PERFORMANCE EVALUATION OF A RIDGE 32 COMPUTER SYSTEM

SEOK TAE YOON

PERFORMANCE EVALUATION OF A RIDGE 32 COMPUTER SYSTEM

by

Seok Tae Yoon

A Thesis Submitted to the Faculty of the

College of Engineering

in Partial Fulfilment of the Requirements for the Degree of

Master of Science in Computer Engineering

Florida Atlantic University

Boca Raton, Florida

December 1986

PERFORMANCE EVALUATION OF A

RIDGE 32 COMPUTER SYSTEM

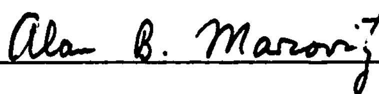
BY

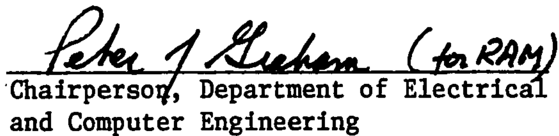
Seok Tae Yoon

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Eduardo B. Fernandez, Department of Electrical Engineering and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

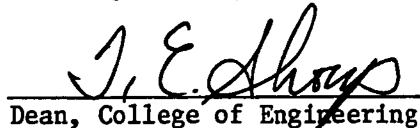
SUPERVISORY COMMITTEE

Thesis Advisor





Chairperson, Department of Electrical and Computer Engineering



Dean, College of Engineering

Dean for Advanced Studies

Date

ACKNOWLEDGEMENTS

I am particularly grateful to my thesis advisor, Professor Eduardo B. Fernandez for his guidance and encouragement throughout all phases of this work.

I want to thank my parents for their endless support.

Lastly, I would like to thank my wife, Hyun Gyung, for her continued support and sacrifice.

ABSTRACT

Author: Seok Tae Yoon

Title: Performance Evaluation of a Ridge 32 Computer System

Institution: Florida Atlantic University

Degree: Master of Science in Computer Engineering

Year: 1986

As a new trend in designing a computer architecture, Reduced Instruction Set Computers(RISC) have been proposed recently. This thesis reviews the new design approach behind the RISC and discuss the controversy between the proponents of the RISC approach and those of the traditional Complex Instruction Set Computer(CISC) approach. Ridge 32 is selected as a case study of the RISCs. Architectural parameters to evaluate the computer performance are considered to analyze the performance of the Ridge 32. A simulator for the Ridge 32 was implemented in PASCAL as a way of measuring those parameters. Measurement results on the several selected benchmark programs are given and analyzed to evaluate the characteristics of the Ridge 32.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF ILLUSTRATIONS	viii
I. INTRODUCTION	1
1.1. BACKGROUND	1
1.2. OBJECTIVES OF THE THESIS	3
1.3. OUTLINE OF THE THESIS	3
II. REDUCED INSTRUCTION SET COMPUTERS (RISC)	5
2.1. EVOLUTION OF THE RISC CONCEPT	5
2.2. RISC VERSUS COMPLEX INSTRUCTION SET COMPUTERS (CISC)	10
2.3. THE RIDGE 32	13
III. PERFORMANCE EVALUATION	19
3.1. PREVIOUS PERFORMANCE EVALUATIONS OF RISC SYSTEMS	19
3.2. PARAMETER SELECTION	24
3.3. MEASUREMENT METHODS	28
3.4. RIDGE 32 SIMULATOR	29
3.5. MEASUREMENT RESULTS	43
3.6. INTERPRETATION OF MEASUREMENTS	70

Page

IV. CONCLUSION 80

REFERENCES 83

APPENDIXES 86

A. BENCHMARK PROGRAM LISTINGS 86

B. EXAMPLE OF USE OF THE RIDGE 32 SIMULATOR 107

C. RIDGE 32 SIMULATOR LISTING 114

D. RIDGE 32 OPCODE MAP 185

LIST OF TABLES

Page

Table 3.5.1. Benchmark Program Execution Time on Ridge 32 47

Table 3.5.2. Program Size of Pascal Benchmarks on Ridge 32 49

Table 3.5.3. Simulation Result of Static Instruction Usage 51

Table 3.5.4. Simulation Result of Dynamic Instruction Usage 57

Table 3.5.5. Statistics of Register-to-Register Formatted Instructions versus Memory Reference Instructions 64

Table 3.5.6. Statistics of Memory Access Instructions 65

Table 3.5.7. Statistics of Branch Instructions 66

Table 3.5.8. Statistics of Code Memory Access versus Data Memory Access 67

Table 3.5.9. Simulation Result of Average Lengh of Loop 68

Table 3.5.10 Simulation Result of Average Lengh of Branch Offset 69

Table 3.6.1. Benchmark Execution Time and MIPS Results 71

Table 3.6.2. Execution Times of Sieve, Puzzle and Ackerman Benchmarks on Several Computers 73

LIST OF ILLUSTRATIONS

		Page
Figure 2.3.1.	Ridge 32 CPU Internal Structure	15
Figure 2.3.2.	Ridge 32 Instruction Formats	16
Figure 3.4.1.	Flowchart Diagram of Ridge 32 Simulator .	33
Figure 3.5.1.	Relationships between System Command and Files	45

CHAPTER I

INTRODUCTION

1.1. BACKGROUND

One of the recent important advances in computer architecture is the Reduced Instruction Set Computer (RISC), proposed in contrast to the traditional Complex Instruction Set Computer (CISC). The computer architects of traditional CISC systems have been trying to pack more and more functions into computer hardware to handle more complex problems and to reduce the semantic gap between the architecture and the high level language. This has resulted in a large number of instructions, complex addressing modes, intensive use of microprogramming, and a high average of memory accesses per instruction. The VAX/11, IBM370, Intel432,286, and 386, Motorola 68010 and 68020, and the National semiconductor 32000, are typical processors which have a CISC architecture.

There have been several recent studies, trying to find out the most frequently used instructions and to optimize

data paths and timing for these instructions. This is coupled with the use of simple addressing modes and simple control logic which permits the use of fast hardwired controllers instead of slower microprogramming. The resulting machine has a very small set of very fast instructions and relies on the compiler to produce optimized instruction sequences. These ideas lead to the RISC concept and to the RISC I and II from the University of California[Patt82b,Patt85b], the 801 at IBM[Radi82], MIPS at Stanford University[Henn82,Henn84], the Ridge 32 of Ridge Corp.[Basa82,Basa83,Basa85], the Pyramid 90x of Pyramid Corp.[Raga83], the Transputer of Inmos Corp.[May84], and some others.

While the interest about RISCs is growing, there is controversy on several points between the proponents of the RISC approach and the proponents of the CISC approach [Colw85, Patt85a, Wall85]. Several studies [Basa82, Basa85, Laru82, Patt82, Rals85] about performance evaluation of these RISCs reveal superior performance compared to those CISC on several well known benchmark programs with the additional benefit of simpler hardware, short development time, and simple design verification. These facts mean faster computers at lower cost. The Department of Electrical & Computer Engineering at F.A.U. has a Ridge 32 computer which is one of the first RISCs in commercial production as an engineering workstation.

1.2. OBJECTIVES OF THE THESIS

The purpose of this thesis is to provide more data to help clarify the RISC/CISC controversy. Nobody has performed a systematic performance evaluation of the Ridge 32 and this evaluation should be useful in pointing out good and bad aspects of this particular RISC implementation and of RISC systems in general. The thesis also surveys the previous measurement work and evaluates the methods used in those studies. The survey of architectural measures is important in its own right, since different parameters have been considered in different studies, measured in different ways, and there is a need to evaluate the methodology itself.

1.3. OUTLINE OF THE THESIS

The remainder of this thesis starts with a review of previous evaluations of RISC systems. Discussion of performance measurement methods is given and performance measurement work on RISCs is reviewed. Actual measurement was done on the Ridge 32 and measurement results are discussed.

In chapter II, theoretical background behind the RISC is discussed and the controversy between the supporters of RISCs and

those of the CISCs is reviewed. Architectural aspects of Ridge 32 are discussed to help the understanding of this particular implementation of the RISC concept.

Chapter III begins with a survey of previous performance evaluations of RISCs. Several architectural parameters of interest are selected and these parameters are discussed. Since a simulator for a certain target computer gives us the most flexible way to measure the several parameters, a simulator for the Ridge 32 was built in this study. The Ridge 32 simulator is described in this chapter and measurement results are presented.

A discussion of the results of this experiment is given in chapter IV.

CHAPTER II

REDUCED INSTRUCTION SET COMPUTERS (RISC)

2.1. EVOLUTION OF THE RISC CONCEPT

Since the introduction of the electromechanical computer Mark I, with only seven instructions in 1944, computer architecture has become more and more complicated with large and complex instruction set computers appearing during the last several decades. This progression from small and simple to large and complex instruction sets has been mirrored in the microprocessor industry. This general trend toward CISCs can be explained by several reasons :

* New models are often required to be upward-compatible with existing models in the same computer family.
* Many computer designers tried to reduce the "semantic gap" between programs and computer instruction sets by adding instructions semantically closer to those used by programmers and such instructions tend to be more complex because of their higher semantic level.

* In striving to develop faster machines, designers constantly moved functions from software to microcode and from microcode to hardware.
* New tools and methodologies aid designers in handling the inherent complexity of large architectures.
* The low cost of the technology has made possible the use of hardware for functions which were normally performed in software.
* Packing more functions in hardware seems to help programmers develop high-level-language programs that are shorter, more efficient, and easier to write, compile, and debug.
* When one vendor introduces a type of instructions, e.g. bit manipulation instructions, the other vendors are obliged to introduce a similar feature to keep up.

In reaction to the traditional trend in computer design, the "801" project[Radi82] at IBM introduced a new wave in computer design. The design of the 801 was based on the analysis of trace tapes which showed the patterns of instructions that are actually executed by a given machine in daily use. They showed that relatively simple instructions such as load, store, and branch are used far more than the other more complex instructions that CISCs may have in their instruction sets. The designers of the 801 therefore used hardwired logic to execute all such

primitive instructions in one instruction cycle. Higher-level instructions were then implemented by software subroutines that use the primitives. The designers also optimized the compiler to speed up program execution. This study was followed by the RISC project at the University of California at Berkeley. These studies raised the question of whether the extra hardware needed to make a more powerful processor would lead to increased design time, increased design errors, and inconsistent implementations. They also observed that several points can limit the microprocessor design under the CISC approach. Those observations lead to the RISC I and II which were designed with the following objectives :

* RISC I instructions should be about as fast and no more complicated than microinstructions in current CISCs.
* All instructions should have the same size to simplify implementation.
* Only load and store instructions can access memory. The rest operate between registers.
* Support high-level languages.

Soon after the RISC I project, The Microprocessor without Interlocked Pipeline Stages (MIPS) project at Stanford University took shape at Stanford University, which lead to a pipelined single-chip MIPS. It used a special systems software to

rearrange programs and schedule instructions so that its pipeline resources are properly managed. Conventional design solves these pipeline problems with special hardware that prefetches instructions at the destination of a branch or with interlocks that prevents instructions from accessing valid information or from trying to use the same resource.

Those RISCs share a set of common features :

* Single-cycle operation for most instructions.
* Operations are register oriented, with only LOAD and STORE accessing memory (also known as load/store architecture).
* A few simple fixed instruction formats.
* More compile time effort to get optimized code.
* Relatively few instructions and addressing modes.

Other features which are typically found in RISCs are :

* Reusability of data in registers, i.e. operations must not destroy the contents of the operand registers.
* No microprogramming. Hardwired control units can increase execution speed.

* No condition code. This improves pipeline optimization [Przy84].

Both the IBM and Stanford machines gained significant benefits from compiler technology to optimize the use of registers and of the pipeline. The Berkeley machine used a large register set to reduce the frequency of register saving and restoring during procedure calls, which significantly improve the performance in a high-level-language environment.

These new approaches in designing computer architecture stimulated the commercial industry and leading computer companies such as IBM, HARRIS Corp., and Hewlett-Packard have already anounced new products based on these approaches.

2.2. RISC VERSUS COMPLEX INSTRUCTION SET COMPUTERS (CISC)

There is a controversy between the proponents of the RISC approach and the proponents of the CISC approach, which is not likely to be settled down in the near future.

The supporters of RISC machines consider the following points are disadvantages of CISCs.

* Complex machine instructions may not match high-level language instructions exactly, in which case it may be awkward to compile a given HLL.
* Rich instruction sets present a complex choice to a compiler, who may not be able of finding the correct specialized instruction to carry out a particular high level function, i.e. same HLL instruction could be executed in many different ways, each with its own advantages and tradeoffs. The inverse is true for RISCs, where there is usually only one way to implement a given high-level construct.
* Instruction sets designed with specialized instructions for several high level languages will carry excess baggage when executing a particular language.
* Since complex machine instructions often have intricate execution sequences and side effects, programs using them can be difficult to optimize.

* Large instruction sets require complex and potentially time consuming hardware steps to decode and execute them.

On the other hand, the supporters of CISC argue that RISCs present the following problems :

* RISC instruction sets have been pared down to the point where certain operations that might take only a few instructions on a conventional computer require complex subroutines. This results in larger programs than those for CISCs and this fact degrades performance since more instructions from slower main memory are needed. Larger programs require more program loading time and are more likely to cause page faults in virtual memory systems which results in reduced performance and fewer cache hits.
* RISC's serialize instruction execution at a (semantically) very low level, which prevents them from exploiting the many opportunities for parallel execution that are inherent in semantically rich architectures.
* RISCs forfeit processor enforced type checking, error detection and monitoring at execution time, thus reducing overall system reliability.
* Overlapped register set RISCs such as RISC I and II suffer dynamic performance losses when executing programs where the

register stack overflows. This will slow down the execution speed of multiple tasks on those machines.

* Performance gains from overlapped register sets in some RISC computers have nothing to do with reduced instruction sets and could be easily implemented in CISCs.
* Although RISC computers have simpler hardware, their compilers must be proportionately more complex to wring performance out of a simple CPU. If the compiler does not perform well, then code size can balloon and execution speed can drop.
* RISC principles make no use of the increasing density available with advances in Integrated Circuit technology. RISC arguments have their place at a certain point of technology but they are now overtaken by the increasing density of semiconductor circuit.
* A rating method for computers used by the U.S. Department of Defense in its research for a new standard computer architecture put RISCs very low on scale, whereas the VAX was rated near top[Colw85].

In response to the above criticisms, the RISC designers have the following points.

* Compilers are now simpler, since there will usually be only one way to perform a particular function.
* Even though RISCs seem to generate larger programs, the RISC machines tend to execute about the same number of instructions as their CISC counterpart for a given problem due to the amount of optimization that compilers can perform on RISC programs.
* It is relatively easy for optimizing programs to combine operations and make software faster, since their instructions break operations down to the simplest possible level.
* It is better to put complexity in the software since bugs in the software are easier to fix.

2.3. THE RIDGE32

Even though some critics question whether a computer like the Ridge 32 should qualify as a RISC, Ridge claims to have created the one of the first commercial RISCs which is designed under the idea of building a high-performance, moderately priced engineering and scientific workstation.

Ridge 32 is 32-bit minicomputer with three instruction formats, which are register-to-register(16 bit long), short

displacement memory address (32 bit long) and long displacement memory address (48 bit long) format. There are 16 32-bit general purpose registers which can be used as data, indexing and addressing. Figure 2.3.1. shows the internal CPU structure of the Ridge 32 computer.

All instructions have an 8 bit op code followed by two four-bit operand fields. The first operand always names a register or a register pair and second operand names a register or a four bit constant. Figure 2.3.2. shows the instruction formats of Ridge 32.

Memory reference instructions have two addressing modes, direct and indexed. These modes can be used in accessing data space as well as code space. Register format instructions consist of only 16 bits. Most of the instructions can be executed in a 125 nanosecond machine cycle, resulting in a maximum instruction rate of 8 MIPS.

There are no condition codes in the Ridge 32, although there are conditional branch instructions. The Ridge compiler sets the branch prediction bit at the time of compilation to inform the instruction fetch unit to prefetch the instructions along the predicted path.

Ridge 32 instructions can be grouped according to their functions as memory reference, integer arithmetic (single,extended precision), real arithmetic (single, double precision), bit manipulation, logical, test, data movement, comparison, sign extended and branch and call instructions.

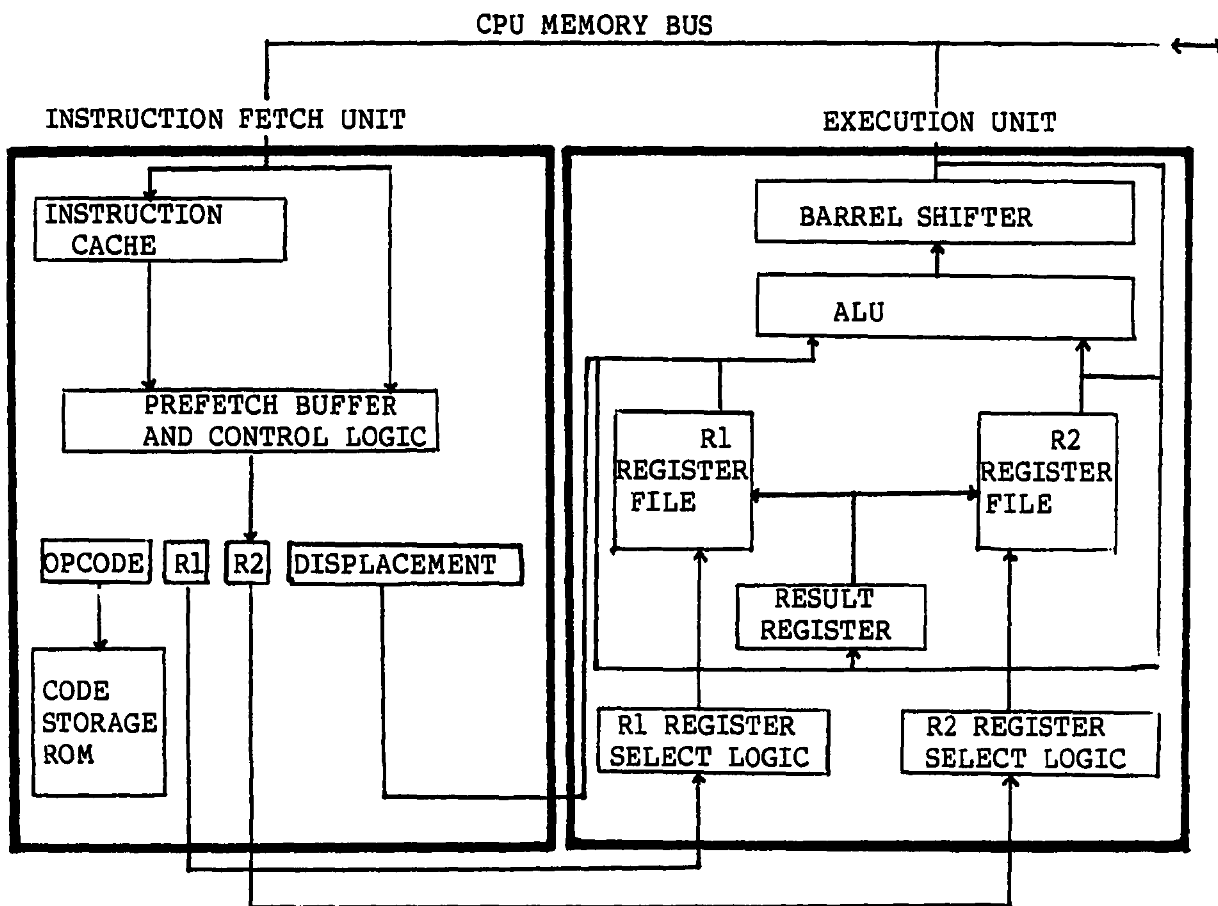
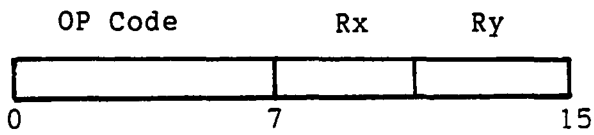
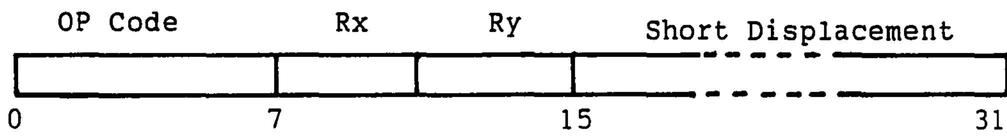


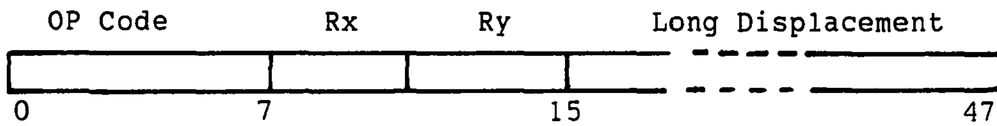
Fig. 2.3.1. Ridge 32 CPU internal structure



Register Format



Memory reference format (Short Displacement)



Memory reference format (Long Displacement)

Figure 2.3.2. Ridge 32 Instruction Formats

If we compare with the previous RISCs, the following points can be considered as deviations from them :

* The Ridge 32 has more than 100 instructions including floating-point instructions.
* It is microcoded and implements virtual memory.

On the other hand, the Ridge 32 shares the following RISC concepts:

* Simple addressing modes. It uses only three modes.
* Simple instruction formats. It uses three instruction formats with three different lengths (16,32, and 48 bits).
* Branch prediction capability.
* No condition codes are used.
* General registers. All 16 32-bit registers are available for use as data, indexing, and addressing.

In addition to the above general RISC characteristics, the Ridge 32 has the following additional characteristics :

* Separated code and data.
  This eliminates the need for logic that detects and resolves self-modifying code.

* Symmetry

  For memory reference instructions there are four operand sizes and three addressing modes and each addressing mode is available for all operands.

* Linear address space

  Code and data space are both linear with a byte-addressable area that is four-gigabytes long.

* To accelerate engineering applications and high speed graphics, a complete set of instructions (including floating point, multiplication, division, and bit manipulation) is performed in hardware.

* To enhance the system's speed and life cycle, the Ridge 32 is designed with multiple bipolar MSI,LSI chips instead of one-chip VLSI architecture.

* To increase processing speed, the Ridge 32 uses a four stage pipeline : instruction fetch, operand fetch, execution, and store the result.

* A 256-byte instruction cache and a maximum of 4096 words of 48-bit wide control store.

CHAPTER III

PERFORMANCE EVALUATION

3.1. PREVIOUS PERFORMANCE EVALUATIONS OF RISC SYSTEMS

Since the introduction of the RISC concept, there has been several performance evaluations of these systems. Since the RISC I and II produced a great impact with this new design approach, they have been the most popular machines for those evaluation studies.

In one of the early simulations of the RISC I[Patt82b], PUZZLE and QUICKSORT benchmark programs were used to verify the effectiveness of window register sets. QUICKSORT has a high density of procedure calls, which is typical in modern structured programs. PUZZLE has a low density of procedure calls but it does have a large nesting depth which will cause more overflows because of limited number of register windows in RISC I. In both cases, they found out that register windows significantly contribute to reduce the memory traffic compared to a VAX 11/780. The effectiveness of delayed jumps was also evaluated and simulation results of program size and execution time of well

known C benchmark programs, such as EDN benchmarks and Towers of Hanoi, were compared to several Minis and Microcomputers (such as MC68000, Z8002, VAX 11/780, PDP 11/70 and C/70).

In another paper, Ditzel and Patterson defined the concept of High Level Computer system [Ditz80]. This means there is no difference to the user of a high level language computer system whether the system is implemented with a CISC or RISC. In this paper, they measured the High Level Language Execution Support Factor(HLLESF) which is defined by the ratio of the execution time of programs in the lowest possible level language to the execution time of the same programs written in a high level language. To compare the results of several architectures, they run a common set of benchmark programs using a single compiler and programming language. Since it is not easy to isolate the implementation as an independent variable from architecture, they do not distinguish these two aspects. For this measurement, a subset of the Computer Family Architecture benchmarks [Full77][Grap81] was selected. They also added Puzzle, Ackerman's Function, SED(a stream editor), QSORT(a sorting program) and Towers of Hanoi, since CFA benchmarks do not include tests for high level language programs. Those programs were programmed in C, compiled in Johnson's Portable C Compiler, and compared with the execution time of assembly language programs to get HLLESF. This study

showed favorable results for RISC I compared to the MC 68000, Z8002, VAX 11/780 and PDP 11/70, where the HLLESF for RISC I was 0.9 on the average and at most 0.5 in the other computers.

In the other paper on the RISC II[Patt84], the floating point arithmetic capability of RISC II was compared with the VAX 11/780 by measuring the execution time of a Whetstone program which was translated into C from Algol-60. Two types of floating point support, software and hardware, were considered in this measurement, through simulation. They concluded that the RISC alone is not an effective vehicle for floating point applications and there is a need to alleviate the heavy overhead between CPU and coprocessor for floating point arithmetic. Large benchmark programs such as the RISC C compiler and the VAX C compiler were used to compare program sizes and execution times.

Since the multiple overlapped register windows in RISC I/II significantly accelerated the execution times in certain benchmark programs and this can not be considered as an exclusive feature of RISC, several papers have tried to decouple the effect of the overlapped register windows to find out the merits of a pure RISC. They have built simulators to decouple the effect of the multiple register sets, and compared the execution time [Heat84] and CPU-memory traffic [Colw85] in benchmark programs with and without register window sets.

J.L. Heath[Heat84] tried to decouple two factors, the register window and the type of compiler, which was not isolated in the performance studies at Berkeley. He pointed out that the existance of register windows and differences in compiler(RISC I programs were compiled with a peephole optimizer whereas portable compilers were used on the other machines) can make a significant difference in the result. In order to eliminate the effect of RISC I's register windows, he coded benchmark programs by hand using only one set of registers and ran this programs on a RISC I simulator written in FORTRAN. In this study, he concluded that hand coded RISC I programs were still comparable to the MC 68000 or Z8000 in terms of program size and execution time, even though the advantages of the RISC I are significantly reduced. It was also found out that dynamic execution statistics reveal that RISC I makes extensive use of a very small subset of its already small instruction set.

The study at Carnegie-Mellon University [Colw85] also measured the effects of multiple register sets. They built simulators for VAX 11/780 and MC 68000 and RISC I without Multiple Register Sets(MRS) and with overlapping MRS. The total amount of processor-memory traffic (bytes read and written) for each benchmark program was used for comparison. Substantial difference in processor-memory traffic for an architecture with

and without MRS was observed and they concluded that any performance claims for reduced instruction set computers that do not remove effects due to MRS are inconclusive.

In work done at the University of Miami[Rals85], execution time and program size of several benchmarks are compared for the RISC,MIPS, and VAX-11/780 by deriving corresponding assembly benchmark programs from the optimized assembler output of the PASCAL compiler for the VAX. By using this method, they tried to avoid the the effects caused by possible variations such as the type of benchmark program, programming language , and compiler. Their results, however, are not based on currently existing compilers or languages for the RISC.

From these performance evaluations on RISCs, we can make the following observations:

* Most of the performance evaluations for RISCs use several well known benchmark programs and compare their execution times and program sizes.
* Most of those studies were made for the RISC I/II and used the C language and assembly language to compare with other machines.
* While there is no complete consensus on the definition of RISC, still more performance evaluation work is required to identify performance gains which result from the merits of the RISC approach.

3.2. PARAMETER SELECTION

What is measured depends on the specific objectives of the evaluation. The following is a list of architectural parameters of interest :

* Program Execution Time

Measure the CPU time elapsed during the benchmark program execution. The result indicates the overall system performance including the effect of the implementation. This parameter can be measured using the system command "TIME" in the case of the Ridge 32 and the run time function "CLOCK" in the case of the VAX. Overhead should be considered, especially in the case of using the system command. When evaluating the result of execution time to measure the contribution of the RISC concept, it is necessary to filter out the effect of branch prediction logic, compiler, pipelining, and instruction cache. In this case, a manual evaluation method[Rals85] can be used to avoid those effects. Since runtime functions can make the result different, all measurements should be done under the same condition. Another important issue here is what programs to use as benchmarks.

* Program Size

Measure the size of the object program in bytes. We can

expect that a RISC uses more space than a CISC. This static program size can be obtained as the result of compilation. A possible problem is that program size should include instructions, indirect addresses and temporary work areas required by the program to strictly compare two computers. It should not include the variable data space.

* Compiler Optimization

Measure the effect of the compiler. Usually RISCs depend heavily upon the efficiency of the compiler[Patt85b] to get high reusability of operands in registers. We can measure this factor by using the register life concept[Lund77] or simply measure the frequency of memory access. A simulator built for the Ridge 32 can be used to measure the dynamic register life or memory access frequency.

* Register Utilization

This parameter is of value by itself, independently of compiler, since it describes how well the number of available registers can satisfy expression evaluation and other calculations.

* Static and Dynamic Instruction Utilization

It can be expected that a RISC has a more balanced

instruction usage than a CISC. Static instruction utilization can be measured by counting the frequency of each instruction in benchmark programs. Dynamic instruction utilization can be measured by using the simulator.

* High Level Language Support Factor

  HLLESF is the ratio of the execution time of a program written in the lowest-level language(usually assembly language) to the execution time of the same program written in the HLL.

