A0 --> Y
        JSR       GETFP        ;FPWR <-- Y
        MOVE.L    1/PI,(A1)    ;FPACC <-- inverse of pi
        MOVE.B    1/PI,2(A1)   ;
        JSR       MULTFP       ;FPACC <-- Y/PI
        MOVEA.L   Y/PI,A0      ;A0 --> Y/PI
        JSR       STFP         ;store Y/PI
        MOVE.L    Y/PI,D1      ;D1 <-- Y/PI
        MOVE.L    D1,D2        ;
        ANDI.L    mask,D1      ;D1 <-- mantissa
        BSET      #bit,D1      ;insert hidden bit
        LSR       #7,D2        ;hi D2 has exponent
        SWAP      D2           ;lo D2 has exponent
        SUBI.B    #127,D2      ;extract bias
        BPL       200          ;if positive go to 200
```

```
        MOVE.W    #0,N          ;clear N
        BRA       300           ;
200     BNE       400           ;if zero go to 400
        MOVE.W    #1,N          ;N <-- 1
        BRA       300           ;
400     ASL.L     D2,D1         ;shift left mantissa by
                                ;exponent value, max = 8
        ANDI      mask,D1       ;leave only integer part
        ASR.L     #7,D1         ;
        SWAP      D1            ;mantissa in lo D1
        MOVE.W    D1,N          ;N <-- integer of mantissa
300     MOVE.L    Y/PI,XN       ;XN <-- FLOAT(N)
        BTST.B    #0,N          ;N even ?
        BEQ       500           ;if even do nothing
                                ;otherwise
        BCHG      #7,SGN        ;change sign of SGN
500     MOVEA.L    XN,A0        ;FPWR <-- XN
        JSR       GETFP         ;
        MOVE.L    #-.5,(A1)     ;FPACC <-- .5
        MOVE.B    #-.5,2(A1)    ;
        JSR       ADDFP         ;FPACC <--XN-.5
        JSR       STFP          ;store XN
                                ;determine F
        MOVEA.L   XN,A0         ;FPWR <-- XN
        JSR       GETFP         ;
        MOVE.L    -C1,(A1)      ;FPACC <-- C1
        MOVE.B    -C1,2(A1)     ;
        JSR       MULTFP        ;FPACC <-- -(XN*C1)
        MOVEA.L   |X|,A0        ;FPWR <-- |X|
        JSR       GETFP         ;
        JSR       ADDFP         ;FPACC <-- |X|-(XN*C1)
        MOVEA.L   TEMP,A0       ;store FPACC
        JSR       STFP          ;
        MOVEA.L   XN,A0         ;FPWR <-- XN
        JSR       GETFP         ;
```

```
MOVE.L    -C2,(A1)      ;FPACC <-- -C2
MOVE.B    -C2,2(A1)     ;
JSR       MULTFP        ;FPACC <-- -(XN*C2)
MOVEA.L   TEMP,A0       ;FPWR <-- |X|-(XN*C1)
JSR       GETFP         ;
JSR       ADDFP         ;FPACC <-- F
MOVEA.L   F,A0          ;store F
JSR       STFP          ;
MOVE.L    F,|F|         ;|F| <-- F
ANDI.L    mask,|F|      ;clear sign bit
CMPI.L    |F|,#eps      ;|F| - eps
BMI       600           ;branch if |f| < eps
                        ;otherwise
                        ;determine R(g)
MOVEA.L   F,A0          ;FPWR <-- F
JSR       GETFP         ;
MOVE.L    (A2),(A1)     ;FPACC <-- F
MOVE.B    2(A2),2(A1)   ;
JSR       MULTFP        ;FPACC <-- F*F
                        ;G = F*F
MOVE.L    (A1),(A2)     ;FPWR <-- G
MOVE.B    2(A1),2(A2)   ;
MOVE.L    R4,(A1)       ;FPACC <-- r4
MOVE.B    R4,2(A1)      ;
JSR       MULTFP        ;FPACC <-- r4*G
MOVEA.L   G,A0          ;store G
JSR       STFP          ;
MOVE.L    R3,(A2)       ;FPWR <-- r3
MOVE.B    R3,2(A2)      ;
JSR       ADDFP         ;FPACC <-- r4*G+r3
MOVEA.L   G,A0          ;FPWR <-- G
JSR       GETFP         ;
JSR       MULTFP        ;FPACC <-- (r4*G+r3)*G
MOVE.L    R2,(A2)       ;FPWR <-- r2
MOVE.B    R2,2(A2)      ;
```