  HLLSSF is the ratio of the size of a program required to write and execute (i.e. source, object) written in the assembly language to the size of HLL program.

  HLLPSF is the ratio of the preparation time (i.e. compilation, linking, loading) for the assembly language to the program preparation time of HLL.

  We can measure the High-Level Language Execution Support Factor (HLLESF), HLL Size Support Factor (HLLSSF) and HLL Preparation time Support Factor (HLLPSF). HLLSSF and HLLPSF are less important than HLLESF[Ditz80].

* Addressing mode utilization

  This is of particular interest in RISC studies since this type of machines have few addressing modes. Especially interesting is the effect of high-level language operations

which may result in several RISC instructions for lack of a more complex mode.

* Loop behavior

  What is the average lengh of loops in a set of programs ? This is useful for determining the size of the instruction cache.

* Average Distance Between Branches

  This is useful for deciding on the size of the instruction buffer in the CPU.

3.3. MEASUREMENT METHODS

The way to determine these parameters is obviously very important. The following is a list of some approaches that have been used :

* Benchmarks on actual machines

  These determine execution time but include effect of implementation. They also require access to the machine, which may not be the case for new designs.

* Simulators

  An instruction level simulator of the architecture is implemented and an object program is run interpretively on it. This is one of the most flexible methods[Barb77]. Its main limitation is speed.

* Addition of Instruction Times

  A given program is analyzed considering each instruction execution time. This can be used to determine execution time. It hides, however, the effect of pipelining and instruction overlap.

* Microcoded Probes

This requires access to the machine microcode. It is used mainly by computer designers who have access to the microcode.

* Maintenance Processors

  These are used mainly for reliability but can be easily adapted to measure performance.

3.4. RIDGE 32 SIMULATOR

The Ridge 32 simulator was programmed in PASCAL under ROS(Ridge Operating System).

Since detailed data on the instruction fetch unit is not available at the time of this study, branch prediction logic is not considered in this thesis. Branch prediction logic can affect the instruction flow in the pipeline which results in variations in instruction execution time.

It was found out from the preliminary study that some of the real arithmetic instructions and bit manipulation instructions are never used in any target benchmark programs in this study. Since some of these instructions do not affect the dynamic program flow of the target programs, the execution of these instructions can be safely simplified, so we just increase their instruction

counter and update the program counter to the next instruction without any memory or register operations.

The simulator takes hexadecimal object code of a target program as a input and generates either static or dynamic instruction usage statistics for the assembly code. It also can provide the following data :

* Distributions of register-to-register instructions, short-displacement instructions and long-displacement instructions in the target program.
* Distribution of direct memory access instructions versus indexed memory access instructions in all memory access instructions.
* Total number of all branch instructions executed and distribution of conditional branch and unconditional branch instructions.
* Distribution of code space access instructions versus data space access instructions in all executed memory access instructions.
* Average length of loops in bytes that appear in the target program.
* Average value of the offsets of the branch instructions in bytes of the target program.

Due to the flexibility of the simulator, we can easily expand this simulator to get further data for future studies.

Implementation of The Ridge 32 Simulator

We present here a brief discussion of the representation of the Ridge 32 in the Ridge 32 simulator.

There are 16 32-bit general purpose registers in the Ridge 32 computer. These registers are implemented as a two dimensional array which is defined as array [0..15,0..31] of integer in the simulator. For example, the third bit of the Second Register is represented by R [2,3] where the first parameter indicates the register and the second parameter indicates the bit. Each bit of the register is of integer type and logic 0 or 1 will be expressed as integer "0" or "1".

The code memory of the Ridge 32 is implemented as CodeMemory [1..MaxCodeMemory] of ByteType in the simulator where ByteType is defined as array [0..1] of char. Once an hexadecimal object code of the benchmark program is obtained, the simulator reads this object code file into the CodeMemory in units of two characters. These characters in the CodeMemory will be interpreted later in the process of simulation.

The program counter is implemented as integer PC and any byte in the CodeMemory can be accessed as CodeMemory [PC]. Data memory is defined as DataMemory similarly to the CodeMemory. Virtual memory is not implemented in the current experiment (A detailed description of virtual memory implementation was not available).

As in the Ridge 32 computer, two's complement form is used to represent integers.

There is an instruction counter for each instruction of the Ridge 32 in this simulator to trace the usage of every instruction during simulation. This counter will be used to provide the statistics of the simulation result.

A flowchart diagram of the Ridge 32 simulator is shown in Figure 3.4.2.

We present below brief descriptions of the major procedures or functions in this simulator.

INPUT PART

* Procedure FirstPass

  INPUT : "XX.o" file of the target program.

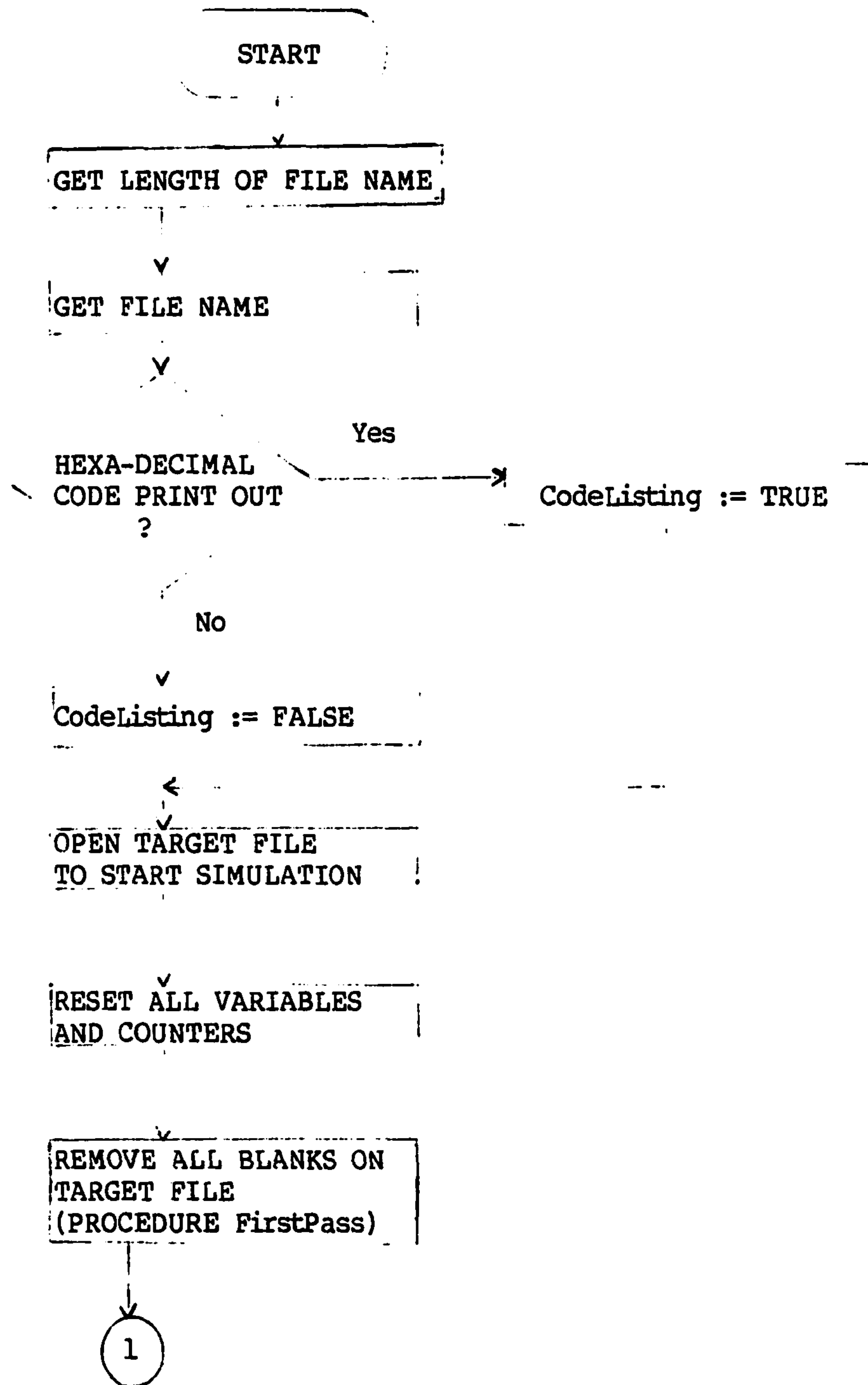


Figure 3.4.1. Flowchart Diagram of Ridge 32 Simulator

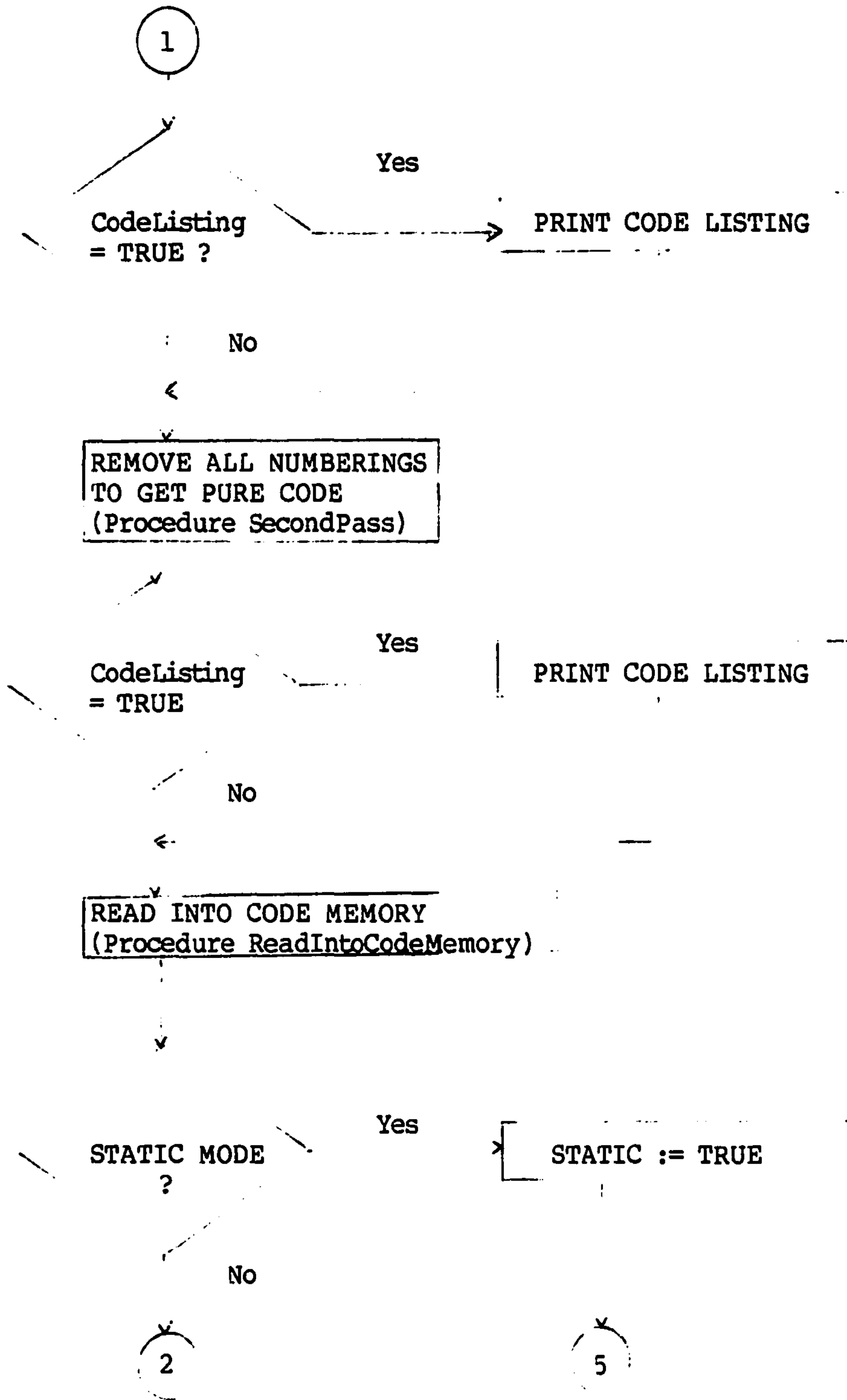


Figure 3.4.1. Flowchart Diagram of Ridge 32 Simulator
(Continued)

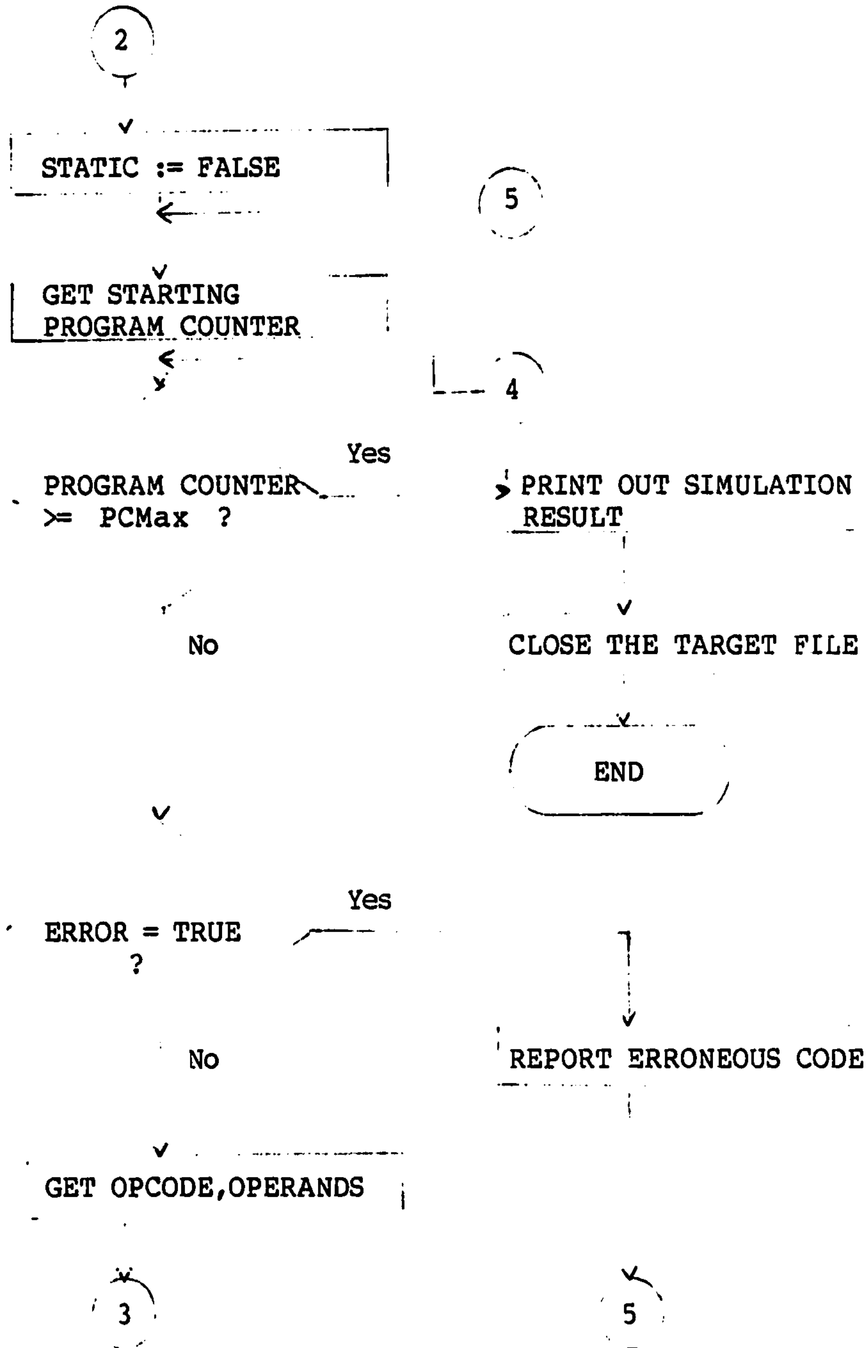


Figure 3.4.1. Flowchart Diagram of Ridge 32 Simulator (Continued)

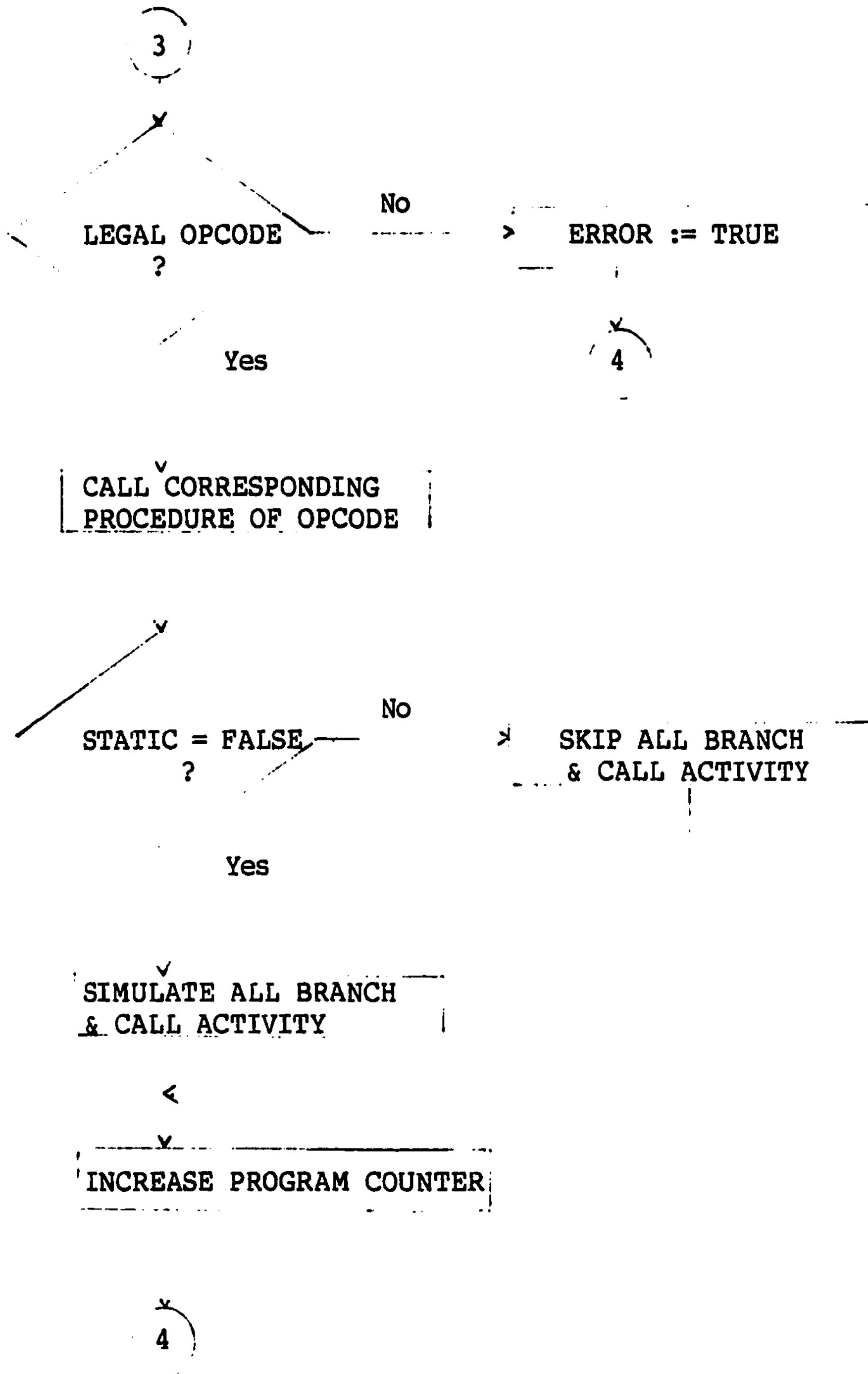


Figure 3.4.1. Flowchart Diagram of Ridge 32 Simulator (Continued)

OUTPUT : FirstPassMemory, I1Max.

OBJECTIVE : Remove any blanks and carriage return characters in "XX.o" file when reading "XX.o" file into FirstPassMemory.

* Procedure SecondPass

INPUT : FirstPassMemory, I1Max.

OUTPUT : FirstPassMemory, I2Max.

OBJECTIVE : Remove all numberings in FirstPassMemory to get just pure hexadecimal code.

* Procedure ReadIntoCodeMemory

INPUT : FirstPassMemory, I2Max.

OUTPUT : CodeMemory, PCMax.

OBJECTIVE : To save code in FirstPassMemory in CodeMemory in the unit of byte type.

EXECUTION PART

* Procedure Execution

INPUT : CodeMemory, PCMax.

OBJECTIVE : Execute sub-procedures which correspond to each OP code fetched from the CodeMemory.

* Procedure CharToBit

INPUT : One character in hexadecimal form.

OUTPUT : Four bit corresponding to the input character.

OBJECTIVE : Convert one hexadecimal character into the corresponding four-bit binary value. Bit 0 is the LSB(Least Significant Bit).

* Procedure SixteenBitTwos

INPUT : A Sixteen bit Variable.

OUTPUT : Two's complement of the input variable.

OBJECTIVE : To return the two's complement output of the input variable.

* Procedure GetShortAdd

INPUT : Short displacement address in the instruction.

OUTPUT : Effective address in integer format which corresponds to the short displacement in the instruction.

OBJECTIVE : To get the effective address of the memory access instructions in short format.

* Procedure ThirtyTwoBitTwos

INPUT : A Thirty two bit variable.

OUTPUT : Two's complement of the input.

OBJECTIVE : To return the two's complement value of the input.

* Procedure GetLongAdd

INPUT : Long displacement address in the instruction.

OUTPUT : Effective address in the integer format which corresponds to the long displacement of the instruction.

* Function Value

INPUT : One character.

OUTPUT : Integer value corresponding to the input.

OBJECTIVE : To return the integer value which corresponds to input character.

* Procedure LoadByte

INPUT : A byte type variable, register index.

OUTPUT : Loaded register corresponding to the input register index.

OBJECTIVE : To load a byte into bit 0 to bit 7 of the register which corresponds to the input register index. Bit 0 is the LSB.

* Procedure LoadHalfWord

INPUT : A half word type variable, register index.

OUTPUT : Loaded register corresponding to the input register index.

OBJECTIVE : To load a half word into bit 0 to bit 15 of the register which coresponds to the input register index. Bit 0 is the LSB.

* Procedure LoadWord

INPUT : A word type variable, register index.

OUTPUT : Loaded register corresponding to the input register index.

OBJECTIVE : To load a word into bit 0 to bit 31 of the register which corresponds to the input register index. Bit 0 is the LSB.

* Procedure IntegerToChar

INPUT : A variable of integer type.

OUTPUT : A character which corresponds to the input.

OBJECTIVE : To return a charcter variable which corresponds to the input integer value.

* Procedure GetByte

INPUT : Register index.

OUTPUT : A byte type variable.

OBJECTIVE : To return a byte-typed variable which corresponds to bit 0 to bit 7 of the indexed register.

* Procedure GetHalfWord

INPUT : Register index.

OUTPUT : A half word type variable.

OBJECTIVE : To return a half word variable which corresponds to bit 0 to bit 15 of the indexed register.

* Procedure GetWord

INPUT : Register index.

OUTPUT : A word type variable.

OBJECTIVE : To return a word variable which corresponds to bit 0 to bit 31 of the indexed register.

* Function RegisterValue

INPUT : Register Index.

OUTPUT : Integer value which corresponds to the contents of the indexed register.

OBJECTIVE : To return the integer value of the indexed register.

* Procedure Base10To2

INPUT : A integer variable.

OUTPUT : Corresponding 32 bit binary variable.

OBJECTIVE : To get the binary value from the decimal input value.

OUTPUT PART

* Procedure PrintStatistics

  INPUT : All counters of the Ridge 32 assembly instructions.

  OBJECTIVE : To calulate the various statistical data from all counters and print out the result.

Appendix B shows the typical scenario to simulate the target benchmark program "XX.p" (where "XX" means any file name) written in Pascal and the typical dialog between the user and the Ridge 32 simulator as it happens during the simulation of a benchmark program. Appendix C shows the simulator listing.

3.5. MEASUREMENT RESULTS

In this thesis, DHRYSTONE, WHETSTONE, SIEVE, PUZZLE, ACKERMAN, QUICKSORT and SEARCH programs are used as benchmarks to measure different aspects of computer performance. Appendix A includes the listing of the above benchmark programs. These benchmarks were run on the Ridge 32 and then on the Ridge 32 simulator. This simulator was also run on the Ridge 32.

DHRYSTONE is a synthetic benchmark program which is based on statistical data about the actual usage of programming language features in systems programming, including static and dynamic measurements[Weic84]. They made a table of the frequency of occurrence of the different types of high-level language statements in both static and dynamic use. Instruction frequency distribution of the Dhrystone benchmark program reflects those analysis results. Data types and operations in Dhrystone were selected to represent the area of systems programming rather than numerical programming or data processing applications.

SIEVE is the Sieve of Eratosthenes program that is well known in computer science[Gilb81]. It measures the CPU's ability to handle Boolean arrays.

PUZZLE, developed at Stanford University, measures the

execution speed of a CPU on integer arithmetic,indexed operations,procedure calls, and looping.

ACKERMAN is a program to compute the value of Ackerman's function for arguments(3,6). It spends most of its time making recursive calls to itself.

QUICKSORT performs a sort algorithm on a worst-case array of 100 real numbers.

WHETSTONE is a well known program which measures the performance of floating point operations.

All of these programs are written in Pascal. Since an optimized Pascal compiler is not available for this system, the object code from the Pascal compiler is not optimized. The Pascal source file is translated into P-code by "PASC" command and the P-code is translated into Ridge Assembly Language by "PTRANS" command. (These are ROS commands.) The resulting object files are linked to make a unified executable file. All these programs are running under ROS(Ridge Operating System) which is derived from UNIX system 5, releases one and two, Berkeley Software Distribution 4.1 and 4.2, and other sources. Figure 3.5.1 shows the relationships between those commands and the resulting files. "XX.p" indicates the Pascal source file and "XX.q" is the corresponding P-code of the "XX.p". "XX.o"

```
             "PASC"             "PTRANS"           "LINK"
"XX.p"   ======> "XX.q"   ======> "XX.o" ======> Executable file
```

Figure 3.5.1 Relationships between system command and files

contains the Ridge 32 assembly code in hexadecimal form and will be used as an input to the Ridge 32 simulator.

EXECUTION TIME

To measure execution times, the system command "TIME" was used. (The Pascal run time function "CLOCK" was not available at this time of measurement). This command tells us the elapsed time during command and the command execution time.

To avoid the effect of overhead, every benchmark was executed 100 times using a FOR loop and measured 10 times to get the mean run time. To measure the overhead caused by the FOR loop itself, the execution time for one million empty loops was measured. From this measurement, 300 nanosec was obtained as overhead time due to the FOR loop. If we consider the actual execution time of the benchmark and the precision of our measurement method, this overhead time can be safely neglected.

Table 3.5.1 shows the result of execution time measurement. This measurement was done without runtime check.

BENCHMARK PROGRAM	EXECUTION TIME (msec)
Dhrystone	0.55
Sieve	76.9
Puzzle	3689.0
Ackerman	1242.0
Quicksort	13.0
Whetstone	699.0

Table 3.5.1. Benchmark Program Execution Time on Ridge 32

PROGRAM SIZE

Table 3.5.2. shows the program size in units of bytes for each benchmark program. Object code size is measured in the following way :

* Compile the Pascal source file "XX.p" using Ridge 32 system command "PASC" to get the "XX.q" file.
* Use system command PTRANS to get the hexadecimal code "XX.o".
* Measure the program size using the simulator which gives the object code size as the result of SecondPass procedure.

In this experiment, program size just includes the size of the benchmark program itself and does not include the indirect addresses and temporary work areas which are required by the program.

It was found that the Puzzle benchmark program has the longest program size with 3484 bytes whereas Ackerman's function benchmark is the shortest one with only 278 bytes. The Dhrystone synthetic benchmark program has a relatively large code size (2520 bytes), which is the second largest of these six benchmark programs.

BENCHMARK PROGRAM	PROGRAM SIZE (Bytes)
Dhrystone	2520
Sieve	486
Puzzle	3484
Ackerman	278
Quicksort	1190
Whetstone	2502

Table 3.5.2. Program Size of Pascal Benchmarks on Ridge 32

STATIC AND DYNAMIC INSTRUCTION UTILIZATION

From this experiment, detailed statistics regarding the individual instructions of the Ridge 32 were obtained.

Table 3.5.3 shows the statistics for static instruction utilization and Table 3.5.4. shows the dynamic instruction utilization. Numbers in each instruction row show how often this specific instruction appeared during either static or dynamic simulation for the benchmark program of that column.

Numbers in parenthesis tell us the distribution of that instruction in percent among the total executed instructions in the target benchmark program.

The rightmost column indicates statistics over all six benchmark programs used in this measurement. To get this data, the number of instruction utilization of each benchmark is added.

Instructions are grouped as Memory Reference, Integer Arithmetic, Real Arithmetic, Bit Manipulation, Logical, Test, Data Movement, Comparison, Sign Extended and Branch & Call instructions.

Memory Reference instructions are further sorted as Load, Store and Load Address instructions.

Integer Arithmetic instructions are sorted as Single-Precision and Extended-Precision.

Table 3.5.3. Simulation Result of Static Instruction Usage

	WHETSTONE STATIC	DHRYSTONE STATIC	QSORT STATIC	PUZZLE STATIC	ACKERMAN STATIC	SIEVE STATIC	TOTAL STATIC
TOTAL INSTRUCTIONS	758	641	356	1100	75	126	3056
MEMORY REFERENCE	381(50.26)	372(58.03)	184(51.69)	506(46.00)	41(54.67)	71(56.35)	1555(50.88)
LOAD	191(25.20)	218(34.01)	108(30.34)	216(19.64)	18(24.00)	32(25.40)	783(25.62)
LoadB	1(0.13)	60(9.36)	2(0.56)	97(8.82)	0	1(0.79)	161(5.27)
LoadbS	0	16	0	73	0	0	89
LoadbIS	1	44	2	24	0	1	72
LoadbL	0	0	0	0	0	0	0
LoadbIL	0	0	0	0	0	0	0
LoadH	0	0	0	21(1.91)	0	0	21(0.69)
LoadhS	0	0	0	4	0	0	4
LoadhIS	0	0	0	17	0	0	17
LoadhL	0	0	0	0	0	0	0
LoadhIL	0	0	0	0	0	0	0
Load	142(18.74)	75(11.70)	85(23.88)	45(4.09)	11(14.67)	11(8.73)	369(12.07)
LoadS	116	12	4	32	2	6	172
LoadIS	26	63	81	13	9	5	197
LoadL	0	0	0	0	0	0	0
LoadIL	0	0	0	0	0	0	0
LoadD	0	0	0	0	0	0	0
LoaddS	0	0	0	0	0	0	0
LoaddIS	0	0	0	0	0	0	0
LoaddL	0	0	0	0	0	0	0
LoaddIL	0	0	0	0	0	0	0
Laddr	48(6.33)	76(11.86)	21(5.90)	53(4.82)	7(9.33)	19(15.08)	224(7.33)
LaddrS	11	54	8	31	2	15	121
LaddrIS	1	5	5	12	2	0	25
LaddrL	29	4	1	5	1	3	43
LaddrIL	7	13	7	5	2	1	35
LaddrP	0	0	0	0	0	0	0
LaddrpS	0	0	0	0	0	0	0
LaddrpIS	0	0	0	0	0	0	0
LaddrpL	0	0	0	0	0	0	0
LaddrpIL	0	0	0	0	0	0	0

Table 3.5.3. Simulation Result of Static Instruction Usage (Continued)

LoadBP	0	3(0.47)	0	0	0	0	3(0.10)
LoadbpS	0	0	0	0	0	0	0
LoadbpIS	0	3	0	0	0	0	3
LoadbpL	0	0	0	0	0	0	0
LoadbpIL	0	0	0	0	0	0	0
LoadHP	0	0	0	0	0	0	0
LoadhpS	0	0	0	0	0	0	0
LoadhpIS	0	0	0	0	0	0	0
LoadhpL	0	0	0	0	0	0	0
LoadhpIL	0	0	0	0	0	0	0
LoadP	0	4(0.62)	0	0	0	1(0.79)	5(0.16)
LoadpS	0	0	0	0	0	0	0
LoadpIS	0	4	0	0	0	1	5
LoadpL	0	0	0	0	0	0	0
LoadpIL	0	0	0	0	0	0	0
LoadDP	0	0	0	0	0	0	0
LoaddpS	0	0	0	0	0	0	0
LoaddpIS	0	0	0	0	0	0	0
LoaddpL	0	0	0	0	0	0	0
LoaddpIL	0	0	0	0	0	0	0
STORE	190(25.07)	154(24.02)	76(21.35)	290(26.36)	23(30.67)	39(30.95)	772(25.26)
StoreB	1(0.13)	73(11.39)	5(1.40)	165(15.00)	0	2(1.59)	246(8.05)
StorebS	0	16	0	130	0	0	146
StorebIS	1	57	5	35	0	2	100
StorebL	0	0	0	0	0	0	0
StorebIL	0	0	0	0	0	0	0
StoreH	0	0	0	40(3.64)	0	0	40(1.31)
StorehS	0	0	0	19	0	0	19
StorehIS	0	0	0	21	0	0	21
StorehL	0	0	0	0	0	0	0
StorehIL	0	0	0	0	0	0	0
Store	189(24.93)	81(12.64)	71(19.94)	85(7.73)	23(30.67)	37(29.37)	486(15.90)
StoreS	94	7	4	57	5	18	185
StoreIS	95	70	67	28	18	18	296
StoreL	0	0	0	0	0	0	0
StoreIL	0	4	0	0	0	1	5
StoreD	0	0	0	0	0	0	0
StoredS	0	0	0	0	0	0	0
StoredIS	0	0	0	0	0	0	0
StoredL	0	0	0	0	0	0	0
StoredIL	0	0	0	0	0	0	0

Table 3.5.3. Simulation Result of Static Instruction Usage (Continued)

INTEGER ARITH.	51(6.73)	65(10.14)	28(7.87)	150(13.64)	5(6.67)	8(6.35)	307(10.05)
SINGLE-PRECI.	51(6.73)	65(10.14)	28(7.87)	150(13.64)	5(6.67)	8(6.35)	307(10.05)
Add	28(3.69)	44(6.86)	23(6.46)	120(10.91)	2(2.67)	8(6.35)	225
Add Reg.	5	16	15	58	0	3	97
Add I.	23	28	8	62	2	5	128
Div	0	1(0.16)	0	0	0	0	1
Neg	0	0	0	0	0	0	0
Mpy	15(1.98)	7(1.09)	0	29(2.64)	0	0	51
Mpy Reg.	15	7	0	29	0	0	51
Mpy I.	0	0	0	0	0	0	0
Rem	0	0	0	0	0	0	0
Sub	8(1.06)	13(2.03)	5(1.40)	1(0.09)	3(4.00)	0	30
Sub Reg.	8	5	1	0	0	0	14
Sub I.	0	8	4	1	3	0	16
EXTENDED PRECI.	0	0	0	0	0	0	0
Eadd	0	0	0	0	0	0	0
Ediv	0	0	0	0	0	0	0
Empy	0	0	0	0	0	0	0
Esub	0	0	0	0	0	0	0
REAL ARITH.	90(11.87)	0	1(0.28)	0	0	0	91(2.98)
SINGLE PRECI.	90(11.87)	0	1(0.28)	0	0	0	91(2.98)
Rneg	8(1.06)	0	0	0	0	0	8
Radd	33(4.35)	0	0	0	0	0	33
Rsub	14(1.85)	0	0	0	0	0	14
Rmpy	24(3.17)	0	0	0	0	0	24
Rdiv	6(0.79)	0	0	0	0	0	6
Float	5(0.66)	0	1(0.28)	0	0	0	6
Fixr	0	0	0	0	0	0	0
Fixt	0	0	0	0	0	0	0
Makerd	0	0	0	0	0	0	0

DOUBLE PRECI.	0	0	0	0	0	0	0
Dfixr	0	0	0	0	0	0	0
Dfixt	0	0	0	0	0	0	0
Dfloat	0	0	0	0	0	0	0
Dradd	0	0	0	0	0	0	0
Drdiv	0	0	0	0	0	0	0
Drmpy	0	0	0	0	0	0	0
Drneg	0	0	0	0	0	0	0
Drsub	0	0	0	0	0	0	0
Makedr	0	0	0	0	0	0	0
BIT MANIPULATION	7(0.92)	10(1.56)	23(6.46)	7(0.64)	0	0	47(1.54)
Asl	0	0	0	0	0	0	0
Asl Reg.	0	0	0	0	0	0	0
Asl I.	0	0	0	0	0	0	0
Asr	0	0	1(0.28)	0	0	0	1
Asr Reg.	0	0	0	0	0	0	0
Asr I.	0	0	1	0	0	0	1
Cbit	0	0	0	0	0	0	0
Csl	0	0	0	0	0	0	0
Csl Reg.	0	0	0	0	0	0	0
Csl I.	0	0	0	0	0	0	0
Dlsl	0	0	0	0	0	0	0
Dlsl Reg.	0	0	0	0	0	0	0
Dlsl I.	0	0	0	0	0	0	0
Dlsr	0	0	0	0	0	0	0
Dlsr Reg.	0	0	0	0	0	0	0
Dlsr I.	0	0	0	0	0	0	0
Lsl	7(0.92)	10(1.56)	22(6.18)	7(0.64)	0	0	46
Lsl Reg.	0	0	0	0	0	0	0
Lsl I.	7	10	22	7	0	0	46
Lsr	0	0	0	0	0	0	0
Lsr Reg.	0	0	0	0	0	0	0
Lsr I.	0	0	0	0	0	0	0
Sbit	0	0	0	0	0	0	0
Tbit	0	0	0	0	0	0	0

Table 3.5.3. Simulation Result of Static Instruction Usage (Continued)

Table 3.5.3. Simulation Result of Static Instruction Usage (Continued)

LOGICAL INS.	1(0.13)	4(0.62)	2(0.56)	1(0.09)	0	0	8(0.26)
And	0	1(0.16)	2(0.56)	0	0	0	3
And Reg.	0	1	2	0	0	0	3
And I.	0	0	0	0	0	0	0
Not	1	2	0	0	0	0	3
Or	0	1(0.16)	0	1(0.09)	0	0	2
Xor	0	0	0	0	0	0	0
TEST INSTRUCTION	0	4(0.62)	6(1.69)	1(0.09)	0	0	11(0.36)
TestGT	0	2	1	0	0	0	3
TestLT	0	0	1	0	0	0	1
TestEQ	0	1	0	1	0	0	2
TestIGT	0	0	1	0	0	0	1
TestILT	0	0	1	0	0	0	1
TestIEQ	0	0	0	0	0	0	0
TestLE	0	1	0	0	0	0	1
TestGE	0	0	0	0	0	0	0
TestNE	0	0	0	0	0	0	0
TestILE	0	0	1	0	0	0	1
TestIGE	0	0	1	0	0	0	1
TestINE	0	0	0	0	0	0	0
DATA MOVEMENT	162(21.37)	82(12.79)	60(16.85)	296(26.91)	12(16.00)	25(19.84)	637(20.84)
Move Reg.	123	36	46	106	4	12	327
Move I.	39	46	14	190	8	13	310
COMPARISON INST.	0	0	3(0.84)	0	0	0	3
Dcomp	0	0	0	0	0	0	0
Drcomp	0	0	0	0	0	0	0
Lcomp	0	0	0	0	0	0	0
Rcomp	0	0	3	0	0	0	3
SIGN EXTEND	0	0	0	0	0	0	0
Seb	0	0	0	0	0	0	0
Seh	0	0	0	0	0	0	0

Table 3.5.3. Simulation Result of Static Instruction Usage (Continued)

BRANCH & CALL	58(7.58)	82(12.37)	42(11.57)	134(12.13)	15(19.48)	20(15.62)	351(11.31)
BRANCH	29(3.79)	50(7.54)	24(6.61)	119(10.77)	7(9.09)	11(8.59)	240(7.73)
BrGTS	0	2	0	0	0	1	3
BrGTL	0	0	0	0	0	0	0
BrEQS	0	1	0	0	0	0	1
BrEQL	0	0	0	0	0	0	0
BrIGTS	0	1	0	0	0	0	1
BrIGTL	0	0	0	0	0	0	0
BrILTS	0	0	0	0	0	0	0
BrILTL	0	0	0	0	0	0	0
BrIEQS	0	2	0	2	0	0	4
BrIEQL	0	0	0	0	0	0	0
BrLES	10	5	0	51	1	3	76
BrLEL	0	0	0	0	0	0	0
BrNES	10	11	5	51	1	3	81
BrNEL	0	0	0	0	0	0	0
BrS	4	25	6	6	2	2	45
BrL	1	1	1	1	1	1	6
BrILES	1	0	0	0	0	0	1
BrILEL	0	0	0	0	0	0	0
BrIGES	2	0	1	0	0	0	3
BrIGEL	0	0	0	0	0	0	0
BrINES	1	2	5	8	2	1	19
BrINEL	0	0	0	0	0	0	0
CALL	29(3.79)	32(4.83)	18(4.96)	15(1.36)	8(10.39)	9(7.03)	111(3.58)
CallS	13	10	14	7	4	0	48
CallL	16	21	4	8	4	9	62
Callr	0	1	0	0	0	0	1
LOOP	1(0.13)	10(1.51)	0	0	0	1(0.78)	12(0.39)
LoopS	0	10	0	0	0	1	12
LoopL	0	0	0	0	0	0	0
RET	7(0.91)	12(1.81)	7(1.93)	5(0.45)	2(2.60)	1(0.78)	34(1.10)

Table 3.5.4. Simulation Result of Dynamic Instruction Usage

	WHETSTONE DYNAMIC	DHRYSTONE DYNAMIC	QSORT DYNAMIC	PUZZLE DYNAMIC	ACKERMAN DYNAMIC	SIEVE DYNAMIC	TOTAL
TOTAL INSTRUCTIONS	815609	1489	35426	11876265	3444694	224801	16398284
MEMORY REFERENCE	357929(43.88)	733(49.23)	12067(34.06)	2710282(22.82)	2152176(62.48)	85943(38.23)	5319130(32.44 %)
LOAD	252175(30.92)	426(28.61)	8575(24.21)	2293995(19.32)	1291001(37.48)	43918(19.54)	3890090(23.72 %)
LoadB	2240(0.24)	152(10.21)	273(0.77)	1714799(14.44)	0	8191(3.64)	1725655(10.52 %)
LoadbS	0	21	0	472	0	0	493
LoadbIS	2240	131	273	1714327	0	8191	1725162
LoadbL	0	0	0	0	0	0	0
LoadbIL	0	0	0	0	0	0	0
LoadH	0	0	0	235547(1.98)	0	0	235547(1.44 %)
LoadhS	0	0	0	16	0	0	16
LoadhIS	0	0	0	235531	0	0	235531
LoadhL	0	0	0	0	0	0	0
LoadhIL	0	0	0	0	0	0	0
Load	206710(25.34)	74(4.96)	7551(21.31)	116647(0.98)	947034(27.49)	18806(8.37)	1296822(7.91 %)
LoadS	100038	12	4	3500	2	18801	122357
LoadIS	106672	62	7547	113147	947032	5	1174465
LoadL	0	0	0	0	0	0	0
LoadIL	0	0	0	0	0	0	0
LoadD	0	0	0	0	0	0	0
LoaddS	0	0	0	0	0	0	0
LoaddIS	0	0	0	0	0	0	0
LoaddL	0	0	0	0	0	0	0
LoaddIL	0	0	0	0	0	0	0
Laddr	43225(5.30)	78(5.24)	751(2.12)	227002(1.91)	343967(9.99)	16916(7.52)	631939 (3.85 %)
LaddrS	27117	52	67	185661	2	16912	229811
LaddrIS	140	5	288	6667	171730	0	178830
LaddrL	667	4	1	1	1	3	677
LaddrIL	15301	17	395	34673	172234	1	222621
LaddrP	0	0	0	0	0	0	0
LaddrpS	0	0	0	0	0	0	0
LaddrpIS	0	0	0	0	0	0	0
LaddrpL	0	0	0	0	0	0	0
LaddrpIL	0	0	0	0	0	0	0

Table 3.5.4. Simulation Result of Dynamic Instruction Usage (Continued)

LoadBP	0	90(6.04)	0	0	0	0	90 (0.00 %)
LoadbpS	0	0	0	0	0	0	0
LoadbpIS	0	90	0	0	0	0	90
LoadbpL	0	0	0	0	0	0	0
LoadbpIL	0	0	0	0	0	0	0
LoadHP	0	0	0	0	0	0	0
LoadhpS	0	0	0	0	0	0	0
LoadhpIS	0	0	0	0	0	0	0
LoadhpL	0	0	0	0	0	0	0
LoadhpIL	0	0	0	0	0	0	0
LoadP	0	32(2.15)	0	0	0	5(0.00)	37 (0.00 %)
LoadpS	0	0	0	0	0	0	0
LoadpIS	0	32	0	0	0	5	37
LoadpL	0	0	0	0	0	0	0
LoadpIL	0	0	0	0	0	0	0
LoadDP	0	0	0	0	0	0	0
LoaddpS	0	0	0	0	0	0	0
LoaddpIS	0	0	0	0	0	0	0
LoaddpL	0	0	0	0	0	0	0
LoaddpIL	0	0	0	0	0	0	0
STORE	105754(12.97)	307(20.62)	3492(9.86)	416287(3.51)	861175(25.00)	42025(18.69)	1429040 (8.71 %)
StoreB	2240(0.27)	199(13.36)	383(1.08)	194539(1.64)	0	23190(10.32)	220551 (1.34 %)
StorebS	0	17	0	604	0	0	621
StorebIS	2240	182	383	193935	0	23190	219930
StorebL	0	0	0	0	0	0	0
StorebIL	0	0	0	0	0	0	0
StoreH	0	0	0	110823(0.93)	0	0	110823 (0.68 %)
StorehS	0	0	0	43	0	0	43
StorehIS	0	0	0	110780	0	0	110780
StorehL	0	0	0	0	0	0	0
StorehIL	0	0	0	0	0	0	0
Store	103514(12.69)	108(7.25)	3109(8.78)	110925(0.93)	861175(25.00)	18835(8.38)	1097666 (6.69 %)
StoreS	22060	6	4	3522	5	18812	44409
StoreIS	81454	70	3105	107403	861170	18	1053220
StoreL	0	0	0	0	0	0	0
StoreIL	0	32	0	0	0	5	37
StoreD	0	0	0	0	0	0	0
StoredS	0	0	0	0	0	0	0
StoredIS	0	0	0	0	0	0	0
StoredL	0	0	0	0	0	0	0
StoredIL	0	0	0	0	0	0	0

Table 3.5.4. Simulation Result of Dynamic Instruction Usage (Continued)

INTEGER ARITH.	59055(7.24)	356(23.91)	2745(7.75)	3065835(25.81)	258099(7.49)	38978(17.34)	3425068 (20.89 %)
SINGLE-PRECI.	59055(7.24)	356(23.91)	2745(7.75)	3065835(25.81)	258099(7.49)	38978(17.34)	3425068 (20.89 %)
Add	33844(4.14)	332(22.30)	2295(6.48)	3062054(25.78)	85867(2.49)	38978(17.34)	3223370 (19.66 %)
Add Reg.	6302	20	840	1637143	0	18797	1663102
Add I.	27542	312	1455	1424911	85867	20181	1560268
Div	0	1(0.07)	0	0	0	0	1
Neg	0	0	0	0	0	0	0
Mpy	14708(1.80)	8(0.54)	0	399(0.00)	0	0	15115 (0.09 %)
Mpy Reg.	14708	8	0	399	0	0	15115
Mpy I.	0	0	0	0	0	0	0
Rem	0	0	0	0	0	0	0
Sub	10503(1.29)	15(1.01)	450(1.27)	3382(0.03)	172232(5.00)	0	186582 (1.14 %)
Sub Reg.	10503	5	61	0	0	0	10569
Sub I.	0	10	389	3382	172232	0	176013
EXTENDED PRECI.	0	0	0	0	0	0	0
Eadd	0	0	0	0	0	0	0
Ediv	0	0	0	0	0	0	0
Empy	0	0	0	0	0	0	0
Esub	0	0	0	0	0	0	0
REAL ARITH.	89411(10.96)	0	101(0.29)	0	0	0	89512 (0.55 %)
SINGLE PRECI.	89411(10.96)	0	101(0.29)	0	0	0	89512
Rneg	847(0.10)	0	0	0	0	0	847
Radd	36779(4.51)	0	0	0	0	0	36779
Rsub	4283(0.53)	0	0	0	0	0	4283
Rmpy	23748(2.91)	0	0	0	0	0	23748
Rdiv	10562(1.29)	0	0	0	0	0	10562
Float	13192(1.62)	0	101(0.29)	0	0	0	13293
Fixr	0	0	0	0	0	0	0
Fixt	0	0	0	0	0	0	0
Makerd	0	0	0	0	0	0	0

Table 3.5.4. Simulation Result of Dynamic Instruction Usage

(Continued)

DOUBLE PRECI.	0	0	0	0	0	0	0
Dfixr	0	0	0	0	0	0	0
Dfixt	0	0	0	0	0	0	0
Dfloat	0	0	0	0	0	0	0
Dradd	0	0	0	0	0	0	0
Drdiv	0	0	0	0	0	0	0
Drmpy	0	0	0	0	0	0	0
Drneg	0	0	0	0	0	0	0
Drsub	0	0	0	0	0	0	0
Makedr	0	0	0	0	0	0	0
BIT MANIPULATION	35000(4.29)	11(0.74)	3197(9.02)	1361017(11.46)	0	0	1399225 (8.53 %)
Asl	0	0	0	0	0	0	0
Asl Reg.	0	0	0	0	0	0	0
Asl I.	0	0	0	0	0	0	0
Asr	0	0	30(0.08)	0	0	0	30
Asr Reg.	0	0	0	0	0	0	0
Asr I.	0	0	30	0	0	0	30
Cbit	0	0	0	0	0	0	0
Csl	0	0	0	0	0	0	0
Csl Reg.	0	0	0	0	0	0	0
Csl I.	0	0	0	0	0	0	0
Dlsl	0	0	0	0	0	0	0
Dlsl Reg.	0	0	0	0	0	0	0
Dlsl I.	0	0	0	0	0	0	0
Dlsr	0	0	0	0	0	0	0
Dlsr Reg.	0	0	0	0	0	0	0
Dlsr I.	0	0	0	0	0	0	0
Lsl	35000(4.29)	11(0.74)	3167(8.94)	1361017(11.46)	0	0	1399195
Lsl Reg.	0	0	0	0	0	0	0
Lsl I.	35000	11	3167	1361017	0	0	1399195
Lsr	0	0	0	0	0	0	0
Lsr Reg.	0	0	0	0	0	0	0
Lsr I.	0	0	0	0	0	0	0
Sbit	0	0	0	0	0	0	0
Tbit	0	0	0	0	0	0	0

LOGICAL INS.	140(0.02)	4(0.27)	1014(2.86)	3381(0.03)	0	0	4539 (0.03 %)
And	0	1(0.07)	1014(2.86)	0	0	0	1015
And Reg.	0	1(0.07)	1014	0	0	0	1015
And I.	0	0	0	0	0	0	0
Not	140(0.02)	2(0.13)	0	0	0	0	142
Or	0	1(0.07)	0	3381	0	0	3382
Xor	0	0	0	0	0	0	0
TEST INSTRUCTION	0	4(0.27)	2267(6.40)	3381	0	0	5652 (0.03 %)
TestGT	0	2(0.13)	224	0	0	0	226
TestLT	0	0	790	0	0	0	790
TestEQ	0	1	0	3381	0	0	3382
TestIGT	0	0	209	0	0	0	209
TestILT	0	0	30	0	0	0	30
TestIEQ	0	0	0	0	0	0	0
TestLE	0	1	0	0	0	0	1
TestGE	0	0	0	0	0	0	0
TestNE	0	0	0	0	0	0	0
TestILE	0	0	790	0	0	0	790
TestIGE	0	0	224	0	0	0	224
TestINE	0	0	0	0	0	0	0
DATA MOVEMENT	195672(23.99)	89(5.98)	8501(24.00)	1695530(14.28)	344977(10.01)	43389(19.30)	288158 (13.95 %)
Move Reg.	176162	51	8051	1575979	344468	20188	2124899
Move I.	19510	38	450	119551	509	23201	163259
COMPARISON INST.	0	0	1223(3.45)	0	0	0	1223
Dcomp	0	0	0	0	0	0	0
Drcomp	0	0	0	0	0	0	0
Lcomp	0	0	0	0	0	0	0
Rcomp	0	0	1223	0	0	0	1223
SIGN EXTEND	0	0	0	0	0	0	0
Seb	0	0	0	0	0	0	0
Seh	0	0	0	0	0	0	0

Table 3.5.4. Simulation Result of Dynamic Instruction Usage (Continued)

BRANCH & CALL	60861(7.31)	98(5.82)	3916(10.93)	3002166(25.20)	517208(14.30)	56485(25.12)	3640734 (21.91 %)
BRANCH	39568(4.75)	66(3.92)	3518(9.82)	2967490(24.91)	344971(9.54)	56476(25.12)	3412089 (20.53 %)
BrGTS	0	2	0	0	0	16898	16900
BrGTL	0	0	0	0	0	0	0
BrEQS	0	3	0	0	0	0	3
BrEQL	0	0	0	0	0	0	0
BrIGTS	0	2	0	0	0	0	2
BrIGTL	0	0	0	0	0	0	0
BrILTS	0	0	0	0	0	0	0
BrILTL	0	0	0	0	0	0	0
BrIEQS	0	2	0	80871	0	0	80873
BrIEQL	0	0	0	0	0	0	0
BrLES	10	5	464	38187	1	3	38670
BrLEL	0	0	0	0	0	0	0
BrNES	22212	33	363	1386869	1	16383	1425861
BrNEL	0	0	0	0	0	0	0
BrS	6015	16	1133	24562	86368	15000	133094
BrL	1	1	1	1	1	1	6
BrILES	3450	0	0	0	0	0	3450
BrILEL	0	0	0	0	0	0	0
BrIGES	4430	0	61	0	0	0	4491
BrIGEL	0	0	0	0	0	0	0
BrINES	3450	2	1496	1437000	258600	8191	1708739
BrINEL	0	0	0	0	0	0	0
CALL	21293(2.56)	32(1.90)	398(1.11)	34676(0.29)	172237(4.76)	9	228645 (1.38 %)
CallS	15300	10	394	34672	172233	0	222609
CallL	5993	21	4	4	4	9	6035
Callr	0	1	0	0	0	0	1
LOOP	2240(0.27)	178(10.58)	0	0	0	5	2423 (0.01 %)
LoopS	2240	178	0	0	0	5	2423
LoopL	0	0	0	0	0	0	0
RET	15301(1.84)	16(0.95)	395(1.10)	34673(0.29)	172234(4.76)	1	222620 (1.34 %)

Table 3.5.4. Simulation Result of Dynamic Instruction Usage (Continued)

Real Arithmetic instructions are sorted as Single-Precision and Double-Precision.

Interesting statistics regarding to the distribution of the excuted instructions between Register-to-Register formatted instructions and Memory-reference formatted instructions are given in Table 3.5.5. Any instruction executed by the simulator should be either Register-to-Register format or Memory-reference format. Distributions between short displacement and long displacement memory reference instructions are also shown in this table.

Table 3.5.6 contains statistics regarding to the Memory access instructions between Direct and Indexed addressing.

Experiment results from Table 3.5.7 tells about Branch instructions, in particular the distributions between Conditional branch and Unconditional Branch instructions.

Memory reference instructions can access Code Space or Data Space of the main memory and distributions between Code Space reference instructions and Data Space reference instructions are given in Table 3.5.8.

Table 3.5.9 shows the average length of the loop in units of byte, and table 3.5.10. shows the average length of the offset in branch instructions in bytes. Since these data have no meaning in the static mode, data were obtained only in dynamic mode.