```
        JSR       ADDFP          ;FPACC <-- (r4*G+r3)*G+r2
        MOVEA.L   G,A0           ;FPWR <-- G
        JSR       GETFP          ;
        JSR       MULTFP         ;FPACC <-- ((   )*G+r2)*G
        MOVE.L    R1,(A2)        ;FPWR <-- r1
        MOVE.B    R1,2(A2)       ;
        JSR       ADDFP          ;FPACC <-- (   )*G+r1
        MOVEA.L   G,A0           ;FPWR <-- G
        JSR       GETFP          ;
        JSR       MULTFP         ;FPACC <-- R(g)
        MOVEA.L   F,A0           ;FPWR <-- F
        JSR       GETFP          ;
        JSR       MULTFP         ;FPACC <-- F*R(g)
        JSR       ADDFP          ;FPACC <-- F*R(g)+F
        MOVEA.L   RESULT,A0      ;store result
        JSR       STFP           ;
        BRA       700            ;
600     MOVE.L    F,RESULT       ;result <-- F
700     MOVE.B    SGN,D3         ;test value of SGN
        BPL       DONE           ;if positive do nothing
                                 ;otherwise
                                 ;change sign of result
        MOVE.L    RESULT,D4      ;
        BCHG      #31,D4         ;
        MOVE.L    D4,RESULT      ;
DONE    MOVEM.L   (A7)+,D0-D4    ;restore registers
        RTS                      ;return to main program
```

LIST OF REFERENCES

1. Katevenis, Manolis H., Reduced Instruction Set Computer Architectures for VLSI, Ph.D. Thesis, University of California at Berkeley, 1983.

2. Radin, George, "The 801 Minicomputer", IBM Journal of Research and Development, Volume 27 Number 3, May 1983.

3. Stanford University Computer Systems Laboratory, Technical Report 223, MIPS: A VLSI Processor Architecture, by Hennessy, and others, November, 1981.

4. Brigham, E. Oran, The Fast Fourier Transform, Prentice-Hall, 1974.

5. Eccles, William J., Microprocessor Systems, a 16-bit Approach, Addison-Wesley, 1985.

6. Cody Jr ,William J. and Waite, William, Software Manual for the Elementary Functions, Prentice-Hall, 1980.

7. Stone, H., and others, Introduction to Computer Architecture, Science Research Associates, 1980.

8. Hill, Frederick J. and Peterson, Gerald R., Digital Systems: Hardware Organization and Design, Wiley, 1978.

AD-A167 873 THE RISC (REDUCED INSTRUCTION SET COMPUTER) ARCHITECTURE AND COMPUTER PERFORMANCE EVALUATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M F BARROS 2/2

UNCLASSIFIED MAR 86 F/G 9/2 NL

MICROCOPY CHART

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Dr. Harriett B. Rigas Code 62Rr Naval Postgraduate School Monterey, California 93943	2
4.	Dr. Larry Abbott Code 62Ab Naval Postgraduate School Monterey, California 93943	1
5.	Dir. Serv. Instrucao e Treino Edificio do Ministerio da Marinha Rua do Arsenal 1000 Lisboa Portugal	1
6.	Manuel Pedrosa de Barros Celula 5 Bloco 5 Lote D, 3 Direito 2795 Linda-a-Velha Portugal	4

END

DTIC

6-86