INSTRUCTION FORMAT	REG. TO REG.		MEMORY REFERENCE			
			SHORT DISPLACEMENT		LONG DISPLACEMENT	
	STATIC	DYNAMIC	STATIC	DYNAMIC	STATIC	DYNAMIC
WHETSTONE	318 (41.95%)	394579 (48.38%)	387 (51.06%)	399068 (48.93%)	53 (6.99%)	21962 (2.69%)
DHRYSTONE	178 (27.77%)	481 (32.30%)	420 (65.52%)	933 (62.66%)	43 (6.71%)	75 (5.04%)
QSORT	130 (36.52%)	19443 (54.88%)	213 (59.83%)	15582 (43.98%)	13 (3.65%)	401 (1.13%)
PUZZLE	460 (41.82%)	6163817 (51.90%)	621 (56.45%)	5677769 (47.81%)	19 (1.73%)	34679 (0.29%)
ACKERMAN	19 (25.33%)	775310 (22.51%)	48 (64.00%)	2497144 (72.49%)	8 (10.67%)	172240 (5.00%)
SIEVE	34 (26.98%)	82368 (36.64%)	77 (61.11%)	142414 (63.35%)	15 (11.90%)	19 (0.01%)
AVG. of %	33.39 %	41.10 %	59.66 %	56.54 %	6.95 %	2.36 %

Table 3.5.5. Statistics of Register-to-Register Formatted Instructions versus Memory Reference Instructions

MEMORY ACCESS	DIRECT MEMORY ACCESS		INDEXED MEMORY ACCESS	
	STATIC	DYNAMIC	STATIC	DYNAMIC
WHETSTONE	309 (70.23%)	212983 (50.59%)	131 (29.78%)	208047 (49.41%)
DHRYSTONE	200 (43.20%)	387 (38.39%)	263 (56.80%)	621 (61.61%)
QSORT	59 (26.11%)	3992 (24.98%)	167 (73.89%)	11991 (75.02%)
PUZZLE	485 (75.78%)	3195985 (55.95%)	155 (24.22%)	2516463 (44.05%)
ACKERMAN	25 (44.64%)	517218 (19.38%)	31 (55.36%)	2152166 (80.62%)
SIEVE	63 (68.48%)	111018 (77.94%)	29 (31.52%)	31415 (22.06%)
AVG. of %	54.74 %	44.54 %	45.26 %	55.46 %

Table 3.5.6 Statistics of Memory Access Instructions

TYPE OF BRANCH	CONDITIONAL BRANCH		UNCONDITIONAL BRANCH	
	STATIC	DYNAMIC	STATIC	DYNAMIC
WHETSTONE	24 (82.76%)	33552 (84.80%)	5 (17.24%)	6016 (15.20%)
DHRYSTONE	24 (48.00%)	49 (74.24%)	26 (52.00%)	17 (25.76%)
QSORT	17 (70.83%)	2384 (67.77%)	7 (29.12%)	1134 (32.23%)
PUZZLE	112 (94.12%)	2942927 (99.17%)	7 (5.88%)	24563 (0.83%)
ACKERMAN	4 (57.14%)	258602 (74.96%)	3 (42.86%)	86369 (25.04%)
SIEVE	8 (72.73%)	41475 (73.44%)	3 (27.27%)	15001 (26.56%)
AVG. of %	70.94 %	79.06 %	29.06 %	20.94 %

Table 3.5.7. Statistics of Branch Instructions

MEMORY REFERENCE	CODE SPACE REFERENCE		DATA SPACE REFERENCE	
	STATIC	DYNAMIC	STATIC	DYNAMIC
WHETSTONE	0 (0.00%)	0 (0.00%)	440 (100.00%)	421030 (100.00%)
DHRYSTONE	7 (1.51%)	122 (12.10%)	456 (98.49%)	886 (87.90%)
QSORT	0 (0.00%)	0 (0.00%)	226 (100.00%)	15983 (100.00%)
PUZZLE	0 (0.00%)	0 (0.00%)	640 (100.00%)	5712448 (100.00%)
ACKERMAN	0 (0.00%)	0 (0.00%)	56 (100.00%)	2669384 (100.00%)
SIEVE	1 (1.08%)	5 (0.00%)	91 (98.92%)	142428 (100.00%)
AVG. of %	0.43 %	2.02 %	99.57 %	97.98 %

Table 3.5.8. Statistics of Code Memory Access versus Data Memory Access

	LENGTH IN BYTES
WHETSTONE	12.00
DHRYSTONE	12.07
QUICKSORT	NO LOOP
PUZZLE	NO LOOP
ACKERMAN	NO LOOP
SIEVE	10.00
Average of Length	5.68

Table 3.5.9. Simulation Result of Average Length of Loop

	LENGTH IN BYTES
WHETSTONE	34.51
DHRYSTONE	63.67
QUICKSORT	70.09
PUZZLE	39.99
ACKERMAN	35.98
SIEVE	31.22
Average of Length	45.91

Table 3.5.10. Simulation Result of Average Length of Branch Offset

3.6. INTERPRETATION OF MEASUREMENTS

EXECUTION TIME

Table 3.6.1. shows the combined data of execution time, total number of instructions executed and resulting MIPS (Million Instructions Per Second).

From this table, it can be seen that the Ridge 32 obtains the highest MIPS number on the Puzzle benchmark program among those six benchmarks (3.22 MIPS). On the other hand, the Ridge 32 shows its lowest MIPS number on Whetstone benchmarks (1.17 MIPS). As average of these six benchmarks, the Ridge 32 performed at 2.87 MIPS.

These results can be explained by examining the simulation results of dynamic instruction distribution (Table 3.5.4.) and instruction execution time (Appendix E). High MIPS in the Puzzle benchmark can be understood from the fact that the majority of all instructions executed are single cycle (125 nanosecond) instructions such as integer addition. On the other hand, the Puzzle benchmark has smallest number of memory reference instructions among the six benchmarks. In the case of the Whetstone benchmark program, integer multiply instruction and real arithmetic instructions seem to contribute to slow down the

	Execution Time (Millisecond)	Total Instructions	MIPS
Whetstone	699.00	815609	1.17
Dhrystone	0.55	1489	2.71
Quicksort	13.00	35426	2.73
Puzzle	3689.00	11876265	3.22
Ackerman	1242.00	3444694	2.77
Sieve	76.90	224801	2.92
Total	5720.45	16398284	2.87

Table 3.6.1. Benchmark Execution Time and MIPS Results

MIPS number (These instructions are particularly complex and take at least 10 times longer to execute than simple instructions).

The results obtained in this experiment have discrepancies with the data given by Ridge Corporation [Basa83]. In the case of the Puzzle benchmark, there are significant differences between this result (3689 msecs) and the measurement performed at Ridge Corporation (2200 msecs)[Basa83]. Our measurement is 1.67 times longer than the value from the Ridge Corporation. In the case of the Whetstone benchmark, they claim 1.5 MIPS whereas 1.17 MIPS was obtained in this measurement and this number is almost same as the VAX 11/780 with FPA (Floating Point Accelerator) which performs at 1.168 MIPS. Since it is believed that there is no difference in hardware, these discrepancies in execution time measurements might be due to the use of a different compiler.

In the case of the Dhrystone benchmark, execution time in VAX 11/780 under Berkeley UNIX was measured as 710 microseconds (Pascal, without run time check). Execution times of Dhrystone for other computers can be found in [Weic85].

Table 3.6.2. shows the execution times of Sieve Puzzle and Ackerman benchmarks on several other computers.

References [Nati83][Gilb81] [Wadl84][Webs86] present more measurement results for microcomputers.

MACHINE	LANGUAGE	WORD SIZE	EXECUTION TIME (Millisecond)		
			SIEVE	PUZZLE	ACKERMAN
VAX 11/780	Pascal(UNIX)	32	220	11900	7800
	Pascal(VMS)	32	259	11530	9850
68000(8MHz)	Pascal	32	960	32520	12320
(16MHz)	Pascal	32	246	9200	3080
80286(10MHz)	Pascal	16	135	7311	1774
HP 32-Bit CPU(18MHz)	Pascal	32	NA	7450	2590
NS 16032 (7MHz)	Pascal	32	NA	24000	9900
Ridge 32	Pascal(ROS)	32	77	3689	1242

Table 3.6.2. Execution Times of Sieve, Puzzle and Ackerman benchmarks on Several Computers

COMPARISON BETWEEN STATIC AND DYNAMIC MODES

Comparison between the numbers of instructions executed in dynamic and in static modes tells the difference between dynamic trace and static trace of the benchmark programs by the simulator. In dynamic mode, the simulator follows all Branch & Call activities, while it ignores those instructions and follows only consecutive instructions in static mode. Over the six benchmark programs used in this study, Branch & Call instructions make up 21.91 % of the total executed instructions in dynamic mode, and only 11.31 % in static mode.

BIASED UTILIZATION OVER SMALL INSTRUCTION GROUP

There are 174 instructions in the Ridge 32 Opcode Map which is published by Ridge Corporation. It is found that only 63 instructions out of these 174 instructions are used in the six benchmark programs used in this experiment. Among them, only 37 instructions are used more than 1 % in any of those six benchmarks and they constitute 98.54 % of the cumulative total instructions executed over the six benchmark programs.

In the case of Ackerman, only 11 instructions constitute 99.98 % of the total executed instructions. This means that only 6.32 % of the available instructions in the Ridge 32 were used to execute the Ackerman benchmark. From this result, it is observed that only a small portion (36.21 %) of the already small instruction set are used in all these benchmark programs and some benchmarks consist of even less instructions. A similar result was observed in an early study of the RISC I [Heat84].

Table 3.6.3. shows the detailed statistics of instruction usage over the six benchmarks. In the average over six benchmarks, 15.33 instructions were used and they constitute 94.85 % of the total executed instructions.

IMPACT OF MEMORY REFERENCE INSTRUCTIONS ON PERFORMANCE

From Table 3.5.4., we can analyze memory traffic in the benchmark programs. In these six benchmark programs, Ackerman's function shows the most intensive memory traffic(62.48 % of the total instructions executed) and Puzzle shows the least traffic(22.82 % of the total executed instructions). This result can be explained because of the fact that the Ackerman's function benchmark requires a large number of recursive calls which

need memory access to save and restore the return information, whereas Puzzle needs more integer arithmetic and branch instructions. In the case of Dhrystone, it was observed that 49.23 % of the total instructions executed are memory reference instructions and it is second in this respect only to Ackerman's function.

In a recent paper [Luke 86], dynamic frequency of instructions was measured over variety of benchmark programs which were run on the Amdahl V6 computer to gather data to aid select the instruction set for a possible reduced instruction set computer (Called by them Precision Architecture). They found out that in the case of Pascal language, 54 % of the total instructions executed over the benchmarks are Load Store instructions and 3.8 % of total instructions are Storage-Storage Move instructions. Since both instructions can be considered as memory reference instructions, 57.8 % of the total instructions executed over various benchmark programs are memory reference instructions.

It is interesting to compare this result with our results with the Ridge 32 simulator. The measurement with the Ridge 32 simulator shows that, in the average of the six benchmarks, 41.78 % of the total instructions are memory reference instructions and most of them (56.54 % of the memory reference instructions) are

short displacement memory reference instructions. Those two results show the importance of memory reference instructions. If we consider the fact that execution time of the memory reference instructions can be up to five times longer than register format instructions in the Ridge 32, it is clear that much more than 41.78 % of the total execution times should be consumed by the memory reference instructions. This result indicates the importance of reducing the memory traffic with approaches such as overlapped register sets and/or optimized compilers. Since current results are based on the unoptimized version of the Pascal compiler, the optimized version of the Ridge 32 Pascal compiler should increase the performance significantly. This should also explain the difference between our measurement and those reported by Ridge Corporation.

INDEXED MEMORY ACCESS VERSUS DIRECT MEMORY ACCESS

Table 3.5.5. shows statistics regarding the way of access to memory of the instruction Store and Load. Indexed memory access use a register (Ry) where the content of register Ry is added to the displacement field of the instruction to get the effective address. In the average, 55.46 % of the total Store

and Load instructions use the indexed addressing and 44.54 % of them use the direct addressing.

BRANCH INSTRUCTIONS

In Table 3.5.6., statistics regarding to the branch instructions are given. Among the six benchmark programs, 20.81 % of the total instructions are branch instructions. Of these, 79.06 % are conditional branch instructions and the rest of them (20.94 %) are unconditional. The Ridge 32 uses Branch Prediction logic which loads the instruction in the pipe which is the most likely result of the branch. Branch instructions contain a static prediction bit in the instruction displacement field that can be set by the compiler and Branch Prediction logic keeps the pipeline full and makes branch instructions full.

For comparison purposes, in the study of dynamic frequency of instruction occurrences over a variety of benchmark programs reported in [Luke86], branch instructions were between 18 % and 24.2 % of the total instructions depending on the language (Fortran,Pascal and COBOL). In that same study, they found that

this percentage was 19.9 % for the Hewlett-Packard HP 3000. A study of the Motorola 68020 [MAcG85] determined its dynamic frequency for Branch/Jump instructions to be 23.95 %. All these values correspond closely to the ones determined in this study where 20.81 % was obtained.

CHAPTER IV

CONCLUSION

This work used six benchmark programs to evaluate the Ridge 32 by direct measurement and by simulation.

It was found that the Ridge 32 simulator is very useful to measure the parameters needed to evaluate the performance of the Ridge 32 and it is the only way to measure static and dynamic instruction utilization. Because of the flexibility of the simulator, it can be used to measure other parameters with minimum effort.

It was also found that only a small portion (36.21 %) of the already small instruction set was used in these benchmark programs. Among the small set of instructions used in these benchmarks, 41.78 % of them are memory reference instructions, which means that any way to reduce the memory traffic can significantly improve the performance of a computer such as the Ridge 32. To reduce the memory traffic, an optimized compiler which enables high reusability of register contents can be considered. A set of registers with overlapped windows is another approach to reduce memory traffic.

Due to several limitations at the time of this experiment, the following interesting aspects were simplified or neglected and could be performed in a future study :

* If the execution time of every instruction is known, the total execution time of a benchmark program can be easily measured by this simulator.

* Implementation of pipeline structure and branch prediction logic in the simulator can provide more data on these logic.

* Implementation of virtual memory structure can be added to the simulator.

* Performance evaluation under a multiuser environment could be of interest. One should not here that the Ridge 32 is designed to be used with a small number of users (five or less).

* Performance evaluation of the effect of the multiple overlapped register windows on Ridge 32.

* Performance evaluation of the high level language support factor of Ridge 32

* Analyze the effect of the instruction cache in the bus traffic.

* Comparison of the results of this work with results obtained with an optimized Pascal compiler.

One should also notice that Ridge Corporation has a new, more powerful version of its system 32. Our benchmark timings certainly do not apply to this new system. However, the results of the simulation are still applicable to the enhanced system since its architecture is the same as the Ridge 32.

REFERENCE

[Barb77] M.R. Barbacci and D.P. Siewiorek
"Evaluation of the CFA Test Programs via Formal Computer Descriptions,"
Computer, October 1977, pp.36-43.

[Basa82] E. Basart, D. Folger and B. Shellooe
"Pipelining and new OS boost mini to 8 MIPS,"
Mini-micro Systems, September 1982, pp. 258-272.

[Basa83] E. Basart, and D. Folger
"RIDGE 32 Architecture -- A RISC variation,"
Proc. Intl. Conf. on Comp. Design, 1983, pp. 315-318.

[Basa85] E. Basart
"RISC Design Streamlines High-Power CPUs,"
Computer Design, July 1985, pp. 119-122.

[Colw85] R. P. Colwell, C. Y. Hitchcock, E. D. Jensen, H. M. Sprunt, and C. P. Kollar
"Computers,Complexity, and Controversy,"
Computer, September 1985, pp. 8-19.

[Ditz80] D. R. Ditzel, D. A. Patterson
"Retrospective on High-Level Language Computer Architecture," Proc. 7th Ann. Symp. Computer Architecture, May 1980, pp. 97-104.

[Fern85] E. B. Fernandez
"An Evaluation of a RISC System,"
Proc. MIDCON, 1985.

[Full77] S.H. Fuller and W.E. Burr
"Measurement and Evaluation of Alternative Computer Architectures," Computer, October 1977, pp.24-35.

[Gilb81] J. Gilbreath
"A High-Level Language Benchmark,"
BYTE, Sep. 1981, pp. 180-198.

[Grap81] R.G. Grappel, J.E. Hemmenway
"A Tale of Four Microprocessors: Benchmarks Quantify Performance," Electronic Design News, Apr.1981, pp. 179-265.

[Heat84] J.L. Heath
"Re-evaluation of the RISC I,"
Computer Architecture News, Mar. 1984, pp. 3-10.

[Henn82] J. Hennessy et al.
"Hardware/Software Tradeoffs for Increased Performance," Proc. Symp. Architectural Support for Programming Languages and Operating Systems 1982, pp. 2-11.

[Henn84] J.L. Hennessy
"VLSI Processor Architecture," IEEE Transactions on Computers, Dec. 1984, pp. 1221-1246.

[Laru82] J. R. Larus
"A Comparison of Microcode, Assembly Code, and High-Level Languages on the VAX-11 and RISC I,"
Comp. Arch. News, 10, September 1982, pp. 10-15.

[Luke86] J. A. Lukes
"HP Precision Architecture Performance Analysis,"
Hewlett-Packard Journal, Aug. 1986.

[Lund77] A. Lunde
"Empirical Evaluation of Some Features of Instruction Set Processors,"
Comm. of the ACM,20, 3, Mar.1977, pp. 143-154.

[MacG85] D. MacGregor and J. Rubinstein
"A Performance Analysis of MC68020-based Systems,"
IEEE Micro, 5, 6, Dec. 1985, pp 50-70.

[Nati83] National Semiconductor Corp.
"The NS 16000 Benchmakrs," Aug. 1983.

[Patt82a] D. A. Patterson, R. S. Piepho
"Assessing RISCs in High-Level Language Support,"
IEEE Micro, Nov. 1982, pp.9-19.

[Patt82b] D. A. Patterson, C.H. Sequin
"A VLSI RISC," Computer, Sept. 1982, pp. 8-21.

[Patt84] D. A. Patterson
"RISC Watch," Computer Architecture News,
Mar. 1984, pp. 11-19.

[Patt85a] D. A. Patterson, J. Hennessy
Response to "Computers, Complexity, and Controversy,"
Computer, 142-143 (Nov. 1985)

[Patt85b] D. A. Patterson
"Reduced Instruction Set Computers,"
Comm. of the ACM, 8-21 (Jan. 1985)

[Radi82] G. Radin
"The 801 Minicomputer," Proc. Symp. Architectural
Support for Programming Languages and Operating Systems
39-47, (1982)

[Rals85] R. Ralston, G. Spicker, B. Furht
"Comparison of Three Advanced RISC/CISC Computer
Architecture," Proc. MIDCON, 1985.

[Wadl84] T. Wadlow
"Turbo Pascal,"
BYTE, July 1984, 267-277.

[Wall85] P. Wallich
"Toward simpler, faster computers,"
IEEE Spectrum, 38-45 (Aug. 1985)

[Webs86] B. Webster
"Benchmarking,"
BYTE, Jan. 1986.

[Weic84] R. P. Weicker
"Dhrystone : A Synthetic Systems Programming Benchmark,"
Communications of the ACM, Oct. 1984, 1013-1030.

[Weic85] R. P. Weicker
"Execution Times for the "Drystone" Benchmark Program,"
Mar. 1985.

APPENDIX A

BENCHMARK PROGRAM LISTINGS

Jul 17 18:54 1986 whet.p Page 1

```
program whet(output);

type four = array [1..4] of real;

var
   etime : integer;
   i,n1,n: integer;
   x1,x2,x3,x4,x,y,z,t,t3 : real;
   j,k,l,m : integer;
   e1 : four;
   n2,n3,n4,n5,n6,n7,n8,n9 : integer;
   n10,n11,n12             : integer;
   t1,t2   : real;
   loopcounter             : integer;(* for 100 run *)
{ import procedure init_sicoln_;}

procedure ldctr;
begin
end;

function rdctr : integer;
begin
rdctr := 0;
end;

procedure pa ( e : four );

var j : integer;

begin
 j := 0;
while  j < 6 do begin
e [1] := (e[1] + e[2] + e[3] - e[4]) * t;
e [2] := (e[1] + e[2] - e[3] + e[4]) * t;
e [3] := (e[1] - e[2] + e[3] + e[4]) * t;
e [4] := (-e[1] + e[2] + e[3] +e[4]) * t;

j := j + 1;
end;
end;{ pa }

procedure p0;
begin
e1[j] := e1[k];
e1[k] := e1[l];
e1[l] := e1[j];
end;{ p0}

procedure p3( var x,y,z : real );
begin
x := t * ( x + y);
y := t * ( x + y);
z := (y + z )/2;
end;

procedure pout ( n1,n2,n3 : integer; x1,x2,x3,x4 : real);
```

Figure A-1 Whetstone Benchmark Program

```
Jul 17 18:54 1986  whet.p Page 2


begin
{writeln (n1,n2,n3);
writeln (x1);
writeln (x2);
writeln (x3);
writeln (x4);}
end;



begin {main}

{init_sicoln_}
etime := 0;
t := 499975/(100.0 * 100.0 * 100.0 );
t1:= 50025/ (100.0 * 100.0 * 10.0);
t2 := 2.0;
i  := 10;
n1 := 1;
n2 := 12 * i;
n3 := 14 * i;   n4 := 345 * i;
n5 := 1      ;   n6 := 210 * i;
n7 := 32 * i;   n8 := 899 * i;
n9 := 616 * i;  n10 := 1;
n11:= 93 * i ;  n12 := 1;

x1 := 1.0;
x2 := -1.0;
x3 := -1.0;
x4 := -1.0;

for i := 1 to n1 do
   begin
   x1 := (x1 + x2 + x3 - x4) * t;
   x2 := (x1 + x2 - x3 + x4) * t;
   x3 := (x1 - x2 + x3 + x4) * t;
   x4 := (-x1 + x2 + x3 +x4) * t;
   end;

pout(n1,n1,n1,x1,x2,x3,x4);

e1 [1] := 1.0;
e1 [2] :=-1.0;
e1 [3] :=-1.0;
e1 [4] :=-1.0;

for i := 1 to n2 do begin
e1[1] := ( e1[1] + e1[2] + e1[3] - e1[4]) * t;
e1[2] := ( e1[4] + e1[1] + e1 [2]- e1 [3])* t;
e1[3] := ( e1[3] + e1 [4] + e1[1] - e1[2])* t;
e1[4] := ( e1[2] + e1[3] + e1[4] - e1[1] )* t;
end;

pout (n2,n3,n2,e1[1],e1[2],e1[3],e1[4]);

for i := 1 to n3 do pa(e1);
```

Figure A-1 Whetstone Benchmark Program

(Continued)

```
pout(n3,n2,n2,e1[1],e1[2],e1[3],e1[4]);

j := 1;

for i := 1 to n4 do begin
if j =1 then j := 2 else j := 3;
if j >2 then j := 0 else j := 1;
if j <1 then j := 1 else j := 0;
end;

pout (n4,j,j,x1,x2,x3,x4);
j := 1; k :=2; l := 3;

for i := 1 to n6 do begin
   j := j * (k - j) * (l - k);
   k := l * k - (l - j) * k;
   l := (l - k) * (k + j);
   e1 [l-1] := i + k + l;
   e1 [k-1] := j * k * l;
end;

pout (n6,j,k,e1[1],e1[2],e1[3],e1[4]);

x := 0.5;
y := 0.5;

for i := 1 to n7 do begin
x := t * arctan(t2 * sin(x) * cos(x)/(cos(x+y) + cos (x-y)-1.0));
y := t * arctan(t2 * sin(y) * cos(y)/(cos(x+y) + cos (x-y)-1.0));
end;
pout (n7,j,k,x,x,y,y);

x := 1.0; y := 1.0; z := 1.0;

for i := 1 to n8 do
p3 (x,y,z);

pout (n8,j,k,x,y,z,z);

j :=1; k := 2; l :=3;
e1[1] := 1.0;
e1 [2] := 2.0; e1[3] :=3.0;
for i := 1 to n9 do p0;
pout (n9,j,k,e1[1],e1[2],e1[3],e1[4]);
j := 2; k := 3;
for i := 1 to n10 do
begin
j := j + k;
k := j + k;
j := k - j;
k := k - j - j;
end;

pout(n10,j,k,x1,x2,x3,x4);
x := 0.75;
```

Figure A-1 Whetstone Benchmark Program

(Continued)

```
for i := 1 to n11 do
x := sqrt(exp(ln(x)/t1));
pout (n11,j,k,x,x,x,x);
end. {main}
```

Figure A-1 Whetstone Benchmark Program

(Continued)

```
program Dhrystone (input,output);
(*******************************)

const (*for measurement*)
Numberofexecutions = 10000;
Numberofmeasurements = 10;
Largerealnumber      = 1000000.0;
Microsecondsperclock = 1000;

type
Enumeration = (ident1,ident2,ident3,ident4,ident5);
onetothirty = 1..30;
onetofifty  = 1..50;
capitalletter = 'A'..'Z';

string30 = packed array [1..30] of char;

array1dimintegr = array [onetofifty] of integer;
array2diminteger = array [onetofifty,onetofifty] of integer;

recordpointer = ^recordtype;

recordtype =
   record
      pointercomp : recordpointer;
      case discr : enumeration of
         ident1 :
            (enumcomp : enumeration;
             intcomp  : onetofifty;
             stringcomp : string30  );
         ident2 :
            (enumcomp2 : enumeration;
             stringcomp2 : string30 );
         ident3,ident4,ident5:
            (charcomp1,charcomp2: char );
   end; (* record *)

var
executionindex : 1..numberofexecutions;
measurementindex : 1..numberofmeasurements;
beginclock,endclock : integer;
sumclocks,emptyloopclocks,
timeperexecution,
sumtime,mintime     : real;

(* end of variables for measurement *)

pointerglob,
nextpointerglob : recordpointer;
intglob         : integer;

boolglob        : boolean;
charglob1,
charglob2       : char;
arrayglob1      : array1diminteger;
arrayglob2      : array2diminteger;
```

Figure A-2 Dhrystone Benchmark Program

```
intglob1,
intglob2,
intglob3        : onetofifty;
charindex       : char;
Enumglob        : enumeration;
stringglob1,
stringglob2     : string30;

procedure proc1 ( pointerparval : recordpointer ); forward;
procedure proc2 ( var intparref : onetofifty    ); forward;
procedure proc3 ( var pointerparref : recordpointer ); forward;
procedure proc4; forward;
procedure proc5; forward;
procedure proc6 (    enumparval: enumeration;
                 var enumparref: enumeration); forward;
procedure proc7 (    intpar1val,
                     intpar2val : onetofifty;
                 var intparref  : onetofifty ); forward;
procedure proc8 (var arraypar1ref : array1diminteger;
                  var arraypar2ref : array2diminteger;
                      intpar1val,
                      intpar2val   : integer ); forward;
function func1 ( charpar1val,
                  charpar2val : capitalletter):enumeration;forward;
function func2 (var stringpar1ref,
                     stringpar2ref : string30):boolean; forward;
function func3 (     enumparval : enumeration):boolean; forward;


(********************************************)

procedure proc1;(* (pointerparval: recordpointer) *)
(* executed onece *)
begin

with pointerparval^.pointercomp^ do
begin
   pointerparval^.pointercomp^ := pointerglob^;
   pointerparval^.intcomp := 5;
   intcomp := pointerparval^.intcomp;
   pointercomp := pointerparval^.pointercomp;
   proc3 ( pointercomp );
   if discr = ident1
   then
      begin
      intcomp := 6;
      proc6 (pointerparval^.enumcomp,enumcomp );
      pointercomp := pointerglob^.pointercomp;
      proc7 (intcomp,10,intcomp);
      end
   else pointerparval^ := pointerparval^.pointercomp^;
end;
end;

(**********************************************)
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

Jul 17 19:40 1986 dh.p Page 3

```
procedure proc2; (* (var intparref : onetofifty) *)
(*executed once *)
var
intloc : onetofifty;
enumloc: enumeration;

begin

intloc := intparref + 10;
repeat
 if charglob1 = 'A'
 then
 begin
   intloc := intloc - 1;
   intparref := intloc - intglob;
   enumloc := ident1;
 end
until enumloc = ident1;
end;

(*********************************************)

procedure proc3;
begin

if pointerglob <> nil
then pointerparref := pointerglob^.pointercomp
else intglob := 100;
proc7 (10,intglob,pointerglob^.intcomp);

end;

(*********************************************)

procedure proc4;
var
boolloc : boolean;
begin
boolloc := charglob1 = 'A';
boolloc := boolloc or boolglob;
charglob2 := 'B';
end;

(*********************************************)

procedure proc5;
begin
charglob1 := 'A';
boolglob  := false;
end;

(*********************************************)

procedure proc6;
begin
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

```
enumparref := enumparval;
if not func3 (enumparval)
then enumparref := ident4;
case enumparval of
ident1 : enumparref := ident1;
ident2 : if intglob > 100
           then enumparref := ident1
           else enumparref := ident4;
ident3 : enumparref := ident2;
ident4:;
ident5: enumparref := ident3;
end;
end;


(****************************************************)

procedure proc7;
var
intloc: onetofifty;

begin

intloc := intpar1val + 2;
intparref := intpar2val + intloc;

end;

(****************************************************)

procedure proc8;

var
intindex,intloc : onetofifty;

begin

intloc := intpar1val + 5;
arraypar1ref [intloc] := intpar2val;
arraypar1ref [intloc + 1] := arraypar1ref [intloc];
arraypar1ref [intloc + 30] := intloc;
for intindex := intloc to intloc+1 do
   arraypar2ref [intloc,intindex] := intloc;
arraypar2ref [intloc,intloc-1] := arraypar2ref [intloc,intloc-1] +1;
arraypar2ref [intloc+20,intloc] := arraypar1ref [intloc];
intglob := 5;
end;

(********************************************************)

function func1;
var charloc1,charloc2 : capitalletter;

begin
charloc1 := charpar1val;
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

```
Jul 17 19:40 1986  dh.p Page 5


charloc2 := charloc1;
if charloc2 <> charpar2val
   then func1 := ident1
   else func1 := ident2;
end;

(*****************************************************)

function func2;
var
intloc : onetothirty;
charloc : capitalletter;
begin

intloc := 2;
while intloc <= 2 do
  if func1 (stringpar1ref[intloc],stringpar2ref[intloc + 1]) = ident1
   then begin
   charloc := 'A';
   intloc := intloc + 1;
   end;

if (charloc >= 'W') and (charloc < 'Z')
   then intloc := 7;
if charloc = 'X'
then func2 := true
else
begin
if stringpar1ref > stringpar2ref
then
begin
intloc := intloc + 7;
func2 := true
end
else func2 := false;
end;
end;

(*******************************************************)

function func3;
var
enumloc : enumeration;
begin

enumloc := enumparval;
if enumloc = ident3
then func3 := true;

end;


(*****************************************************************)
(*****************************************************************)
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

```
begin

new (nextpointerglob);
new (pointerglob);

pointerglob^.pointercomp := nextpointerglob;
pointerglob^.discr       := ident1;
pointerglob^.enumcomp    := ident3;
pointerglob^.intcomp     := 40;
pointerglob^.stringcomp  := 'DHRYSTONE PROGRAM, SOME STRING';

stringglob1 := 'DHRYSTONE PROGRAM, 1''ST STRING';

writeln;
writeln ('Dhrystone Benchmark (Mar 84), Version Pascal / 2');
writeln ('Times are CPU user time per execution, in     ');
writeln;

sumtime := 0.0;
mintime := largerealnumber;

{Beginclock := clock;}

writeln (' Start Timer !');

(*******************)
(* start Timer     *)
(*******************)

for  executionindex := 1 to NumberOfExecutions do
begin

proc5;
proc4;
intglob1 := 2;
intglob2 := 3;
stringglob2 := 'DHRYSTONE PROGRAM, 2''ND STRING';
enumglob := ident2;
boolglob := func2 (stringglob1,stringglob2);

while intglob1 < intglob2 do
begin
intglob3 := 5 * intglob1 - intglob2;
proc7 (intglob1,intglob2,intglob3);
intglob1 := intglob1 + 1;
end; {while}

proc8 (arrayglob1,arrayglob2,intglob1,intglob3);
proc1 (pointerglob);
for charindex := 'A' to charglob2 do
 if enumglob = func1 ( charindex,'C')
then  proc6 (ident1,enumglob);

intglob3 := intglob2 * intglob1;
intglob2 := intglob3 div intglob1;
intglob2 := 7 * (intglob3 - intglob2) - intglob1;
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

```
proc2 ( intglob1);

end;{for execution index}

{endclock := clock;}
(*******************)
(* stop timer      *)
(*******************)

writeln (' Stop Timer !');
end.
```

Figure A-2 Dhrystone Benchmark Program

(Continued)

```
Jul 17 19:51 1986  qsort.p Page 1


program qsort (input,output );
{This program is modified according to BYTE July 1984, pg.275}
const
        max = 100;
type
        standardarray = array [0..max] of real;
var
        numbers : standardarray;
        last : integer;
        loopcounter :integer;(* to measure the 100 run time *)

(**************************************************************)

procedure swap (var a,b :real);
var
        t : real;
begin

t := a;
a := b;
b := t;

end;

(**************************************************************)

procedure printarray (top : integer);
var
        i : integer;
begin

{writeln ('-MARK-');}
end;

procedure getarray (var top : integer );


var
k,
i          : integer;
(* typed constant is substituted by for loop *)
{worstcase[1:100] :=
(100.0,99.0,98.0,97.0,96.0,95.0,94.0,93.0,92.0,91.0,
  90.0,89.0,88.0,87.0,86.0,85.0,84.0,83.0,82.0,81.0,
  80.0,79.0,78.0,77.0,76.0,75.0,74.0,73.0,72.0,71.0,
  70.0,69.0,68.0,67.0,66.0,65.0,64.0,63.0,62.0,61.0,
  60.0,59.0,58.0,57.0,56.0,55.0,54.0,53.0,52.0,51.0,
  50.0,49.0,48.0,47.0,46.0,45.0,44.0,43.0,42.0,41.0,
  40.0,39.0,38.0,37.0,36.0,35.0,34.0,33.0,32.0,31.0,
  30.0,29.0,28.0,27.0,26.0,25.0,24.0,23.0,22.0,21.0,
  20.0,19.0,18.0,17.0,16.0,15.0,14.0,13.0,12.0,11.0,
  10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,
   0.0);}
begin

top := 100;
```

Figure A-3 Quicksort Benchmark Program

```
(* for loop do the func. of typed const in original program *)
k := 100;
for i := 0 to 100 do
begin
numbers[i] := k;
k := k - 1;
end;(*for*)

printarray (top);
end;

procedure bubblesort (start,top: integer;var subarray : standardarray);

var
        index : integer;
        switched : boolean; begin repeat
begin
switched := false;
for index := start to top - 1
do
   begin
   if subarray [index] > subarray [index + 1]
   then
        begin
        swap (subarray[index],subarray [index + 1]);
        switched := true;       end;
   end;
end;

until switched = false;
end;


procedure findmedian (start,top : integer; var subarray : standardarray);

var
middle : integer;
sorted : standardarray;
begin

middle := (start + top)div 2;
sorted[1] := subarray [start];
sorted[2] := subarray [top];
sorted[3] := subarray [middle];
bubblesort (1,3,sorted);
if sorted[2] = subarray[middle]
then
   swap(subarray[start],subarray[middle])
else
   if sorted[2] = subarray [top]
   then
        swap(subarray[start],subarray[top]);
end;
```

Figure A-3 Quicksort Benchmark Program

(Continued)

```
Jul 17 19:51 1986  qsort.p Page 3


procedure sortsection (start,top : integer);
var
        swapup : boolean;
        s,e,m : integer;
begin

if top - start < 6
then
   bubblesort (start,top,numbers)
else
   begin
   findmedian (start,top,numbers);
   swapup := true;

   s := start;
   e := top;
   m := start;
   while e > s do
   begin
   if swapup = true
   then
        begin
        while (numbers [e] >= numbers [m]) and ( e > m)
        do
           e := e - 1;
        if e > m
        then
           begin
           swap ( numbers [e], numbers [m]);
           m := e;
           end;
        swapup := false;
        end
   else
        begin
        while (numbers [s] <= numbers[m]) and ( s < m)
        do
           s := s + 1;
        if s <m
        then
           begin
           swap(numbers[s],numbers[m]);
           m := s;
           end;
        swapup := true;
        end;
   end;
sortsection (start,m-1);
sortsection (m + 1,top);
end;
end;

(********************************************************)

begin
```

Figure A-3 Quicksort Benchmark Program

(Continued)

```
getarray (last);
sortsection(0,last);
printarray(last);

end.{Main}
```

Figure A-3 Quicksort Benchmark Program

(Continued)

Jul 17 16:29 1986 puzzle.p Page 1

```
program xpuzzle (input,output);
const   xsize = 511;
        classmax = 3;
        typemax  = 12;
        d        = 8;

type    piececlass = 0..classmax;
        piecetype  = 0..typemax;
        position   = 0..xsize;

var     piececount :array [piececlass] of 0..13;
        class      :array [piecetype] of piececlass;
        piecemax   :array [piecetype] of position;
        puzzle     :array [position] of boolean;
        p          :array [piecetype,position] of boolean;
        m,n        :position;
        i,j,k      :0..13;
        kount      : integer;
{*********************************************************}

function fit (i:piecetype; j:position) : boolean;
label  1;
var    k:position;
begin

        fit :=false;
        for k := 0 to piecemax[i] do
           if p[i,k] then if puzzle[j+k] then goto 1;
        fit := true;
1 :
end;

function place (i :piecetype; j : position) : position;
label 1;
var    k : position;
begin
        for k := 0 to piecemax[i] do
                if p[i,k] then puzzle[j + k] := true;
        piececount [class[i]] := piececount [class[i]] - 1;
        for k := j to xsize do
                if not puzzle [k] then begin
                        place := k;
                        goto 1;
                end;
        {writeln ('puzzle filled');}
        place := 0;
1 :
end;

procedure remove(i : piecetype; j:position );
var k : position;
begin
        for k := 0 to piecemax[i] do
                if p[i,k] then puzzle[j +k] := false;
        piececount [class[i]] := piececount[class[i]] +1;
end;
```

Figure A-4 Puzzle Benchmark Program

Jul 17 16:29 1986 puzzle.p Page 2

```
function trial (j : position) : boolean;
label 1;
var i: piecetype;
    k: position;
begin
        for i := 0 to typemax do
                if piececount[class[i]] <> 0 then
                        if  fit (i,j) then begin
                                k := place (i,j);
                                if trial (k) or (k = 0) then begin
                                        {writeln ('piece',i+1,'at',k+1);}
                                        trial := true;
                                        goto 1;
                                end else remove (i,j);
                        end;
        trial := false;
1 :     kount := kount + 1;
end;
{*****************************************************************************}

begin

for m := 0 to size do puzzle[m] := true;
for i := 1 to 5 do for j := 1 to 5 do for k := 1 to 5 do
        puzzle[i+d*(j+d*k)] := false;
for i := 0 to typemax do for m := 0 to size do p[i,m] := false;
for i := 0 to 3 do for j := 0 to 1 do for k := 0 to.0 do
        p[0,i + d*(j + d*k)] := true;
class[0] := 0;
piecemax [0] := 3 + d*1 + d*d*0;
for i := 0 to 1 do for j := 0 to 0 do for k := 0 to 3 do
        p [1,i + d*(j + d* k)] := true;
class[1] := 0;
piecemax[1] := 1+d*0+d*d*3;
for i := 0 to 0 do for j := 0 to 3 do for k := 0 to 1 do
        p[2,i + d*(j + d*k)] := true;
class[2] := 0;
piecemax[2] := 0 + d*3+d*d*1;
for i := 0 to 1 do for j := 0 to 3 do for k := 0 to 0
do      p[3,i + d*(j + d*k)] := true;
class[3] := 0;
piecemax[3] := 1 +d*3 +d*d*0;
for i := 0 to 3 do for j := 0 to 0 do for k := 0 to 1 do
        p[4,i + d*(j + d*k)] := true;
class [4] := 0;
piecemax[4] := 3 + d*0 + d*d*1;
for i := 0 to 0 do for j := 0 to 1 do for k := 0 to 3 do
        p[5,i+d*(j + d*k)] := true;
class[5] := 0;
piecemax [5] :=0 + d*1 + d*d*3;
for i:= 0 to 2 do for j := 0 to 0 do for k := 0 to 0 do
        p[6,i + d*(j + d*k)] := true;
class [6] := 1;
piecemax [6] := 2 + d * 0 + d*d*0;
for i := 0 to 0 do for j := 0 to 2 do for k := 0 to 0 do
```

Figure A-4 Puzzle Benchmark Program

(Continued)

```
        p[7,i+d*(j + d*k)] := true;
class[7] := 1;
piecemax[7] := 0 + d*2 + d*d*0;
 for i := 0 to 0 do for j := 0 to 0 do for k := 0 to 2 do
        p[8,i + d*(j + d*k)] := true;
class[8] := 1;
piecemax [8] := 0 + d*0 + d*d*2;
for i := 0 to 1 do for j := 0 to 1 do for k := 0 to 0 do
        p[ 9,i + d*(j + d*K)] := true;
class [9] := 2;
piecemax [9] := 1 + d*1 + d*d*0;
for  i := 0 to 1 do for j := 0 to 0 do for k := 0 to 1 do;
        p[ 10,i + d*(j + d* k)] := true;
class [10] := 2;
piecemax [10] := 1 + d*0 + d*d*1;
for i := 0 to 0 do for j := 0 to 1 do for k := 0 to 1 do
        p [ 11,i + d*(j+d*k)] := true;
class [11] := 2;
piecemax [11] := 0 + d*1 + d*d*1;
for i := 0 to 1 do for j := 0 to 1 do for k := 0 to 1 do
        p [ 12,i + d*(j + d*k)] := true;
class [12] := 3;
piecemax [12] := 1 + d* 1 + d*d*1;
piececount[0] := 13;
piececount [1] := 3;
piececount [2] := 1;
piececount [3] := 1;
m := 1 + d*(1 +d *1);
kount := 0;
if fit (0,m) then n:= place(0,m) else writeln ('error1');
if trial(n) then {writeln('success in',kount,'trials')}
else writeln ('failure');

end.
```

Figure A-4 Puzzle Benchmark Program

(Continued)

Nov 25 15:07 1986 ackerman.p Page 1

```
program ackerman (input,output);
var i,r : integer;
function acker (m,n : integer) : integer;
begin
   if m=0 then
            acker := n +1
   else if n = 0 then
         acker := acker (m-1,1)
         else
          acker := acker (m-1,acker(m,n-1));
end;

begin
   r := acker(3,6);
end.
```

Figure A-5 Ackerman Benchmark Program

Jul 17 16:11 1986 sieve.p Page 1

```
program main(input,output);
{ Sieve of Erasthenes benchmark from Byte, Sept. '81, pg. 186}
const
   size = 8190;

var
   flags : array[0..size] of boolean;
   prime,k,count : integer;
   iter,i        : integer;

begin
   write ('100 iterrations: ');
   writeln;
   for iter := 1 to 100 do begin
      count := 0;
      for i := 0 to size do flags[i] := true;
      for i := 0 to size  do begin
         if (flags[i]) then begin
            prime := i+i+3;
            k     := i + prime;
            while (k <= size) do begin
               flags[k] := false;
               k := k + prime;
            end;
            count := count + 1;
          end;
       end;
    end;
    write ( count);
    write (' primes');
    writeln;

end.
```

Figure A-6 Sieve Benchmark Program

APPENDIX B

EXAMPLE OF USE OF THE RIDGE 32 SIMULATOR

Appendix B shows the typical scenario to simulate the target benchmark program "XX.p" (where "XX" means any file name) written in Pascal and the typical dialog between the user and the Ridge 32 simulator as it happens during the simulation of a benchmark program.

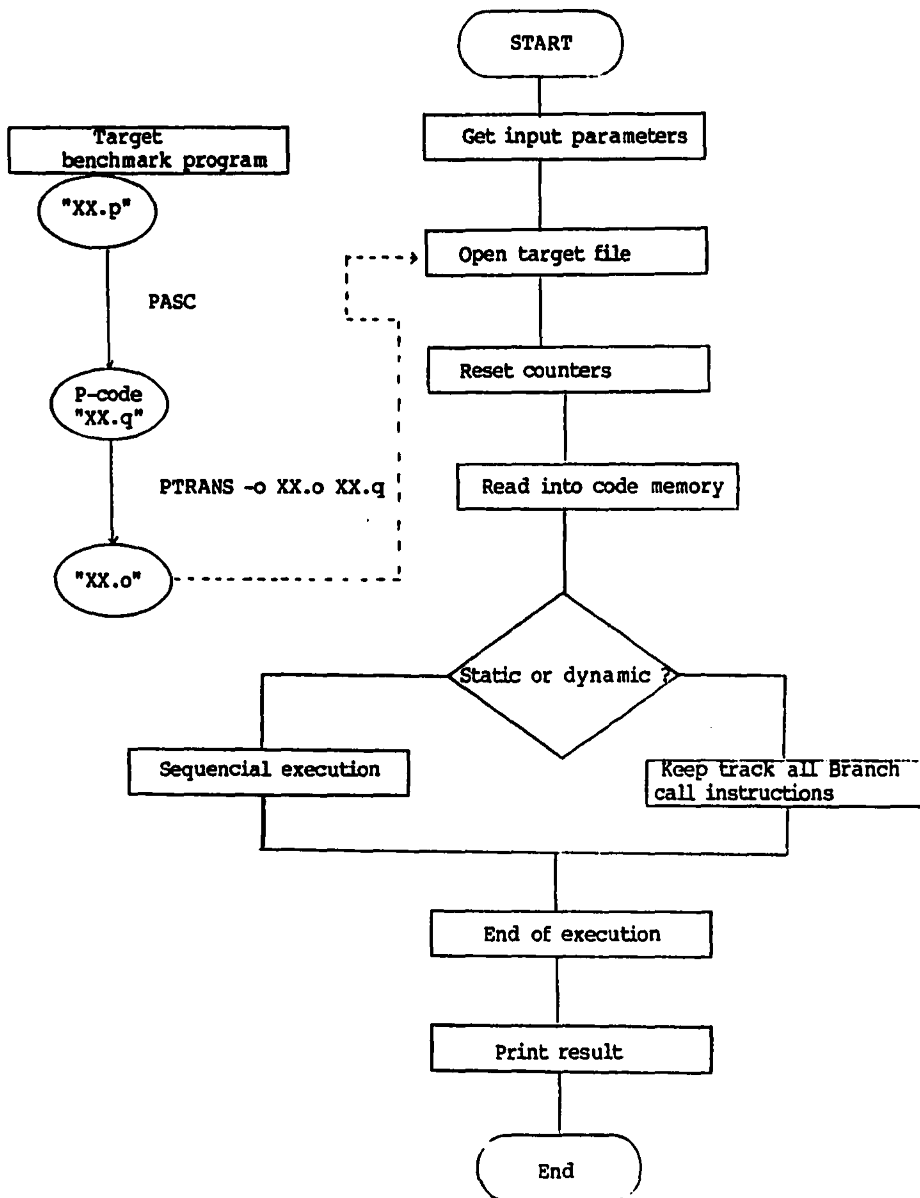


Figure B-1 Typical Scenario of Using the Ridge 32 Simulator

```
$ a.out
* Type the number of characters in File Name :
10
* Please type the File Name each character at once      :
a
c
k
e
r
m
a
n
.
o

* Thank you !
* Do you need code print out ? (y/n) :
y

Start of analyzing
a
c
k
e
r
m
a
n
.
o
 file.....
00000000DEA0FFFFFFFBA6A00004011E011F09B00
00000010000000088A7BE0000A7FE000801FEDFEE
000000200000000280C70F00180E000013C71F0020
00000030131 1A71F00108D000050C71F00208E10
00000040001D1401 1121A70E0018A72E002083B0
00000050FFC7A70F00108D0000301401A70E0018
00000060CFEE0020C72F00181411A72E0018A71E
00000070002083B0FFA3A70E0000CFEEFFE083B0
00000080FF97A70F0010C70F0010C7BF000001EF
00000090C7FF000857BBA7BE0000A7FE000801FE
000000A0DFEE0000101093B0FFFFFFFFFCE00008C
000000B0A70E001893B0FFFFFFFFFCE100084A71E
000000C0001893B0FFFFFFF21111A61010001122
000000D0A6201008802100 2A110311 16A70E0018
000000E0A71E002083B0FF31A6001004C6101000
000000F01311A6101000C6201008BA12FFDF1100
00000100A70E001893B0FFFFFFFFFC7BF000001EF
00000110C7FF000857BB0000000000000000000000

* End of FirstPass.  I1Max is   720
DEA0FFFFFFF8A6A0004011E011F09B00
000000B8A7BE0000A7FE000801FEDFEE
00000028C70F00180E000013C71F0020
1311A71F00108B000050C71F00208E10
001D14011121A70E0018A72E002083B0
FFC7A70F00108B0000301401A70E0018
CFEE0020C72F00181411A72E0018A71E
002083B0FFA3A70E0000CFEEFFE083B0
FF97A70F0010C70F0010C7BF000001FF
C7FF000857BBA7BE0000A7FE000801FE
DFEE0000101093B0FFFFFFFFCE00008C
```

```
0013973B0FFFFFFF21111A61010001122
A6201008B0210C2A11031116A70E0018
A71E0020B3B0FF31A6001504C6101000
1311A6101000C62C1308BA12FFDF1100
A70E001893B0FFFFFFFFFC7BF000001EF
C7FF000857B000000000000000000000

* End of SecondPass.  I2Max is   576

* End of ReadIntoCodeMemory.  PCMax is   288
* Start of Execution !!.
* Do you want Static data ? (y/n)
y
* Starting PC ? :
1
* Exceptional Code : 00 at PC   279
* Do you want to continue run ? (y/n):
n
*** End of Execution !! ***
**** MEMORY REFERENCE INSTRUCTIONS   =        41 ( 5.4666666E1   )
 *** LOAD  INSTRUCTIONS              =        18 ( 2.4000000E1   )
     ** LoadB =            0( 0.0000000E0  )
        * LoadbS :            0 * LoadbIS :           0 * LoadbL :           0 * LoadbIL :          0
     ** LoadH =            0( 0.0000000E0  )
        * LoadhS :            0 * LoadhIS :           0 * LoadhL :           0 * LoadhIL :          0
     ** Load  =           11( 1.4666666E1  )
        * LoadS  :            2 * LoadIS :           9 * LoadL  :           0 * LoadIL  :          0
     ** Loadd =            0( 0.0000000E0  )
        * LoaddS :            0 * LoaddIS :           0 * LoaddL :           0 * LoaddIL :          0
     ** Laddr  =            7( 9.3333330E0  )
        * LaddrS :            2 * LaddrIS :           2 * LaddrL :           1 * LaddrIL :          2
     ** Laddrp =            0( 0.0000000E0  )
        * LaddrpS :            0 * LaddrpIS :           0 * LaddrpL :           0 * LaddrpIL :          0
     ** Loadbp =            0( 0.0000000E0  )
        * LoadbpS :            0 * LoadbpIS :           0 * LoadbpL :           0 * LoadbpIL :          0
     ** Loadhp =            0( 0.0000000E0  )
        * LoadhpS :            0 * LoadhpIS :           0 * LoadhpL :           0 * LoadhpIL :          0
     ** Loadp =            0( 0.0000000E0  )
        * LoadpS :            0 * LoadpIS :           0 * LoadpL :           0 * LoadpIL :          0
     ** Loaddp =            0( 0.0000000E0  )
        * LoaddpS :            0 * LoaddpIS :           0 * LoaddpL :           0 * LoaddpIL :          0
    ** Laddrp =            0( 0.0000000E0  )
        * LaddrpS            0 * LaddrpIS :           0 * LaddrpL :           0 * LaddrpIL :          0

 *** STORE INSTRUCTIONS              =        23 ( 3.0666668E1   )
     ** StoreB =            0( 0.0000000E0  )
        * StorebS :            0 * StorebIS :           0 * StorebL :           0 * StorebIL :          0
     ** StoreH =            0( 0.0000000E0  )
        * StorehS :            0 * StorehIS :           0 * StorehL :           0 * StorehIL :          0
     ** Store  =           23( 3.0666668E1  )
        * StoreS :            5 * StoreIS :          18 * StoreL :           0 * StoreIL :          0

**** INTEGER ARITHMETIC INSTRUCTIONS  =         5 ( 6.6666669E0   )
 *** SINGLE-PRECISION                 =         5 ( 6.6666669E0   )
     ** Add    =            2( 2.6666667E0  )
        * Add Res. :            0 * Add I. :           2
     ** Div    =            0( 0.0000000E0  )
     ** Neg    =            0( 0.0000000E0  )
     ** Mpy    =            0( 0.0000000E0  )
        * Mpy Res. :            0 * Mpy I. :           0
     ** Rem    =            0( 0.0000000E0  )
     ** Sub    =            3( 4.0000000E0  )
        * Sub Res. :            0 * Sub I.           3
```

```
*** EXTENDED-PRECISION                    =          0 ( 0.0000000E0

**** REAL ARITHMETIC INSTRUCTIONS '       =          0 ( 0.0000000E0    )
 *** SINGLE-PRECISION                     =          0 ( 0.0000000E0    )
     ** Rneg   =          0( 0.0000000E0  )
     ** Radd   =          0( 0.0000000E0  )
     ** Rsub   =          0( 0.0000000E0  )
     ** Rmpy   =          0( 0.0000000E0  )
     ** Rdiv   =          0( 0.0000000E0  )
     ** Float  =          0( 0.0000000E0  )
     ** Fixr   =          0( 0.0000000E0  )
     ** Fixt   =          0( 0.0000000E0  )
     ** Makerd =          0( 0.0000000E0  )

 *** DOUBLE-PRECISION                       =          0 ( 0.0000000E0    )

**** BIT MANIPULATION INSTRUCTIONS          =          0 ( 0.0000000E0    )
     ** Asl    =          0( 0.0000000E0  )
     .   * Asl Reg. :          0      * Asl I :            0
     ** Asr    =          0( 0.0000000E0  )
         * Asr Reg. :         0      * Asr I :           0
     ** Cbit   =          0( 0.0000000E0  )
     ** Csl    =          0( 0.0000000E0  )
         * Csl Reg. :         0      * Csl I :           0
     ** Dlsl   =          0( 0.0000000E0  )
         * Dlsl Reg.:         0      * Dlsl I :           0
     ** Dlsr   =          0( 0.0000000E0  )
         * Dlsr Reg.:         0      * Dlsr I :           0
     ** Lsl    =          0( 0.0000000E0  )
         * Lsl Reg. :         0      * Lsl I :           0
     ** Lsr    =          0( 0.0000000E0  )
         * Lsr Reg. :         0      * Lsr I :           0
     ** Sbit   =          0( 0.0000000E0  )
     ** Tbit   =          0( 0.0000000E0  )

**** LOGICAL INSTRUCTIONS                   =          0 ( 0.0000000E0    )
     ** And    =          0( 0.0000000E0  )
         * And Reg. :          0 * And I. :         0
     ** Not    =          0( 0.0000000E0  )
     ** Or     =          0( 0.0000000E0  )
     ** Xor    =          0( 0.0000000E0  )

**** TEST INSTRUCTIONS                      =          0 ( 0.0000000E0    )
     ** TestGT  =          0     ** TestLT  =          0
     ** TestEQ  =          0     ** TestIGT =          0
     ** TestILT =          0     ** TestIEQ =          0
     ** TestLE  =          0     ** TestGE  =          0
     ** TestNE  =          0     ** TestILE =          0
     ** TestIGE =          0     ** TestINE =          0

**** DATA MOVEMENT INSTRUCTIONS             =         12 ( 1.6000000E1    )
     * Move Reg. :              4 * Move I. :         8

**** COMPARISON INSTRUCTIONS                =          0 ( 0.0000000E0    )
     ** Dcomp   =          0     ** Drcomp  =          0
     ** Lcomp   =          0     ** Rcomp   =          0

**** SIGN EXTEND INSTRUCTIONS               =          0 ( 0.0000000E0    )
     ** Seb     =          0     ** Seh     =          0
```

```
**** BRANCH AND CALL INSTRUCTIONS        =          17 ( 2.2666660E1    )
 *** BRANCH INSTRUCTIONS                 =           7( 9.3333330E0   )
     ** BrGTS  =            0     ** BrGTL  =           0
     ** BrEQS  =            0     ** BrEQL  =           0
     ** BrIGTS =            0     ** BrIGTL =           0
     ** BrILTS =            0     ** BrILTL =           0
     ** BrIEQS =            0     ** BrIEQL =           0
     ** BrLES  =            1     ** BrLEL  =           0
     ** BrNES  =            1     ** BrNEL  =           0
     ** BrS    =            2     ** BrL    =           1
     ** BrILES =            0     ** BrILEL =           0
     ** BrIGES =            0     ** BrIGEL =           0
     ** BrINES =            2     ** BrINEL =           0

 *** CALL INSTRUCTIONS                   =           8( 1.0666667E1   )
     ** CallS  =            4     ** CallL  =           4
     ** Callr  =            0

 *** LOOP INSTRUCTIONS                   =           0( 0.0000000E0   )
     ** LoopS  =            0     ** LoopL  =           0

 *** RETURN INSTRUCTIONS                 =           2( 2.6666667E0   )

************************ TOTAL INSTRUCTIONS ....          75


*****************************************************************
************************ ANALYSIS RESULT ************************
*****************************************************************

####     Number of Reg. to Reg. format instructions    !           19
           % in total Memory Reference instructions    !   2.5333333E1

  ##     Number of Short displacement instructions     !           48
           % in total Memory Reference instructions    !   6.4000000E1

  ##     Number of Long displacement instructions      !            8
           % in total Memory Reference instructions    !   1.0666667E1

  ##     Number of DIRECT MEMORY ACCESS instructions   !           25
           % in total Memory Reference instructions    !   4.4642858E1

  ##     Number of INDEXED MEMORY ACCESS instructions  !           31
           % in total Memory Reference instructions    !   5.5357141E1

####     Number of total BRANCH instructions           !            7
                     % in Total Instructions           !   9.3333330E0

  ##     Number of Conditional Branch instructions     !            4
                    % in total Branch instructions     !   5.7142858E1

  ##     Number of Unconditional Branch instructions   !            3
                      % in total Branch instructions   !   4.2857141E1

####     Total # of code space access instructions     !            0
             % in total memory access instructions     !   0.0000000E0

####     Total # of data space access instructions     !           56
             % in total memory access instructions     !   1.0000000E2
```

```
####      Average length of loop displacement in byte   :  0.0000000E0
####    Average length of branch displacement in byte :  0.0000000E0
####    Average length of call displacement in byte   :  0.0000000E0
$
```

APPENDIX C

RIDGE 32 SIMULATOR LISTING

Oct 17 13:40 1986 fastt.r Page 1

```
PROGRAM RIDGE32(INPUT,OUTPUT);

(****************************************************************)
(*  Ridge 32 Simulator program.                                 *)
(*  Programmed by Seok Tae  Yoon                                *)
(*  Department of Electrical and Computer Engineering          *)
(*  Florida Atlantic University                                 *)
(*  Oct. 19th 1986.                                             *)
(****************************************************************)

CONST
MaxFirstPassMemory  = 9000;     {Maximum size of FirstPassMemory}
MaxCodeMemory       = 1500;     {Maximum size of CodeMemory}
MaxDataMemory       = 100000;   {Maximum size of DataMemory}

TYPE
RType             = array[0..15,0..31] of integer;{Register Type}
FourCharType      = array[0..3] of char;
EightCharType     = array[0..7] of char;
FourBitType       = array[0..3] of integer;
SixteenBitType    = array[0..15] of integer;
ThirtytwoBitType  = array[0..31] of integer;
ByteType          = array[0..1] of char;
HalfWordType      = array[0..1] of ByteType;
WordType          = array[0..3] of ByteType;
CodeMemoryType    = array[1..MaxCodeMemory] of ByteType;
DataMemoryType    = array[0..MaxDataMemory] of ByteType;
StringBody        = record{This type definition is required by Ridge 32 PASCAL}
                     length : integer;
                     chars  : array[1..1] of char;
                    end;
string            = ^StringBody;

VAR
R                 : RType;          {Register}
R1Index,
R2Index,
R3Index,
EA                : Integer;        {Effective Address}
FourBit           : FourBitType;
Static,                             {Flag for Static analysis}
Error,                              {Flag for exceptional OP Code}
CodeListing       : boolean;        {Flag for object code listing}
TempChar          : char;
DataFile          : text;
FileName          : string;
FirstPassMemory   : packed array[1..MaxFirstPassMemory] of char;
CharArray         : array[1..1] of char;
DataMemory        : DataMemoryType;
CodeMemory        : CodeMemoryType;
OPCode,
Operands,
TempByte
                  : ByteType;
 TempHalfWord     : HalfWordType;
 TempWord         : WordType;
```

```
(*This variable will hold the accumulated branch displacement.*)
TotalBr,

(*This variable will hold the accumulated loop displacement.*)
TotalLoop,

(*This variable will hold the accumulated call displacement.*)
TotalCall        : Integer;
M, NumberOfName,
PC, PCmax,
i, i1Max, i2Max,

(* Individual instruction counter.*)

MoveC,NegC,AddC,SubC,MpyC,DivC,RemC,NotC,OrC,XorC,AndC,CbitC,SbitC,TbitC,ChkC,
NopC,MoveIC,AddIC,SubIC,MpyIC,NotIC,AndIC,ChkIC,
FixtC,FixrC,RnegC,RaddC,RsubC,RmpyC,RdivC,MakerdC,LcompC,FloatC,RcompC,LaddC,LsubC,EmpyC,EdivC,
DfixtC,DfixrC,DnegC,DaddC,DsubC,DmpyC,DdivC,MakedrC,DcompC,DfloatC,DrcompC,TrapC,
SupC,LupC,RumC,LdregsC,TransC,DirtC,MoveRSRC,MoveSRRC,MaintC,ReadC,WriteC,
TestGTC,TestLTC,TestEQC,CallrC,TestIGTC,TestILTC,TestIEQC,RetC,TestLEC,TestGEC,TestNEC,
RcallC,TestILEC,TestIGEC,TestINEC,
LslC,LsrC,AslC,AsrC,DlslC,DlsrC,CslC,SebC,
LslIC,LsrIC,AslIC,AsrIC,DlslIC,DlsrIC,CslIC,SehC,
BrGTSC,BrEQSC,CallSC,BrIGTSC,BrILTSC,BrIEQSC,LoopSC,BrLESC,BrNESC,BrSC,BrILESC,BrIGESC,BrINESC,
BrGTLC,BrEQLC,CallLC,BrIGTLC,BrILTLC,BrIEQLC,LoopLC,BrLELC,BrNELC,BrLC,BrILELC,BrIGELC,BrINELC,
StorebSC,StorebISC,StorehSC,StorehISC,StoreSC,StoreISC,StoredSC,StoredISC,
StorebLC,StorebILC,StorehLC,StorehILC,StoreLC,StoreILC,StoredLC,StoredILC,
LoadbSC,LoadbISC,LoadhSC,LoadhISC,LoadSC,LoadISC,LoaddSC,LoaddISC,LaddrSC,LaddrISC,
LoadbLC,LoadbILC,LoadhLC,LoadhILC,LoadLC,LoadILC,LoaddLC,LoaddILC,LaddrLC,LaddrILC,
LoadbpSC,LoadbpISC,LoadhpSC,LoadhpISC,LoadpSC,LoadpISC,LoaddpSC,LoaddpISC,LaddrpSC,LaddrpISC,
LoadbpLC,LoadbpILC,LoadhpLC,LoadhpILC,LoadpLC,LoadpILC,LoaddpLC,LoaddpILC,LaddrpLC,LaddrpILC,


(* Group Counter *)

TotalInstructionC,
TotalMemoryReferenceC,
MemoryReferenceC,IntegerArithmeticC,RealArithmeticC,
LoadInstC,StoreInstC,LaddrC,
SingleIntegerC,ExtendedIntegerC,
SingleRealC,DoubleRealC,
BitManipulationC,LogicC,
ComparisonC,SignExtendC,BranchCallC,
LoadT,LoadbT,LoaddT,LoadhT,LaddrT,
LoadbpT,LoadhpT,LoaddpT,LoadpT,LaddrpT,
StoreT,StorebT,StoredT,StorehT,
MoveT,AddT,SubT,MpyT,NotT,AndT,ChkT,
LslT,LsrT,AslT,AsrT,DlslT,DlsrT,CslT,
TestT,BrT,CallT,LoopT,
[illegible]ToRegFormatC,ShortOffsetC,LongOffsetC,
DirectC,IndexC,
ConditionalBrC,UnconditionalBrC,
CodeSpaceAccessC,DataSpaceAccessC
                                        : Integer;
```

```
(* Represent the average length of loop & branch displacement.*)

LoopLength,BrLength,CallLength   : Real;


(* Those Following variables will hold distribution results in Percentage.*)

RegToRegFormatP,ShortOffsetP,LongOffsetP,DirectP,IndexP,
ConditionalBrP,UnconditionalBrP,
CodeSpaceAccessP,DataSpaceAccessP,

MemoryP,IntegerP,RealP,
LoadFastP,StoreFastP,LoadirP,
SingleIntegerP,ExtendedIntegerP,
SingleRealP,DoubleRealP,
BitManipulationP,LogicP,TestP,MoveP,
ComparisonP,SignExtendP,BranchCallP,
LoadP,LoadbP,LoadhP,LoadnP,
LoadaP,LoadaaP,LoadabP,LoadrP,
StoreP,StorebP,StoredP,StorehP,
LoadirP,
AddP,DivP,NegP,MpyP,RemP,SubP,
RnegP,RaddP,RsubP,RmpyP,RdivP,FloatP,FixrP,FixtP,MakerdP,
AslP,AsrP,CbitP,CslP,DlslP,DlsrP,LslP,LsrP,SbitP,TbitP,
AndP,NotP,OrP,XorP,
DcomP,DrcomP,LcomP,RcomP,
SetP,SehP,BrP,CallP,CallrP,LoopP,RetP
                    : Real;

(These functions are external library in Ridge PASCAL.}
Function NewString (length:Integer):String;External;
Procedure OpenFile(Var f:text;name:string;mode:char);external;
Procedure Closefile(var f:text);external;


(*************************************************)
(* Procedure to remove all blanks from XX.o file. *)
(*************************************************)
Procedure FirstPass;

VAR
 i1         : integer;
 TempChar : char;

begin

  i1 := 1;
  read(datafile,tempchar);
  while not EOF(datafile) do
   begin
    while tempchar = ' ' do   (Skip in case of blank character}
     read (datafile,tempchar);
    If tempchar = CHR(10)      (To check whether the character is Carrage return)
    then(Skip in case of Carriage Reture.}
       read (datafile,tempchar)
    else begin (Otherwise read into FirstPassMemory}
```

```
          FirstPassMemory[i1] := Tempchar;
          i1 := i1 + 1;
          read (datafile,tempchar);
         end {Else}
  end;{while}

 i1Max := i1 - 1;
 If CodeListing{Code listing flag is TRUE ?}
   Then For i1 := 1 to i1Max Do{Then print out the characters in FirstPassMemory}
          Begin Write (FirstPassMemory[i1]);
                If i1 MOD 40 = 0 Then Writeln;
          End;{For}
 Writeln;
 writeln ('* End of FirstPass.  I1Max is',i1Max:6);

end;{FirstPass}

(****************************************************************************)
(* Procedure to remove all numberings to get only object code from XX.o file. *)
(****************************************************************************)

Procedure SecondPass;

VAR
 Remainder,i1,i2 : integer;

begin
 i1 := 1;
 i2 := 1;
 For i1 := 1 to i1Max do
  begin
   Remainder := i1 MOD 40;
   If  Remainder <= 8{To check  the Charator belongs to numbering ?}
    Then begin{Yes}
          If Remainder = 0{Skip first seven characters of numbering}
             Then Begin
                   FirstPassMemory [i2] := FirstPassMemory[i1];
                   i2 := i2+1;
                  end
             Else {Just increase i1.}
         end{then}

    Else Begin
           FirstPassMemory [i2] := FirstPassMemory[i1];
           i2 := i2+1;
         end;{Else}
  end;{For}

  i2Max := i2 - 1;
  If CodeListing{CodeListing required ?}
      Then  For i2 := 1 to i2Max do{Yes}
              Begin  Write (FirstPassMemory[i2]);
               If i2 MOD 32 = 0 Then Writeln;
             End;{For}
  Writeln;
  Writeln('* End of SecondPass.  I2Max is',i2Max:6);
```

```
end;{SecondPass}

(****************************************************************************)
(* Procedure to store code into CodeMemory in the unit of two characters. *)
(****************************************************************************)
Procedure ReadIntoCodeMemory;

VAR
 i2      : integer;
 Byte    : ByteType;

begin
 i2 := 1;
 PC := 1;
 While i2 <= i2Max do
  begin
   Byte[0] := First(PassMemory[i2]);{To pack two characters into one Byte}
   i2 := i2 + 1;
   Byte[1] := First(PassMemory[i2]);
   CodeMemory[PC] := Byte;          {Save that byte into one component of CodeMemory}
   i2 := i2+1;
   PC := PC + 1;
  end;{while}
 PCMax := PC - 1;{To return the Max. index of CodeMemory}
 Writeln;
 Writeln ('* End of ReadIntoCodeMemory.  PCMax is',PCMax:6);

end;{ReadIntoCodeMemory}

(*************************************************************************)
(* Procedure to set 4 bit pattern of one character. Out[0] has LSB. *)
(*************************************************************************)

Procedure CharToBit (Ii  : Char;
                     Var Out : FourBitType );
Var
i       : Integer;

begin
Case Ii of
'0': Begin
     Out[0] := 0;
     Out[1] := 0;
     Out[2] := 0;
     Out[3] := 0;
     end;
'1': Begin
     Out[0] := 1;
     Out[1] := 0;
     Out[2] := 0;
     Out[3] := 0;
     End;
 '2': Begin
     Out[0] := 0;
     Out[1] := 1;
```

```
     Out[2] := 0;
     Out[3] := 0;
     end;
'3': Begin
        Out[0] := 1;
        Out[1] := 1;
        Out[2] := 0;
        Out[3] := 0;
     end;
'4': Begin
        Out[0] := 0;
        Out[1] := 0;
        Out[2] := 1;
        Out[3] := 0;
     end;
'5': begin
        Out[0] := 1;
        Out[1] := 0;
        Out[2] := 1;
        Out[3] := 0;
     end;
'6': begin
        Out[3] := 0;
        Out[2] := 1;
        Out[1] := 1;
        Out[0] := 0;
     end;
'7': begin
        Out[3] := 0;
        Out[2] := 1;
        Out[1] := 1;
        Out[0] := 1;
     end;
'8': Begin
        Out[3] := 1;
        Out[2] := 0;
        Out[1] := 0;
        Out[0] := 0;
     End;
'9': begin
        Out[3] := 1;
        Out[2] := 0;
        Out[1] := 0;
        Out[0] := 1;
     end;
'A': begin
        Out[3] := 1;
        Out[2] := 0;
        Out[1] := 1;
        Out[0] := 0;
     end;
'B': begin
        Out[3] := 1;
        Out[2] := 0;
        Out[1] := 1;
        Out[0] := 1;
```

```
      end;
'C': Begin
         Out[3] := 1;
         Out[2] := 1;
         Out[1] := 0;
         Out[0] := 0;
      end;
'D': Begin
         Out[3] := 1;
         Out[2] := 1;
         Out[1] := 0;
         Out[0] := 1;
      end;
'E': Begin
         Out[3] := 1;
         Out[2] := 1;
         Out[1] := 1;
         Out[0] := 0;
      end;
'F': Begin
         Out[3] := 1;
         Out[2] := 1;
         Out[1] := 1;
         Out[0] := 1;
      end;
end;{Case}
end;{CharToBit}


(*******************************************************************)
(* Procedure to Convert 16 Bit to Two's complement                *)
(*******************************************************************)
Procedure SixteenBitTwos ( X : SixteenBitType;
                           Var XX : SixteenBitType );
Var
         i : Integer;
         C : SixteenBitType;{To contain the carry bit.}

Begin
{Get 1's Complement First.}
For i := 0 TO 15 Do
 Begin
   If X[i] = 0 Then XX[i] := 1
              else XX[i] := 0;
   C[i] := 0;
 End;{For}
 {Get 2's Complement.}
If XX[0] = 1 Then Begin
                     XX[0] := 0;
                     C[1] := 1;
                     For i := 1 to 15 Do
                      Begin
                             If XX[i] = 1 Then If C[i] = 1 Then
                                                            Begin
                                                             XX[i] := 0;
```

```
                                                                       C[i+1] := 1;
                                                                     End
                                                                   Else
                                                      Else XX[i] := C[i];
                           End(For)
                      End(Then)
                 Else XX[0] := 1;
End(SixteenBitTwos)



(*****************************************************************)
(* Procedure to return Short Effective Address(EA) in Integer Form *)
(*****************************************************************)
Procedure GetShortEA;
Var
        FourBit   : FourBitType;
        X,Y           : SixteenBitType;
begin
CharToBit (CodeMemory[PC+3][1],FourBit);
X[0] := FourBit[0];
X[1] := FourBit[1];
X[2] := FourBit[2];
X[3] := FourBit[3];
CharToBit(CodeMemory[PC+3][0],Fourbit);
X[4] := FourBit[0];
X[5] := FourBit[1];
X[6] := FourBit[2];
X[7] := FourBit[3];
CharToBit(CodeMemory[PC+2][1],FourBit);
X[8]  := FourBit[0];
X[9]  := FourBit[1];
X[10] := FourBit[2];
X[11] := FourBit[3];
CharToBit(CodeMemory[PC+2][0],FourBit);
X[12] := FourBit[0];
X[13] := FourBit[1];
X[14] := FourBit[2];
X[15] := FourBit[3];

If X[15] = 1
Then begin
        SixteenBitTwos (X,Y);
        EA := -(y[0]+2*y[1]+4*y[2]+8*y[3]+16*y[4]+32*y[5]+64*y[6]+128*y[7]
              +256*y[8]+512*y[9]+1024*y[10]+2048*y[11]+4096*y[12]+8192*y[13]
              +16384*y[14]);
     end
Else EA := x[0]+2*x[1]+4*x[2]+8*x[3]+16*x[4]+32*x[5]+64*x[6]+128*x[7]+
           256*x[8]+512*x[9]+1024*x[10]+2048*x[11]+4096*x[12]+8192*x[13]+
           16384*x[14];
end;(GetShortEA)

(**********************************************)
(* Procedure to return Two's Comelement of 32 bit. *)
(**********************************************)
Procedure ThirtyTwoBitTwos ( X : ThirtyTwoBitType;
```

```
                              Var  XX : ThirtyTwoBitType );
Var
        i : Integer;
        C : ThirtyTwoBitType;{To save carry}

Begin
{To Get 1's Complement}
 For i := 0 to 31 Do
        begin
                If X[i] = 0 then XX[i] := 1
                            else XX[i] := 0;
                C[i] := 0;
        end;{For}
{To get 2's Complement}
 If XX[0] = 1 then begin
                        XX[0] := 0;
                        C[1] := 1;
                        For i := 1 to 31 Do
                         Begin
                                If XX[i] = 1 then If C[i] = 1
                                                        then begin
                                                                XX[i] := 0;
                                                                C[I+1] := 1;
                                                             end
                                                        else

                                               else XX[i] := C[i]
                          end{For}
                    end{then}
             else XX[0] := 1;
end;{ThirtyTwoBitTwos}


(*****************************************************************)
(* Procedure to return Long Effective Address in Integer Form. *)
(*****************************************************************)

Procedure GetLongAdd;
Var
FourBit   : FourBitType;
X,Y : ThirtyTwoBitType;

begin
CharToBit(CodeMemory[PC+5][1],FourBit);
X[0] := FourBit[0];
X[1] := FourBit[1];
X[2] := FourBit[2];
X[3] := FourBit[3];
CharToBit(CodeMemory[PC+5][0],Fourbit);
X[4] := FourBit[0];
X[5] := FourBit[1];
X[6] := FourBit[2];
X[7] := FourBit[3];
CharToBit(CodeMemory[PC+4][1],FourBit);
X[8]  := FourBit[0];
X[9]  := FourBit[1];
```

```
X[10] := FourBit[2];
X[11] := FourBit[3];
CharToBit(CodeMemory[PC+4][0],FourBit);
X[12] := FourBit[0];
X[13] := FourBit[1];
X[14] := FourBit[2];
X[15] := FourBit[3];
CharToBit(CodeMemory[PC+3][1],FourBit);
X[16] := FourBit[0];
X[17] := FourBit[1];
X[18] := FourBit[2];
X[19] := FourBit[3];
CharToBit(CodeMemory[PC+3][0],FourBit);
X[20] := FourBit[0];
X[21] := FourBit[1];
X[22] := FourBit[2];
X[23] := FourBit[3];
CharToBit(CodeMemory[PC+2][1],FourBit);
X[24] := FourBit[0];
X[25] := FourBit[1];
X[26] := FourBit[2];
X[27] := FourBit[3];
CharToBit(CodeMemory[PC+2][0],FourBit);
X[28] := FourBit[0];
X[29] := FourBit[1];
X[30] := FourBit[2];
X[31] := FourBit[3];


If X[31] = 1 then
                begin
                 ThirtyTwoBitTwos (X,Y);
                 EA := -(y[0]+2*y[1]+4*y[2]+8*y[3]+16*y[4]+32*y[5]+64*y[6]+128*y[7]+
                         256*y[8]+512*y[9]+1024*y[10]+2048*y[11]+4096*y[12]+8192*y[13]+
                         16384*y[14]+32768*y[15]+65536*y[16]+131072*y[17]+262144*y[18]+
                         524288*y[19]+1048576*y[20]+2097152*y[21]+4194304*y[22]+
                         8388608*y[23]+16777216*y[24]+33554432*y[25]+67108864*y[26]+
                         134217728*y[27]+268435456*y[28]+536870912*y[29]+1073741824*y[30]);
                end
             else EA := x[0]+2*x[1]+4*x[2]+8*x[3]+16*x[4]+32*x[5]+64*x[6]+128*x[7]+256*x[8]
                        +512*x[9]+1024*x[10]+2048*x[11]+4096*x[12]+8192*x[13]+16384*x[14]+
                        32768*x[15]+65536*x[16]+131072*x[17]+262144*x[18]+524288*x[19]+
                        1048576*x[20]+2097152*x[21]+4194304*x[22]+8388608*x[23]+16777216*x[24]
                        +33554432*x[25]+67108864*x[26]+134217728*x[27]+268435456*x[28]+
                        536870912*x[29]+1073741824*x[30];
end;{GetLongAdd}


(**********************************************)
(* Procedure to reset all bit of a register. *)
(**********************************************)
Procedure ClearRegister (Z : Integer);
Var
        i : Integer;
```

```
For i := 0 to 31 do R[I1,i] := 0;
end;

(**********************************************)
(* Function to convert a character to integer. *)
(**********************************************)
Function Value (Ii: Char):Integer;

begin
Case Ii of
'0' : Value := 0;
'1' : Value := 1;
'2' : Value := 2;
'3' : Value := 3;
'4' : Value := 4;
'5' : Value := 5;
'6' : Value := 6;
'7' : Value := 7;
'8' : Value := 8;
'9' : Value := 9;
'A' : Value := 10;
'B' : Value := 11;
'C' : Value := 12;
'D' : Value := 13;
'E' : Value := 14;
'F' : Value := 15;
end{Case}
end;{Value}


(**********************************************)
(* Procedure to load 8 bits in LSBs of [illegible] *)
(**********************************************)
Procedure LoadByte( TempByte : ByteType;
                    Var RIndex : Integer);
Var
             i : Integer;
       FourBit : FourBitType;

begin
For i := 8 to 31 Do R[Rindex,i] := 0;
CharToBit (TempByte[0],FourBit);
R[RIndex,4] := FourBit[0];
R[RIndex,5] := FourBit[1];
R[RIndex,6] := FourBit[2];
R[RIndex,7] := FourBit[3];
CharToBit (TempByte[1],FourBit);
R[RIndex,0] := FourBit[0];
R[RIndex,1] := FourBit[1];
R[RIndex,2] := FourBit[2];
R[RIndex,3] := FourBit[3];
end;{LoadByte}


(**********************************************)
(* Procedure to load 16 bits to LSBs of [illegible] *)
```

```
(*****************************************************************)
Procedure LoadHalfWord (Ii : HalfWordType;
                        Var Rx : Integer    );
Var
        i : Integer;
        FourBit : FourBitType;

Begin
 For i := 16 to 31 Do R[Rx,i] := 0;
 LoadByte  (Ii[1],Rx );
 CharToBit ( Ii[0][0],FourBit );
 R[Rx,15] := FourBit[3];
 R[Rx,14] := FourBit[2];
 R[Rx,13] := FourBit[1];
 R[Rx,12] := FourBit[0];
 CharToBit ( Ii[0][1],FourBit);
 R[Rx,11] := FourBit[3];
 R[Rx,10] := FourBit[2];
 R[Rx,9] := FourBit[1];
 R[Rx,8] := FourBit[0];
End;


(**********************************************************)
(* Procedure to load 32 bits of [illegible]               *)
(**********************************************************)
Procedure LoadWord (Ii : WordType;
                    Var Rx : Integer );
Var
        FourBit : FourBitType;

begin
 TempHalfWord[0] := Ii[2];
 TempHalfWord[1] := Ii[3];
 LoadHalfWord (TempHalfWord,Rx);
 CharToBit (Ii[1][0],FourBit);
 R[Rx,23] := FourBit[3];
 R[Rx,22] := FourBit[2];
 R[Rx,21] := FourBit[1];
 R[Rx,20] := FourBit[0];
 CharToBit (Ii[1][1],FourBit);
 R[Rx,19] := FourBit[3];
 R[Rx,18] := FourBit[2];
 R[Rx,17] := FourBit[1];
 R[Rx,16] := FourBit[0];
 CharToBit (Ii[0][0],FourBit);
 R[Rx,31] := FourBit[3];
 R[Rx,30] := FourBit[2];
 R[Rx,29] := FourBit[1];
 R[Rx,28] := FourBit[0];
 CharToBit (Ii[0][1],FourBit);
 R[Rx,27] := FourBit[3];
 R[Rx,26] := FourBit[2];
 R[Rx,25] := FourBit[1];
 R[Rx,24] := FourBit[0];
```

```
End;



(*************************************************************************)
(* Procedure to return a character corresponding to input integer. *)
(*************************************************************************)
Procedure IntegerToChar(Ii : Integer;
                        Var Out: Char);

begin
Case Ii of
0 : Out := '0';
1 : Out := '1';
2 : Out := '2';
3 : Out := '3';
4 : Out := '4';
5 : Out := '5';
6 : Out := '6';
7 : Out := '7';
8 : Out := '8';
9 : Out := '9';
10 : Out := 'A';
11 : Out := 'B';
12 : Out := 'C';
13 : Out := 'D';
14 : Out := 'E';
15 : Out := 'F';
end;{case}
end;{IntegerToChar}



(*************************************************************************)
(* Procedure to convert one byte to corresponding two characters. *)
(*************************************************************************)
Procedure GetByte (Ii : Integer;
                   Var Out : ByteType);
Var
        i : Integer;
        TempChar : Char;

begin
i := R[Ii,4]+2*R[Ii,5]+4*R[Ii,6]+8*R[Ii,7];
IntegerToChar(i,TempChar);
Out[0] := TempChar;
i := R[Ii,0]+2*R[Ii,1]+4*R[Ii,2]+8*R[Ii,3];
IntegerToChar(i,TempChar);
Out[1] := TempChar;
end;


(*********************************************************)
(* Procedure to return a Half Word of "Ii"th register. *)
```

```
(**************************************************)
Procedure GetHalfWord (Ii : Integer;
                       Var Out : HalfWordType);
Var
        i : Integer;
        TempByte : ByteType;

begin
 i := R[Ii,0]+2*R[Ii,1]+4*R[Ii,2]+8*R[Ii,3];
 IntegerToChar (i,TempByte[1]);
 i := R[Ii,4]+2*R[Ii,5]+4*R[Ii,6]+8*R[Ii,7];
 IntegerToChar (i,TempByte[0]);
 Out[1] := TempByte;
 i := R[Ii,8]+2*R[Ii,9]+4*R[Ii,10]+8*R[Ii,11];
 IntegerToChar (i,TempByte[1]);
 i := R[Ii,12]+2*R[Ii,13]+4*R[Ii,14]+8*R[Ii,15];
 IntegerToChar (i,TempByte[0]);
 Out[0] := TempByte;
end;


(**********************************************)
(* Procedure to return a Word of 'Ii'th register. *)
(**********************************************)
Procedure GetWord (Ii : Integer;
                   Var Out : WordType );
Var
        i : Integer;
        TempByte : ByteType;

begin
 i := R[Ii,0]+2*R[Ii,1]+4*R[Ii,2]+8*R[Ii,3];
 IntegerToChar (i,TempByte[1]);
 i := R[Ii,4]+2*R[Ii,5]+4*R[Ii,6]+8*R[Ii,7];
 IntegerToChar (i,TempByte[0]);
 Out[3] := TempByte;
 i := R[Ii,8]+2*R[Ii,9]+4*R[Ii,10]+8*R[Ii,11];
 IntegerToChar(i,TempByte[1]);
 i := R[Ii,12]+2*R[Ii,13]+4*R[Ii,14]+8*R[Ii,15];
 IntegerToChar (I,TempByte[0]);
 Out[2] := TempByte;
 i := R[Ii,16]+2*R[Ii,17]+4*R[Ii,18]+8*R[Ii,19];
 IntegerToChar (i,TempByte[1]);
 i := R[Ii,20]+2*R[Ii,21]+4*R[Ii,22]+8*R[Ii,23];
 IntegerToChar (i,TempByte[0]);
 Out[1] := TempByte;
 i := R[Ii,24]+2*R[Ii,25]+4*R[Ii,26]+8*R[Ii,27];
 IntegerToChar(i,TempByte[1]);
 i := R[Ii,28]+2*R[Ii,29]+4*R[Ii,30]+8*R[Ii,31];
 IntegerToChar(i,TempByte[0]);
 Out[0] := TempByte;
end;
```

```
[illegible]

(*****************************************************************)
(* Function to return integer [illegible] *)
(*****************************************************************)
Function RegisterValue ( I: [illegible] Integer [illegible] Integer;
Var
        i : Integer;
        X,Y : ThirtyTwoBitType;

Begin
 For i := 0 to 31 Do X[i] := [illegible]
 If X[31] = 1
        Then Begin ( Negative )
                ThirtyTwoBit[illegible]
                RegisterValue := [illegible]
                                [illegible]
                                [illegible]
                                [illegible]
                                2097152*x[21]+4194304*x[22]+8388608*x[23]+16777216*x[24]+
                                33554432*x[25]+67108864*x[26]+134217728*x[27]+268435456*x[28]
                                +536870912*x[29]+1073741824*x[30]);
          end
        Else RegisterValue := X[0]+2*x[1]+4*x[2]+8*x[3]+16*x[4]+32*x[5]+64*x[6]+128*x[7]+
                                256*x[8]+512*x[9]+1024*x[10]+2048*x[11]+4096*x[12]+8192*x[13]
                                +16384*x[14]+32768*x[15]+65536*x[16]+131072*x[17]+262144*x[18]
                                +524288*x[19]+1048576*x[20]+2097152*x[21]+4194304*x[22]+
                                8388608*x[23]+16777216*x[24]+33554432*x[25]+67108864*x[26]+
                                134217728*x[27]+268435456*x[28]+536870912*x[29]+1073741824*x[30];
end;


(******************************************)
(* Procedure to change base 10 to base 2. *)
(******************************************)
Procedure Base10To2 (Ii: Integer;
                      Var Y : ThirtyTwoBitType );
Var
        i,XX,ix : Integer;
        X       : ThirtyTwoBitType;

begin
 {Clear}
 For i := 0 to 31 Do X[i] := 0;
 i := 0;
 ix := ABS (Ii);
 While ix [illegible] 0 Do
   begin
        XX := ix DIV 2;
        X[i] := ix MOD 2;
        ix := XX;
        i := i+1;
   end;{While}
{In case of negative, [illegible] complement}
 If Ii [illegible] 0 Then ThirtyTwo[illegible] (X,Y)
          Else For [illegible]
   [illegible]
```

```
(*****************************************************************)
(* Procedure to shift a certain register to left filling right with 0. *)
(*****************************************************************)
Procedure LogicalShiftLeft(Ii : Integer;
                           Var n : Integer);
Var i,j : Integer;

Begin
If n >= 1 Then
       Begin
         For j := 1 to n Do
          Begin
            For i := 31 Downto 1 Do R[Ii,i] := R[Ii,i-1];
            R[Ii,0] := 0;
          End;(For)
       End;(Then)
end;


(*****************************************************************)
(* Procedure to shift a certain register to right filling left with MSB. *)
(*****************************************************************)
Procedure ArithmeticShiftRight (Ii : Integer;
                                Var n : Integer );
Var i,j : Integer;

Begin
If n >= 1 Then
         Begin
            For j := 1 to n Do For i := 0 to 30 Do R[Ii,i] := R[Ii,i+1];
         End;(then)
end;




(************************************************************)
(************************************************************)
(* Each procedure represent one assembly instruction of Riddle32. *)
(************************************************************)
(************************************************************)


Procedure Move;
Var
         i : Integer;

begin
 MoveC := MoveC+1;
 For i := 0 to 31 Do
```

```
        RFRdIndex,i] := RFRsIndex,i];
 PC := PC+2;
end;


Procedure Neg;

begin
 NegC := NegC+1;
 PC := PC+2;
end;


Procedure Add;
Var i,Rx : Integer;
     ThirtyTwoBit : ThirtyTwoBitType;

begin
 AddC := AddC+1;
 Rx := RegisterValue(RdIndex);
 Rx := Rx+RegisterValue(RsIndex);
 Base10To2 (Rx,ThirtyTwoBit);
 For i := 0 to 31 Do RFRdIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure Sub;
Var i,Rx : Integer;
     ThirtyTwoBit : ThirtyTwoBitType;
begin
 SubC := SubC+1;
 Rx := RegisterValue(RdIndex);
 Rx := Rx-RegisterValue(RsIndex);
 Base10To2 (Rx,ThirtyTwoBit);
 For i := 0 to 31 Do RFRdIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure Mul;
Var i,Rx : Integer;
     ThirtyTwoBit : ThirtyTwoBitType;
begin
 MulC := MulC+1;
 Rx := RegisterValue(RdIndex);
 Rx := Rx*RegisterValue(RsIndex);
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 to 31 Do RFRdIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure Udiv;
Var i,Rx : Integer;
     ThirtyTwoBit : ThirtyTwoBitType;
```

```
begin
 DivC := DivC+1;
 Rx := RegisterValue(RxIndex);
 Rx := Rx DIV RegisterValue(RyIndex);
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure Rem;


begin
 RemC := RemC+1;
 PC := PC+2;
end;


Procedure Not;
begin
 NotC := NotC+1;
 For i := 0 to 31 Do
        If R[RxIndex,i] = 1 then R[RxIndex,i] := 0
                               else R[RxIndex,i] := 1;
 PC := PC+2;
end;


Procedure Oor;

begin
 OrC := OrC+1;
 For i := 0 to 31 Do
    If R[RxIndex,i]=0 then If R[RyIndex,i]=0
                            then R[RxIndex,i] := 0
                            else R[RxIndex,i] := 1
                      else   R[Rxindex,i] := 1;
 PC :=PC+2;
end;


Procedure Xor;

begin
 XorC := XorC+1;
 PC := PC+2;
end;


Procedure And;

begin
 AndC := AndC+1;
 For i := 0 to 31 Do
```

```
  If R[RxIndex,i] = 1 Then If R[RyIndex,i] = 1
                              Then R[RxIndex,i] := 1
                              Else R[RxIndex,i] := 0
                      Else R[RxIndex,i] := 0;
 PC := PC+2;
end;


Procedure cbit;

begin
 CbitC := CbitC+1;
 PC := PC+2;
end;



Procedure sbit;

begin
 SbitC := SbitC+1;
 PC := PC+2;
end;



Procedure tbit;

begin
 TbitC := TbitC+1;
 PC := PC+2;
end;


Procedure MoveI;
Var FourBit : FourBitType;
i: Integer;

begin
 MoveIC := MoveIC+1;
 CharToBit (Operands[1],FourBit);
 ClearRegister(RxIndex);
 R[RxIndex,0] := FourBit[0];
 R[RxIndex,1] := FourBit[1];
 R[RxIndex,2] := FourBit[2];
 R[RxIndex,3] := FourBit[3];
 PC := PC+2;
end;


Procedure AddI;
Var x,i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;
begin
 AddIC := AddIC + 1;
 Rx := RegisterValue(RxIndex)+RyIndex;
```

```
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure SubI;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;

begin
 SubIC := SubIC+1;
 Rx := RegisterValue(RxIndex);
 Rx := Rx-RyIndex;
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure MpyI;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;
begin
 MpyIC := MpyIC+1;
 Rx := RegisterValue(RxIndex);
 Rx := Rx*RyIndex;
 Base10To2 (Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 PC := PC+2;
end;


Procedure NotI;
Var FourBit : FourBitType;

begin
 NotIC := NotIC+1;
 ClearRegister(RxIndex);
 CharToBit(Operands[1],FourBit);
 R[RxIndex,0] := FourBit[0];
 R[RxIndex,1] := FourBit[1];
 R[RxIndex,2] := FourBit[2];
 R[RxIndex,3] := FourBit[3];
 For i := 0 To 31 Do
        If R[RxIndex,i]=0 Then R[RxIndex,i] := 1
                           Else R[RxIndex,i] := 0;
 PC := PC+2;
end;


Procedure AndI;
Var ThirtyTwoBit : ThirtyTwoBitType;
    FourBit      : FourBitType;

begin
```

```
 AndIC := AndIC+1;
 For i := 0 To 31 Do ThirtyTwoBit[i] := 0;
 CharToBit(Operands[1],FourBit);
 ThirtyTwoBit[0] := FourBit[0];
 ThirtyTwoBit[1] := FourBit[1];
 ThirtyTwoBit[2] := FourBit[2];
 ThirtyTwoBit[3] := FourBit[3];
 For i := 0 To 31 Do
   If R[RxIndex,i] = 1 Then If ThirtyTwoBit[i]=1
                               Then R[RxIndex,i] := 1
                               Else R[RxIndex,i] := 0
                        Else R[RxIndex,i] := 0;
 PC := PC+2;
end;


Procedure fixt;

begin
 FixtC := FixtC+1;
 PC := PC+2;
end;


Procedure fixr;

begin
 FixrC := FixrC+1;
 PC := PC+2;
end;


Procedure rneg;

begin
 RnegC := RnegC+1;
 PC := PC+2;
end;


Procedure radd;

begin
 RaddC := RaddC+1;
 PC := PC+2
end;


Procedure rsub;

begin
 RsubC := RsubC+1;
 PC := PC+2;
end;
```

```
Procedure rmpy;

begin
 RmpyC := RmpyC+1;
 PC := PC+2;
end;


Procedure rdiv;

begin
 RdivC := RdivC+1;
 PC := PC+2;
end;


Procedure makerd;

begin
 MakerdC := MakerdC+1;
 PC := PC +2;
end;


Procedure lcomp;

begin
 LcompC := LcompC+1;
 PC := PC+2;
end;


Procedure Float;

begin
 FloatC := FloatC+1;
 PC := PC+2;
end;


Procedure Rcomp;
Var i : Integer;
{This procedure should be changed for real arithmetic.}
begin
 RcompC := RcompC+1;
 If RegisterValue(RxIndex)>RegisterValue(RyIndex)
   Then Begin
        For i := 1 to 31 Do R[RxIndex,i] := 0;
        R[RxIndex,0] := 1;
        end
   Else If RegisterValue(RxIndex)=RegisterValue(RyIndex)
            Then For i := 0 to 31 Do R[RxIndex,i] := 0
            Else For i := 0 to 31 Do R[RxIndex,i] := 1;
```

```
 PC := PC+2;
end;


Procedure eadd;

begin
 EaddC := EaddC+1;
 PC := PC+2;
end;


Procedure esub;

begin
 EsubC := EsubC+1;
 PC := PC+2;
end;


Procedure empy;

begin
 EmpyC := EmpyC+1;
 PC := PC+2;
end;


Procedure ediv;

begin
 EdivC := EdivC+1;
 PC := PC+2;
end;


Procedure dfixt;

begin
 DfixtC := DfixtC+1;
 PC := PC+2;
end;


Procedure dfixr;

begin

 DfixrC := DfixrC+1;
 PC := PC+2;
end;


Procedure dneg;

begin
```

```
Oct 19 13:40 1986  fastl.p Page 24

 DrnegC := DrnegC+1;
 PC := PC+2;
end;


Procedure dradd;

begin
 DraddC := DraddC+1;
 PC := PC+2;
end;


Procedure drsub;

begin
 DrsubC := DrsubC+1;
 PC := PC+2;
end;


Procedure drmpy;

begin
 DrmpyC := DrmpyC+1;
 PC := PC+2;
end;


Procedure drdiv;

begin
 DrdivC := DrdivC+1;
 PC := PC+2;
end;


Procedure makedr;

begin
 MakedrC := makedrC+1;
 PC := PC+2;
end;


Procedure dcomp;

begin
 DcompC := DcompC+1;
 PC := PC+2;
end;


Procedure  dfloat;

begin
```

```
 DfloatC := DfloatC+1;
 PC := PC+2;
end;


Procedure drcomp;

begin
 DrcompC := DrcompC+1;
 PC := PC+2;
end;


Procedure trans;

begin
 TransC := TransC+1;
 PC := PC+2;
end;


Procedure dirt;

begin
 DirtC := DirtC+1;
 PC := PC+2;
end;


Procedure sus;

begin
 SusC := SusC+1;
 PC := PC+2;
end;

Procedure lus;

begin
 LusC := LusC+1;
 PC := PC+2;
end;

Procedure rum;

begin
 RumC := RumC+1;
 PC := PC+2;
end;

Procedure read;
begin
 ReadC := ReadC+1;
 PC := PC+2;
end;
```

```
Procedure write;

begin
 WriteC := WriteC+1;
 PC := PC+2;
end;
Procedure  TestGT;

begin
 TestGTC := TestGTC+1;
 If RegisterValue(RxIndex) > RegisterValue(RyIndex)
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister (RxIndex);
 PC :=PC+2;
end;


Procedure TestLT;

begin
 TestLTC := TestLTC+1;
 If RegisterValue(RxIndex) < RegisterValue(RyIndex)
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1
             end
        else ClearRegister (RxIndex);
 PC := PC+2;
end;


Procedure TestEQ;

begin
 TestEQC := TestEQC+1;
 If RegisterValue(RxIndex) = RegisterValue(RyIndex)
        then begin
                ClearRegister (RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure CallR;
Var
        Rx,i : Integer;
        BinaryRx : ThirtyTwoBitType;

begin
 CallrC := CallrC+1;
```

```
IF STATIC = TRUE THEN PC := PC+2
ELSE
BEGIN
 Rx := PC+2;
 PC := PC+RegisterValue(RyIndex);
 Base10To2 (Rx,BinaryRx);
 For i := 0 to 31 Do R[RxIndex,i] := BinaryRx[i];
END;
end;


Procedure TestIGT;

begin
 TestIGTC := TestIGTC+1;
 If RegisterValue(RxIndex) > RyIndex
        then begin
                ClearRegister (RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister (RxIndex);
 PC := PC+2;
end;


Procedure TestILT;

begin
 TestILTC := TestILTC+1;
 If RegisterValue(RxIndex) < RyIndex
        then begin
                ClearRegister (RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure TestIEQ;

begin
 TestIEQC := testIEQC+1;
 If RegisterValue(RxIndex) = RyIndex
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure Ret;
Var i : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;
```

```
begin
 RetC := RetC+1;
 IF STATIC = TRUE THEN PC := PC+2
 ELSE
 BEGIN
 i := PC+2;
 Base10To2(i,ThirtyTwoBit);
 PC := RegisterValue(RyIndex);
 For i := 0 to 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 END
end;


Procedure TestLE;

begin
 TestLEC := TestLEC+1;
 If RegisterValue(RxIndex) <= RegisterValue(RyIndex)
        then begin
               ClearRegister (RxIndex);
               R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure TestGE;

begin
 TestGEC := TestGEC+1;
 IF RegisterValue(RxIndex) >= RegisterValue(RyIndex)
        then begin
               ClearRegister (RxIndex);
               R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure TestNE;

begin
 TestNEC := TestNEC+1;
 If RegisterValue(RxIndex) <> RegisterValue(RyIndex)
        then begin
               ClearRegister (RxIndex);
               R[RxIndex,0] := 1;
             end
        else ClearRegister (RxIndex);
 PC := PC+2;
end;
```

```
Procedure TestILE;

begin
 TestILEC := TestILEC+1;
 If RegisterValue(RxIndex) <= RyIndex
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure TestIGE;

begin
 TestIGEC := TestIGEC+1;
 If RegisterValue(RxIndex) >= RyIndex
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure TestINE;

begin
 TestINEC := TestINEC+1;
 If RegisterValue(RxIndex) <> RyIndex
        then begin
                ClearRegister(RxIndex);
                R[RxIndex,0] := 1;
             end
        else ClearRegister(RxIndex);
 PC := PC+2;
end;


Procedure Lsl;
Var n : Integer;

begin
 LslC := LslC+1;
 n := RegisterValue(RyIndex);
 LogicalShiftLeft(RxIndex,n);
 PC := PC+2;
end;


Procedure lsr;

begin
```

```
 LsrC := LsrC+1;
 PC := PC+2;
end;


Procedure asl;

begin
 AslC := AslC+1;
 PC := PC+2;
end;


Procedure Asr;
Var n : Integer;

begin
 AsrC := AsrC+1;
 n := RegisterValue(RyIndex);
 ArithmeticShiftRight(RxIndex,n);
 PC := PC+2;
end;


Procedure dlsl;

begin
 DlslC := DlslC+1;
 PC := PC+2;
end;


Procedure dlsr;

begin
 DlsrC := DlsrC+1;
 PC := PC+2;
end;


Procedure csl;

begin
 CslC := CslC+1;
 PC := PC+2;
end;


Procedure seb;

begin
 SebC := SebC+1;
 PC := PC+2;
end;
```

```
Procedure LslI;

begin
 LslIC := LslIC+1;
 LogicalShiftLeft(RxIndex,RyIndex);
 PC := PC+2;
end;


Procedure lsri;

begin
 LsrIC := LsrIC+1;
 PC := PC+2;
end;


Procedure asli;

begin
 AslIC := AslIC+1;
 PC := PC+2;
end;


Procedure AsrI;

begin
 AsrIC := AsrIC+1;
 ArithmeticShiftRight(RxIndex,RyIndex);
 PC := PC+2;
end;


Procedure dlsli;

begin
 DlslIC := DlslIC+1;
 PC := PC+2;
end;


Procedure dlsri;

begin
 DlsrIC := DlsrIC+1;
 PC := PC+2;
end;


Procedure csli;

begin
 CslIC := CslIC+1;
 PC := PC+2;
end;
```

```
Procedure  seh;

begin
 SehC := SehC+1;
 PC := PC+2;
end;


Procedure BrGTS;
Var
Rem : Integer;

begin
 BrGTSC := BrGTSC+1;
 IF STATIC = TRUE THEN PC := PC+4
 ELSE
 BEGIN
 GetShortAdd;
 Rem := ABS(EA) MOD 2;
 If Rem = 1 Then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) > RegisterValue(RyIndex)
        Then    PC := PC+EA
        Else    PC := PC+4;
 END;
end;


Procedure BrGTL;
Var EA,Rem : Integer;

begin
 BrGTLC := BrGTLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) > RegisterValue(RyIndex)
        Then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrEQS;
Var Rem : Integer;

begin
 BrEQSC := BrEQSC+1;
 IF STATIC = TRUE THEN PC := PC+4
```

```
 ELSE
 BEGIN
 GetShortAdd;
 Rem := ABS(EA) MOD 2;
 If Rem = 1 then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex)  = RegisterValue(RyIndex)
        then    PC := PC+EA
        else
             PC := PC+4;
 END;
end;


Procedure BrEQL;
Var Rem : Integer;

begin
 BrEQLC := BrEQLC+1;
 IF STATIC = TRUE THEN PC := PC+6
 ELSE
 BEGIN
 GetLongAdd;
 Rem := ABS(EA) MOD 2;
 If Rem = 1 Then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) = RegisterValue(RyIndex)
        then    PC := PC+EA
        else
             PC := PC+6;
 END;
end;


Procedure CallS;
Var
        i,Rx : Integer;
        Out  : ThirtyTwoBitType;

begin
 CallSC := CallSC+1;
 IF STATIC = TRUE THEN PC := PC+4
 ELSE
 BEGIN
 Rx := PC+4;
 Base10To2 (Rx,out);
 For i := 0 to 31 Do R[RxIndex,i] := Out[i];
 GetShortAdd;
 If EA < 0 Then EA := EA-1;
 PC := PC+EA;
 TotalCall := TotalCall+ABS(EA);
 END;
end;


Procedure CallL;
```

```
Var
        i,Rx : Integer;
        Out  : ThirtyTwoBitType;
        ADD  : ByteType;
begin
 CallLC := CallLC+1;
 IF STATIC = TRUE THEN PC := PC+6
 ELSE
 BEGIN
 Rx := PC + 6;
{To skip the RTL call.}
Add := CodeMemory[PC+2];
If Add[1] = 'F' Then PC := PC+6
Else begin
     Base10To2 (Rx,Out);
     For i := 0 To 31 Do R[RxIndex,i] := Out[i];
     GetLongAdd;
     If EA < 0 Then EA := EA-1;
     PC := PC+EA;
     TotalCall := TotalCall+ABS(EA);
     end;
END;
end;


Procedure BrIGtS;
var Rem : Integer;

begin
 BrIGTSC := BrIGTSC+1;
 IF STATIC = TRUE THEN PC:= PC+4
 ELSE
 BEGIN
 GetShortAdd;
 Rem := ABS(EA) MOD 2;
 If Rem = 1 then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) > RyIndex
        then    PC := PC+EA
        else
               PC := PC+4;
 END;
end;


Procedure BrIGtL;
Var Rem : Integer;

begin
 BrIGTLC := BrIGTLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
```

```
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) > RyIndex
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrILtS;
Var Rem : Integer;

begin
 BrILTSC := BrILTSC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) < RyIndex
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrILtL;
Var Rem : Integer;

begin
 BrILTLC := BrILTLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) < RyIndex
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrIEqS;
Var Rem : Integer;

begin
 BrIEQSC := BrIEQSC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
```

Oct 19 13:40 1986 fast1.p Page 36

```
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) = RyIndex
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrIEqL;
Var Rem : Integer;

begin
 BrIEQLC := BrIEQLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) = RyIndex
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure LoopS;
Var
        i,Rx,Rem : Integer;
        BinaryRx : ThirtyTwoBitType;

begin
 LoopSC := LoopSC+1;
 IF STATIC = TRUE THEN PC := PC+4
 ELSE
 BEGIN
 Rx := RegisterValue(RxIndex) + RyIndex;
 Base10To2 (Rx,BinaryRx);
 For i := 0 to 31 Do R[Rxindex,i] := BinaryRx[i];
 GetShortAdd;
 Rem := ABS(EA) MOD 2;
 IF Rem = 1  then  EA := EA-1;
 If Rx < 0 then PC := PC+EA
           else PC := PC+4;
 TotalLoop := TotalLoop+ABS(EA);
 END;
end;
```

```
Procedure LoopL;
Var
        i,Rx,Rem : Integer;
        BinaryRx : ThirtyTwoBitType;

begin
 LoopLC := LoopLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
 Rx := RegisterValue(RxIndex)+RyIndex;
 Base10To2 (Rx,BinaryRx);
 For i := 0 to 31 Do R[RxIndex,i] := BinaryRx[i];
 GetLongAdd;
 Rem := ABS(EA) MOD 2;
 If Rem = 1  Then EA := EA-1;
 If Rx <> 0 then PC := PC+EA
           else PC := PC+6;
 TotalLoop := TotalLoop+ABS(EA);
END;
end;


Procedure BrLeS;
Var Rem : Integer;

begin
 BrLESC := BrLESC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <= RegisterValue(RyIndex)
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrLeL;
Var AbsEA,Rem : Integer;

begin
 BrLELC := BrLELC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <= RegisterValue(RyIndex)
```

```
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrNeS;
Var Rem : Integer;
x,y : Integer;

begin
 BrNESC := BrNESC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <> RegisterValue(RyIndex)
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrNeL;
Var Rem : Integer;

begin
 BrNELC := BrNELC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <> RegisterValue (RyIndex)
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrS;

begin
 BrSC := BrSC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
 GetShortAdd;
```

```
 If EA < 0 Then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 PC := PC+EA;
END;
end;


Procedure BrL;

begin
 BrLC := BrLC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
 GetLongAdd;
 If EA < 0 Then EA := EA-1;
 TotalBr := TotalBr+ABS(EA);
 PC := PC+EA;
END;
end;


Procedure BrIleS;
Var Rem : Integer;

begin
 BrILESC := BrILESC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <= RyIndex
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrIleL;
Var Rem : Integer;

begin
 BrILELC := BrILELC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <= RyIndex
        then PC := PC+EA
```

```
        else
             PC := PC+6;
END;
end;


Procedure BrIGeS;
Var AbsEA,Rem : Integer;

begin
 BrIGESC := BrIGESC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) >= RyIndex
        then PC := PC+EA
        else
             PC :=PC+4;
END;
end;


Procedure BrIGeL;
Var Rem : Integer;

begin
 BrIGELC := BrIGELC+1;
 IF STATIC = TRUE THEN PC := PC+6
ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) >= RyIndex
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure BrINeS;
Var Rem : Integer;

begin
 BrINESC := BrINESC+1;
 IF STATIC = TRUE THEN PC := PC+4
ELSE
BEGIN
GetShortAdd;
Rem := ABS(EA) MOD 2;
```

```
If Rem = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
 If RegisterValue(RxIndex) <> RyIndex
        then PC := PC+EA
        else
             PC := PC+4;
END;
end;


Procedure BrINeL;
Var Rem : Integer;

begin
 BrINELC := BrINELC+1;
 IF STATIC = TRUE THEN PC := PC+6
 ELSE
BEGIN
GetLongAdd;
Rem := ABS(EA) MOD 2;
If REm = 1 then EA := EA-1;
TotalBr := TotalBr+ABS(EA);
If RegisterValue(RxIndex) <> RyIndex
        then PC := PC+EA
        else
             PC := PC+6;
END;
end;


Procedure StorebS;
Var TempByte : ByteType;

begin
 StorebSC := StorebSC+1;
 IF STATIC = FALSE THEN
BEGIN
 GetByte (RxIndex,TempByte);
 GetShortAdd;
 DataMemory[EA] := TempByte;
END;
 PC := PC+4;
end;


Procedure StorebL;
Var TempByte : ByteType;

begin
 StorebLC := StorebLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetByte (RxIndex,TempByte);
 GetLongAdd;
 DataMemory[EA] := TempByte;
 END;
```

```
 PC := PC+6;
end;


Procedure StorebIS;
Var TempByte : ByteType;
begin
 StorebISC := StorebISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetByte (RxIndex,TempByte);
 GetShortAdd;
 EA := RegisterValue(RyIndex)+EA;
 DataMemory[EA] := TempByte;
 END;
 PC := PC+4;
end;


Procedure StorebIL;
Var TempByte : ByteType;

begin
 StorebILC := StorebILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetByte (RxIndex,TempByte);
 GetLongAdd;
 EA := RegisterValue(Ryindex)+EA;
 DataMemory[EA] := TempByte;
 END;
 PC := PC+6;
end;


Procedure StorehS;
Var TempByte : ByteType;

begin
 StorehSC := StorehSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetHalfWord (RxIndex,TempHalfWord);
 GetShortAdd;
 DataMemory[EA] := TempHalfWord[0];
 DataMemory[EA+1] := TempHalfWord[1];
 END;
 PC := PC+4;
end;


Procedure StorehL;
Var TempHalfWord : HalfWordType;

begin
```

```
 StorehLC := StorehLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetHalfWord (RxIndex,TempHalfWord);
 GetLongAdd;
 DataMemory[EA] := TempHalfWord[0];
 DataMemory[EA+1] := TempHalfWord[1];
 END;
 PC := PC+6;
end;

Procedure StoreHIS;
Var TempHalfWord : HalfWordType;

begin
 StorehISC:= StorehISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetHalfWord(RxIndex,TempHalfWord);
 GetShortAdd;
 EA := RegisterValue(RyIndex)+EA;
 DataMemory[EA] := TempHalfWord[0];
 DataMemory[EA+1] := TempHalfWord[1];
 END;
 PC := PC+4;
end;


Procedure StoreHIL;
Var TempHalfWord : HalfWordType;

begin
 StorehILC := StorehILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetHalfWord (RxIndex,TempHalfWord);
 GetLongAdd;
 EA := RegisterValue(RyIndex)+EA;
 DataMemory[EA] := TempHalfWord[0];
 DataMemory[EA+1] := TempHalfWord[1];
 END;
 PC := PC+6;
end;


Procedure StoreS;
Var TempWord : WordType;

begin
 StoreSC := StoreSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetWord(RxIndex,TempWord);
 GetShortAdd;
 DataMemory[EA] := TempWord[0];
 DataMemory[EA+1] := TempWord[1];
```

```
 DataMemory[EA+2] := TempWord[2];
 DataMemory[EA+3] := TempWord[3];
 END;
 PC := PC+4;
end;


Procedure StoreL;
Var TempWord : WordType;

begin
 StoreLC := StoreLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetWord(RxIndex,TempWord);
 GetLongAdd;
 DataMemory[EA] := TempWord[0];
 DataMemory[EA+1] := TempWord[1];
 DataMemory[EA+2] := TempWord[2];
 DataMemory[EA+3] := TempWord[3];
 END;
 PC := PC+6;
end;


Procedure StoreIS;
Var TempWord : WordType;
    x : Integer;
begin
 StoreISC := StoreISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetWord(RxIndex,TempWord);
 GetShortAdd;
 EA := RegisterValue(RyIndex)+EA;
 DataMemory[EA] := TempWord[0];
 DataMemory[EA+1] := TempWord[1];
 DataMemory[EA+2] := TempWord[2];
 DataMemory[EA+3] := TempWord[3];
 END;
 PC := PC+4;
end;


Procedure StoreIL;
Var TempWord : WordType;

begin
 StoreILC := StoreILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetWord(RxIndex,TempWord);
 GetLongAdd;
 EA := RegisterValue (RyIndex)+EA;
 DataMemory[EA] := TempWord[0];
 DataMemory[EA+1] := TempWord[1];
```

```
 DataMemory[EA+2] := TempWord[2];
 DataMemory[EA+3] := TempWord[3];
 END;
 PC := PC+6;
end;


Procedure storedS;
begin
 StoredSC := StoredSC+1;
 PC := PC+4;
end;

Procedure StoredL;
begin

 StoredLC := StoredLC+1;
 PC := PC+6;
end;


Procedure StoredIS;
begin
 StoredISC := StoredISC+1;
 PC := PC+4;
end;

Procedure StoredIL;
begin
 StoredILC := StoredILC+1;
 PC := PC+6;
end;


Procedure LoadbS;
Var TempByte : ByteType;

begin
 LoadbSC := LoadbSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 TempByte := DataMemory[EA];
 LoadByte (TempByte,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadbL;
Var TempByte : ByteType;

begin
 loadbLC := LoadbLC+1;
 IF STATIC = FALSE THEN
```

```
 BEGIN
 GetLongAdd;
 TempByte := DataMemory[EA];
 LoadByte (TempByte,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadbIS;
Var TempByte : ByteType;

begin
 LoadbISC := LoadbISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 EA := EA+RegisterValue(RyIndex);
 TempByte := DataMemory[EA];
 LoadByte (TempByte,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadbIL;
Var TempByte : ByteType;

begin
 LoadbILC := LoadbILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 EA := EA+RegisterValue(RyIndex);
 TempByte := DataMemory[EA];
 LoadByte (TempByte,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadhS;
begin
 LoadhSC := LoadhSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 TempHalfWord[0] := DataMemory[EA];
 TempHalfWord[1] := DataMemory[EA+1];
 LoadHalfWord (TempHalfWord,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadhL;
```

```
begin
 LoadhLC := LoadhLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 TempHalfWord[0] := DataMemory[EA];
 TempHalfWord[1] := DataMemory[EA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadhIS;
begin
 LoadhISC := LoadhISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 EA := EA+RegisterValue(RyIndex);
 TempHalfWord[0] := DataMemory[EA];
 TempHalfWord[1] := DataMemory[EA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadhIL;
begin
 LoadhILC := LoadhILC+1;
 IF STATIC = FALSE THEN
 BEGIN  .
 GetLongAdd;
 EA := EA+RegisterValue(RyIndex);
 TempHalfWord[0] := DataMemory[EA];
 TempHalfWord[1] := DataMemory[EA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadS;
begin
 LoadSC := LoadSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 Tempword[0] := DataMemory[EA];
 TempWord[1] := DataMemory[EA+1];
 TempWord[2] := DataMemory[EA+2];
 TempWord[3] := DataMemory[EA+3];
 LoadWord (TempWord,RxIndex);
 END;
 PC := PC+4;
```

```
end;


Procedure LoadL;
begin
 LoadLC := LoadLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 TempWord[0] := DataMemory[EA];
 TempWord[1] := DataMemory[EA+1];
 TempWord[2] := DataMemory[EA+2];
 TempWord[3] := DataMemory[EA+3];
 LoadWord (TempWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadIS;
Var TempWord : WordType;

begin
 LoadISC := LoadISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 EA := EA+RegisterValue(RyIndex);
 TempWord[0] := DataMemory[EA];
 TempWord[1] := DataMemory[EA+1];
 TempWord[2] := DataMemory[EA+2];
 TempWord[3] := DataMemory[EA+3];
 LoadWord(TempWord,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadIL;
begin
 LoadILC := LoadILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 EA := EA+RegisterValue(RyIndex);
 TempWord[0] := DataMemory[EA];
 TempWord[1] := DataMemory[EA+1];
 TempWord[2] := DataMemory[EA+2];
 TempWord[3] := DataMemory[EA+3];
 LoadWord(TempWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadDS;
```

```
begin
 LoaddSC := LoaddSC+1;
 PC := PC+4;
end;


Procedure LoaddL;
begin
 LoaddLC := LoaddLC+1;
 PC := PC+6;
end;


Procedure LoaddIS;
begin
 LoaddISC := LoaddISC+1;
 PC := PC +4;
end;


Procedure LoaddIL;
begin
 LoaddILC := LoaddILC+1;
 PC := PC+6;
end;


Procedure LaddrS;
Var      i : Integer;
         ThirtyTwoBit : ThirtyTwoBitType;
begin
 LaddrSC := LaddrSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 Base10To2(EA,ThirtyTwoBit);
 For i := 0 to 31 Do  R[Rx.Index,i] := ThirtyTwoBit[i];
 END;
 PC := PC+4;
end;


Procedure LaddrL;
Var      i : Integer;
         ThirtyTwoBit : ThirtyTwoBitType;

begin
 LaddrLC := LaddrLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 Base10To2(EA,ThirtyTwoBit);
 For i := 0 to 31 Do R[Rx.Index,i] := ThirtyTwoBit[i];
 END;
 PC := PC+6;
end;
```

```
Procedure LaddrIS;
Var i,x : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;
begin
 LaddrISC := LaddrISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 X := EA+RegisterValue(RyIndex);
 Base10To2(X,ThirtyTwoBit);
 For i := 0 to 31 Do RER[xIndex,i] := ThirtyTwoBit[i];
 END;
 PC := PC+4;
end;


Procedure LaddrIL;
Var  i,x : Integer;
     ThirtyTwoBit : ThirtyTwoBitType;
begin
 LaddrILC := LaddrILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 X := EA+RegisterValue(RyIndex);
 Base10To2(X,ThirtyTwoBit);
 For i := 0 to 31 Do RER[xIndex,i] := ThirtyTwoBit[i];
 END;
 PC := PC+6
end;


Procedure LoadbpS;
Var CodeEA : Integer;

begin
 LoadbpSC := LoadbpSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 CodeEA := PC+EA;
 TempByte := CodeMemory[CodeEA];
 LoadByte(TempByte,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadbpL;
Var CodeEA : Integer;

begin
 LoadbpLC := LoadbpLC+1;
 IF STATIC = FALSE THEN
```

```
 BEGIN
 GetLongAdd;
 CodeEA := PC +EA;
 TempByte := CodeMemory[CodeEA];
 LoadByte(TempByte,R::Index);
 END;
 PC := PC+6;
end;


Procedure LoadbpIS;
Var CodeEA : Integer;

begin
 LoadbpISC := LoadbpISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 CodeEA := PC+EA+RegisterValue(RyIndex);
 TempByte := CodeMemory[CodeEA];
 LoadByte(TempByte,R::Index);
 END;
 PC := PC+4;
end;


Procedure LoadbpIL;
Var CodeEA : Integer;

begin
 LoadbpILC := LoadbpILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 CodeEA := PC+EA+RegisterValue(RyIndex);
 TempByte := CodeMemory[CodeEA];
 LoadByte(TempByte,R::Index);
 END;
 PC := PC+6;
end;


Procedure LoadhpS;
Var CodeEA : Integer;

begin
 LoadhpSC := LoadhpSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 CodeEA := PC+EA;
 TempHalfWord[0] := CodeMemory[CodeEA];
 TempHalfWord[1] := CodeMemory[CodeEA+1];
 LoadHalfWord(TempHalfWord,R::Index);
 END;
 PC := PC+4;
```

```
end;


Procedure LoadhwL;
Var CodeEA : Integer;

begin
 LoadhwLC := LoadhwLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 CodeEA := PC+EA;
 TempHalfWord[0] := CodeMemory[CodeEA];
 TempHalfWord[1] := CodeMemory[CodeEA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadhwIS;
Var CodeEA : Integer;

begin
 LoadhwISC := LoadhwISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 CodeEA := PC+EA+RegisterValue(RyIndex);
 TempHalfWord[0] := CodeMemory[CodeEA];
 TempHalfWord[1] := CodeMemory[CodeEA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadhwIL;
Var CodeEA : Integer;

begin
 LoadhwILC := LoadhwILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 CodeEA := PC+EA+RegisterValue(RyIndex);
 TempHalfWord[0] := CodeMemory[CodeEA];
 TempHalfWord[1] := CodeMemory[CodeEA+1];
 LoadHalfWord(TempHalfWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadwS;
Var CodeEA : Integer;
```

```
begin
 LoadpSC := LoadpSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 CodeEA := PC+EA;
 TempWord[0] := CodeMemory[CodeEA];
 TempWord[1] := CodeMemory[CodeEA+1];
 TempWord[2] := CodeMemory[codeEA+2];
 TempWord[3] := CodeMemory[CodeEA+3];
 LoadWord(TempWord,RxIndex);
 END;
 PC := PC+4;
end;


Procedure LoadpL;
Var CodeEA : Integer;

begin
 LoadpLC := LoadpLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 CodeEA := PC+EA;
 TempWord[0] := CodeMemory[CodeEA];
 TempWord[1] := CodeMemory[CodeEA+1];
 TempWord[2] := CodeMemory[CodeEA+2];
 TempWord[3] := CodeMemory[CodeEA+3];
 LoadWord(TempWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoadpIS;
Var CodeEA : Integer;

begin
 LoadpISC := LoadpISC+1;
{ GetShortAdd;}
 {CodeEA := PC+EA+RegisterValue(RyIndex);}
 {TempWord[0] := CodeMemory[CodeEA];}
 {TempWord[1] := CodeMemory[CodeEA+1];}
 {TempWord[2] := CodeMemory[CodeEA+2];}
 {TempWord[3] := CodeMemory[CodeEA+3];}
 {LoadWord(TempWord,RxIndex);}
 PC := PC+4;
end;


Procedure LoadpIL;
Var CodeEA : Integer;

begin
```

```
 LoadpILC := LoadpILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 CodeEA := PC+EA+RegisterValue(RyIndex);
 TempWord[0] := CodeMemory[CodeEA];
 TempWord[1] := CodeMemory[CodeEA+1];
 TempWord[2] := CodeMemory[CodeEA+2];
 TempWOrd[3] := CodeMemory[CodeEA+3];
 LoadWord(TempWord,RxIndex);
 END;
 PC := PC+6;
end;


Procedure LoaddpS;
begin
 LoaddpSC := LoaddpSC+1;
 PC := PC+4;
end;


Procedure LoaddpL;
begin
 LoaddpLC := LoaddpLC+1;
 PC := PC+6;
end;


Procedure LoaddpIS;
begin
LoaddpISC := LoaddpISC+1;
PC := PC+4;
end;


Procedure LoaddpIL;
begin
LoaddpILC := LoaddpILC+1;
PC := PC+6;
end;

Procedure LaddrpS;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;

begin
 LaddrpSC := LaddrpSC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 Rx := PC+EA;
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 To 31 Do R[RxIndex,i] := ThirtyTwoBit[i];
 END;
 PC := PC+4;
```

```
end;


Procedure LaddrpL;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;

begin
 LaddrpLC := LaddrpLC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 Rx := PC+EA;
 Base10To2 (Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[Rx:Index,i] := ThirtyTwoBit[i];
 END;
 PC := PC+6;
end;


Procedure LaddrpIS;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;

begin
 LaddrpISC := LaddrpISC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetShortAdd;
 Rx := PC+EA+RegisterValue(RyIndex);
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 To 31 Do R[Rx.Index,i] := ThirtyTwoBit[i];
 END;
 PC :=PC+4
end;


Procedure LaddrpIL;
Var i,Rx : Integer;
    ThirtyTwoBit : ThirtyTwoBitType;

begin
 LaddrpILC := LaddrpILC+1;
 IF STATIC = FALSE THEN
 BEGIN
 GetLongAdd;
 Rx := PC+EA+RegisterValue(RyIndex);
 Base10To2(Rx,ThirtyTwoBit);
 For i := 0 to 31 Do R[Rx:Index,i] := ThirtyTwoBit[i];
 END;
 PC := PC+6
end;


(**********************************************)
```

```
(* Procedure to start simulation of Ridge-32.*)
(**********************************************)

Procedure Execution;
Var
     i : Integer;

begin
 Writeln ('* Start of Execution !!.');
 For i := 0 to MaxDataMemory Do DataMemory[i] := '00';
 Writeln ('* Do you want Static data ? (y/n)');
 Readln  (TempChar);
 If TempChar = 'y' Then Static := True
                Else Static := False;
 Writeln ('* Starting PC ? : ');
 Readln (PC);
 Error := FALSE;
 While PC <= PcMax Do
  Begin
   If Error = TRUE Then Begin
                        Writeln ('* Do you want to continue run ? (y/n):');
                        Readln (TempChar);
                        If TempChar = 'y' Then Begin
                                               Writeln ('* Continuing PC ? : ');
                                               Readln (PC);
                                               Error := FALSE;
                                              end{then}
                                          Else PC := PCMax+1;
                        End{then}
                   Else
   BEGIN{ELSE}

   OpCode := CodeMemory[PC];
IF CodeMemory[25] = '00'
     THEN Writeln('@@PC:',PC:5,'** OpCode**',OpCode:2);
   Operands := CodeMemory[PC+1];
   RxIndex := Value(Operands[0]);
   RyIndex := Value(Operands[1]);
   Case OpCode[0] of
   '0' : Case OpCode[1] of
         '1' : Move;
         '2' : Neg;
         '3' : Add;
         '4' : Sub;
         '5' : Mpy;
         '6' : Ddiv;
         '7' : Rem;
         '8' : Nnot;
         '9' : Oor;
         'A' : Xor;
         'B' : Aand;
         'C' : Cbit;
         'D' : Sbit;
         'E' : Tbit;
         Otherwise : begin
                        Error := true;
```

```
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                      end;
           end;
  '1' :Case OpCode[1] of
           '1' : MoveI;
           '3' : AddI;
           '4' : SubI;
           '5' : MpyI;
           '8' : NotI;
           'B' : AndI;
           Otherwise : begin
                         Error := true;
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                      end;
          end;
   '2' : Case OpCode[1] of
           '0' : Fixl;
           '1' : Fixr;
           '2' : Rneg;
           '3' : Radd;
           '4' : Rsub;
           '5' : Rmpy;
           '6' : Rdiv;
           '7' : Makerd;
           '8' : Lcomp;
           '9' : Float;
           'A' : Rcomp;
           'C' : Eadd;
           'D' : Esub;
           'E' : Empy;
           'F' : Ediv;
           Otherwise : begin
                         Error := true;
                         writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                      end;
           end;
   '3' : Case OpCode[1] of
           '0' : Dfixl;
           '1' : Dfixr;
           '2' : Drneg;
           '3' : Dradd;
           '4' : Drsub;
           '5' : Drmpy;
           '6' : Drdiv;
           '7' : Makedr;
           '8' : Dcomp;
           '9' : Dfloat;
           'A' : Drcomp;
           Otherwise : begin
                         PC := PCMax+1;(*Quit Execution*)
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                      end;
           end;
   '4' : begin
                 Error := true;
                 Writeln ('* Exceptional Code: ',OpCode, 'at PC',PC:6);
```

```
        end;
    '5' : Case OpCode[1] of
        '0' : TestGT;
        '1' : TestLT;
        '2' : TestEQ;
        '3' : CallR;
        '4' : TestIGT;
        '5' : TestILT;
        '6' : TestIEQ;
        '7' : Ret;
        '8' : TestLE;
        '9' : TestGE;
        'A' : TestNE;
        'C' : TestILE;
        'D' : TestIGE;
        'E' : TestINE;
        Otherwise : begin
                        Error := true;
                        Writeln ('* Exceptional Code: ',OpCode,' at PC',PC:6);
                    end;
        end;
    '6' : Case OpCode[1] of
        '0' : Lsl;
        '1' : Lsr;
        '2' : Asl;
        '3' : Asr;
        '4' : Dlsl;
        '5' : Dlsr;
        '8' : Csl;
        'A' : Seb;
        Otherwise : begin
                        Error := true;
                        Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                    end;
        end;
    '7' : Case OpCode[1] of
        '0' : LslI;
        '1' : LsrI;
        '2' : AslI;
        '3' : AsrI;
        '4' : DlslI;
        '5' : DlsrI;
        '8' : CslI;
        'A' : Seb;
        Otherwise : begin
                        Error := true;
                        Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                    end;
        end;
    '8' : Case OpCode[1] of
        '0' : BrGTS;
        '2' : BrEQS;
        '3' : CallS;
        '4' : BrIGTS;
        '5' : BrILTS;
        '6' : BrIEQS;
```

```
      '7' : LoopS;
      '8' : BrLtS;
      'A' : BrNeS;
      'B' : BrS;
      'C' : BrLtES;
      'D' : BrLtES;
      'E' : BrLtES;
      Otherwise : begin
                      Error := true;
                      Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                  end;
      end;
  '9' : Case OpCode[1] of
      '0' : [illegible];
      '2' : [illegible];
      '3' : [illegible];
      '4' : [illegible];
      '5' : [illegible];
      '6' : [illegible];
      '7' : [illegible];
      '8' : [illegible];
      'A' : [illegible];
      'B' : [illegible];
      'C' : [illegible];
      'D' : [illegible];
      'E' : [illegible];
      Otherwise : begin
                      Error := true;
                      Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                  end;
      end;
  'A' : Case OpCode[1] of
      '0' : StoreS;
      '1' : StoreHS;
      '2' : StoreS;
      '3' : StoreHS;
      '6' : StoreS;
      '7' : StoreS;
      '8' : StoreS;
      '9' : StoreHS;
      Otherwise : begin
                      Error := true;
                      Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                  end;
      end;
  'B' : Case OpCode[1] of
      '0' : StoreH;
      '1' : StoreHL;
      '2' : StoreH;
      '3' : StoreHL;
      '6' : StoreL;
      '7' : StoreLL;
      '8' : StoreDL;
      '9' : StoreHL;
      Otherwise : begin
                      Error := true;
```

```
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                       end;
        end;
   'C' : Case OpCode[1] of
        '0' : LoadbS;
        '1' : LoadbIS;
        '2' : LoadhS;
        '3' : LoadhIS;
        '6' : LoadS;
        '7' : LoadIS;
        '8' : LoaddS;
        '9' : LoaddIS;
        'E' : LaddrS;
        'F' : LaddrIS;
        Otherwise : begin
                         Error := true;
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                       end;
        end;
   'D' : Case OpCode[1] of
        '0' : LoadbL;
        '1' : LoadbIL;
        '2' : LoadhL;
        '3' : LoadhIL;
        '6' : LoadL;
        '7' : LoadIL;
        '8' : LoaddL;
        '9' : LoaddIL;
        'E' : LaddrL;
        'F' : LaddrIL;
        Otherwise : begin
                         Error := true;
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                       end;
        end;
   'E' : Case OpCode[1] of
        '0' : LoadbpS;
        '1' : LoadbpIS;
        '2' : LoadhpS;
        '3' : LoadhpIS;
        '6' : LoadpS;
        '7' : LoadpIS;
        '8' : LoaddpS;
        '9' : LoaddpIS;
        'E' : LaddrpS;
        'F' : LaddrpIS;
        Otherwise : begin
                         Error := true;
                         Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                       end;
        end;
   'F' : Case OpCode[1] of
        '0' : LoadbpL;
        '1' : LoadbpIL;
        '2' : LoadhpL;
        '3' : LoadhpIL;
```

```
         '6' : LoadpL;
         '7' : LoadpIL;
         '8' : LoaddpL;
         '9' : LoaddpIL;
         'E' : LaddrpL;
         'F' : LaddrpIL;
         Otherwise : begin
                          Error := true;
                          Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                      end;
         end;
   Otherwise : begin
                    Error := true;
                    Writeln ('* Exceptional Code : ',OpCode,' at PC',PC:6);
                end;
  end;{CASE}
 END;{ELSE}
 end;{While}


writeln ('*** End of Execution !! ***');

end; (*Execution*)


(****************************************************************)
(* Procedure to calulate analysis data and print out result. *)
(****************************************************************)
Procedure PrintStatistics;

Var
T : Integer;

begin

LoadbT    := LoadbSC+LoadbLC+LoadbISC+LoadbILC;
LoadhT    := LoadhSC+LoadhLC+LoadhISC+LoadhILC;
LoadT     := LoadSC+LoadLC+LoadISC+LoadILC;
LoaddT    := LoaddSC+LoaddLC+LoaddISC+LoaddILC;
LaddrT    := LaddrSC+LaddrLC+LaddrISC+LaddrILC;
LoadbpT   := LoadbpSC+LoadbpLC+LoadbpISC+LoadbpILC;
LoadhpT   := LoadhpSC+LoadhpLC+LoadhpISC+LoadhpILC;
LoadpT    := LoadpSC+LoadpLC+LoadpISC+LoadpILC;
LoaddpT   := LoaddpSC+LoaddpLC+LoaddpISC+LoaddpILC;
LaddrpT   := LaddrpSC+LaddrpLC+LaddrpISC+LaddrpILC;

StorebT   := StorebSC+StorebLC+StorebISC+StorebILC;
StorehT   := StorehSC+StorehLC+StorehISC+StorehILC;
StoreT    := StoreSC+StoreLC+StoreISC+StoreILC;
StoredT   := StoredSC+StoredLC+StoredISC+StoredILC;

AddT      := AddC+AddIC;
MpyT      := MpyC+MpyIC;
SubT      := SubC+SubIC;
```

```
AndT      := AndC+AndIC;
NotT      := NotC+NotIC;

LslT     := LslC+LslIC;
LsrT     := LsrC+LsrIC;
AslT     := AslC+AslIC;
AsrT     := AsrC+AsrIC;
DlslT    := DlslC+DlslIC;
DlsrT    := DlsrC+DlsrIC;
CslT     := CslC+CslIC;

BrT       := BrGTSC+BrGTLC+BrEQSC+BrEQLC+BrIGTSC+BrIGTLC+BrILTSC+BrILTLC+BrIEQSC+BrIEQLC
             +BrLESC+BrLELC+BrNESC+BrNELC+BrSC+BrLC+BrILESC+BrILELC+BrIGESC+BrIGELC
             +BrINESC+BrINELC;

CallT     := CallSC+CallLC+CallrC;

LoopT     := LoopSC+LoopLC;

LoadInstC        := LoadbT+LoadhT+LoadT+LoaddT+LaddrT+LoadbpT+LoadhpT+LoadpT+LoaddpT+LaddrpT;
StoreInstC       := StorebT+StorehT+StoreT+StoredT;

SingleIntegerC   := AddT+DivC+NegC+MpyT+RemC+SubT;
ExtendedIntegerC := EaddC+EdivC+EmpyC+EsubC;

SingleRealC      := RnegC+RaddC+RsubC+RmpyC+RdivC+FloatC+FixrC+FixtC+MakerdC;
DoubleRealC      := DfixrC+DfixtC+DfloatC+DraddC+DrDivC+DrmpyC+DrnegC+DrsubC+MakedrC;

BitManipulationC := AslT+AsrT+CbitC+CslT+DlslT+DlsrT+LslT+LsrT+SbitC+TbitC;

LogicC           := AndT+NotT+OrC+XorC;

TestT            := TestGTC+TestLTC+TestEQC+TestIGTC+TestILTC+TestIEQC+TestLEC+TestGEC
                    +TestNEC+TestILEC+TestIGEC+TestINEC;

MoveT            := MoveC+MoveIC;

ComparisonC      := DcompC+DrCompC+LcompC+RcompC;

SignExtendC      := SebC+SehC;

BranchCallC      := BrT+CallT+LoopT+RetC;

MemoryReferenceC    := LoadInstC+StoreInstC;
TotalMemoryReferenceC := MemoryReferenceC+BrT+CallT-CallrC+LoopT;
IntegerArithmeticC := SingleIntegerC+ExtendedIntegerC;
RealArithmeticC     := SingleRealC+DoubleRealC;
TotalInstructionC   := MemoryReferenceC+IntegerArithmeticC+RealArithmeticC+BitManipulationC
                       +LogicC+TestT+MoveT+ComparisonC+SignExtendC+BranchCallC;
T := TotalInstructionC;

(* Register to register formatted instructions counter. *)
RegToRegFormatC     := IntegerArithmeticC+RealArithmeticC+BitmanipulationC+LogicC+TestT+MoveT
                       +ComparisonC+SignExtendC+CallrC+RetC;
```

```
(* Short displacement instructions counter. *)
ShortOffsetC        := StorebSC+StorebISC+StorehSC+StorehISC+StoreSC+StoreISC+StoredSC+StoredISC
                       +LoadbSC+LoadbISC+LoadhSC+LoadhISC+LoadSC+LoadISC+LoaddSC+LoaddISC
                       +LaddrSC+LaddrISC+LoadbpSC+LoadbpISC+LoadhpSC+LoadhpISC+LoadpSC+LoadpISC
                       +LoaddpSC+LoaddpISC+LaddrpSC+LaddrpISC
                       +BrGTSC+BrEQSC+CallSC+BrIGTSC+BrILTSC+BrIEQSC+LoopSC+BrLESC+BrNESC+BrSC
                       +BrILESC+BrIGESC+BrINESC;

(* Long displacement instructions counter. *)
LongOffsetC         := StorebLC+StorebILC+StorehLC+StorehILC+StoreLC+StoreILC+StoredLC+StoredILC
                       +LoadbLC+LoadbILC+LoadhLC+LoadhILC+LoadLC+LoadILC+LoaddLC+LoaddILC
                       +LaddrLC+LaddrILC+LoadbpLC+LoadbpILC+LoadhpLC+LoadhpILC+LoadpLC+LoadpILC
                       +LoaddpLC+LoaddpILC+LaddrpLC+LaddrpILC
                       +BrGTLC+BrEQLC+CallLC+BrIGTLC+BrILTLC+BrIEQLC+LoopLC+BrLELC+BrNELC+BrLC
                       +BrILELC+BrILELC+BrINELC;
RegToRegFormatP     := RegToRegFormatC/T*100;
ShortOffsetP        := ShortOffsetC/T*100;
LongOffsetP         := LongOffsetC/T*100;

(* Direct Memory Access Counter. *)
DirectC  := BrT+CallT-CallrC+LoopT+StorebSC+StorebLC+StorehSC+StorehLC+StoreSC+StoreLC
            +StoredSC+StoredLC+LoadbSC+LoadbLC+LoadhSC+LoadhLC+LoadSC+LoadLC
            +LoaddSC+LoaddLC+LaddrSC+LaddrLC+LoadbpSC+LoadbpLC+LoadhpSC+LoadhpLC
            +LoaddpSC+LoaddpLC+LoadpSC+LoadpLC+LaddrpSC+LaddrpLC;

(* Indexed Memory Access Counter. *)
IndexC      := StorebISC+StorebILC+StorehISC+StorehILC+StoreISC+StoreILC+StoredISC+StoredILC
               +LoadbISC+LoadbILC+LoadhISC+LoadhILC+LoadISC+LoadILC+LoaddISC+LoaddILC
               +LaddrISC+LaddrILC+LoadbpISC+LoadbpILC+LoadhpISC+LoadhpILC+LoadpISC+LoadpILC
               +LoaddpISC+LoaddpILC+LaddrpISC+LaddrpILC;
DirectP     := DirectC/TotalMemoryReferenceC*100;
IndexP      := IndexC/TotalMemoryReferenceC*100;

UnconditionalBrC  := BrSC+BrLC;
ConditionalBrC    := BrT-UnconditionalBrC;

(* To prevent overflow in case BrT = 0. *)
If BrT = 0 Then Begin
                  UnconditionalBrP := 0;
                  ConditionalBrP := 0;
                End
           Else Begin
                  UnconditionalBrP  := UnconditionalBrC/BrT*100;
                  ConditionalBrP    := ConditionalBrC/BrT*100;
                End;

CodeSpaceAccessC  := LoadbpT+LoadhpT+LoadpT+LoaddpT+LaddrpT;
DataSpaceAccessC  := TotalMemoryReferenceC-CodeSpaceAccessC;
CodeSpaceAccessP  := CodeSpaceAccessC/TotalMemoryReferenceC*100;
DataSpaceAccessP  := DataSpaceAccessC/TotalMemoryReferenceC*100;

If LoopT = 0   Then LoopLength := 0
               Else LoopLength := TotalLoop/LoopT;
If BrT = 0     Then BrLength := 0
               Else BrLength := TotalBr/BrT;
If CallT = 0   Then CallLength := 0
```

```
              Else CallLength := TotalCall/(CallSC+CallLC);

MemoryP                        := MemoryReferenceC/T*100;
LoadInstP                      := LoadInstC/T*100;
                LoadP    := LoadT/T*100;
                LoadbP   := LoadbT/T*100;
                LoaddP   := LoaddT/T*100;
                LoadhP   := LoadhT/T*100;
StoreInstP                     := StoreInstC/T*100;
                StoreP   := StoreT/T*100;
                StorebP  := StorebT/T*100;
                StoredP  := StoredT/T*100;
                StorehP  := StorehT/T*100;

                LaddrP   := LaddrT/T*100;

                LoadbpP  := LoadbpT/T*100;
                LoadhpP  := LoadhpT/T*100;
                LoadpP   := LoadpT/T*100;
                LoaddpP  := LoaddpT/T*100;

                LaddrpP  := LaddrpT/T*100;
IntegerP                       := IntegerArithmeticC/T*100;
SingleIntegerP                 := SingleIntegerC/T*100;
                AddP     := AddT/T*100;
                DivP     := DivC/T*100;
                NegP     := NegC/T*100;
                MpyP     := MpyT/T*100;
                RemP     := RemC/T*100;
                SubP     := SubT/T*100;
ExtendedIntegerP               := ExtendedIntegerC/T*100;
RealP                          := RealArithmeticC/T*100;
SingleRealP                    := SingleRealC/T*100;
                RnegP    := RnegC/T*100;
                RaddP    := RaddC/T*100;
                RsubP    := RsubC/T*100;
                RmpyP    := RmpyC/T*100;
                RdivP    := RdivC/T*100;
                FloatP   := FloatC/T*100;
                FixrP    := FixrC/T*100;
                FixtP    := FixtC/T*100;
                MakerdP  := MakerdC/T*100;
DoubleRealP                    := DoubleRealC/T*100;
BitManipulationP               := BitManipulationC/T*100;
                AslP     := AslT/T*100;
                AsrP     := AsrT/T*100;
                CbitP    := CbitC/T*100;
                CslP     := CslT/T*100;
                DlslP    := DlslT/T*100;
                DlsrP    := DlsrT/T*100;
                LslP     := LslT/T*100;
                LsrP     := LsrT/T*100;
                SbitP    := SbitC/T*100;
                TbitP    := TbitC/T*100;
LogicP                         := LogicC/T*100;
                AndP     := AndT/T*100;
```

```
                    NotP     := NotT/T*100;
                    OrP      := OrC/T*100;
                    XorP     := XorC/T*100;
TestP                        := TestT/T*100;
MoveP                        := MoveT/T*100;
ComparisonP                  := ComparisonC/T*100;
                    DcompP   := DcompC/T*100;
                    DrcompP  := DrcompC/T*100;
                    LcompP   := LcompC/T*100;
                    RcompP   := RcompC/T*100;
SignExtendP                  := SignExtendC/T*100;
                    SebP     := SebC/T*100;
                    SehP     := SehC/T*100;
BranchCallP                  := BranchCallC/T*100;
                    BrP      := BrT/T*100;
                    CallP    := CallT/T*100;
                    CallrP   := CallrC/T*100;
                    LoopP    := LoopT/T*100;
                    RetP     := RetC/T*100;

Writeln ('**** MEMORY REFERENCE INSTRUCTIONS   =',MemoryReferenceC,' (',MemoryP,' )');
Writeln (' *** LOAD  INSTRUCTIONS              =',LoadInstC,' (',LoadInstP,' )');
Writeln ('     ** LoadB = ',LoadbT,'(',LoadbP,')');
Writeln ('         * LoadbS : ',LoadbSC,' * LoadbIS : ',LoadbISC,' * LoadbL : ',LoadbLC,
        ' * LoadbIL : ',LoadbILC);
Writeln ('     ** LoadH = ',LoadhT,'(',LoadhP,')');
Writeln ('         * LoadhS : ',LoadhSC,' * LoadhIS : ',LoadhISC,' * LoadhL : ',LoadhLC,
        ' * LoadhIL : ',LoadhILC);
Writeln ('     ** Load  = ',LoadT,'(',LoadP,')');
Writeln ('         * LoadS  : ',LoadSC,' * LoadIS : ',LoadISC,' * LoadL  : ',LoadLC,
        ' * LoadIL  : ',LoadILC);
Writeln ('     ** Loadd = ',LoaddT,'(',LoaddP,')');
Writeln ('         * LoaddS : ',LoaddSC,' * LoaddIS : ',LoaddISC,' * LoaddL : ',LoaddLC,
        ' * LoaddIL : ',LoaddILC);
Writeln ('     ** Laddr  = ',LaddrT,'(',LaddrP,')');
Writeln ('         * LaddrS : ',LaddrSC,' * LaddrIS : ',LaddrISC,' * LaddrL : ',LaddrLC,
        ' * LaddrIL : ',LaddrILC);
Writeln ('     ** Laddrp = ',LaddrpT,'(',LaddrpP,')');
Writeln ('         * LaddrpS : ',LaddrpSC,' * LaddrpIS : ',LaddrpISC,' * LaddrpL : ',LaddrpLC,
        ' * LaddrpIL : ',LaddrpILC);
Writeln ('     ** Loadbp = ',LoadbpT,'(',LoadbpP,')');
Writeln ('         * LoadbpS : ',LoadbpSC,' * LoadbpIS : ',LoadbpISC,' * LoadbpL : ',LoadbpLC,
        ' * LoadbpIL : ',LoadbpILC);
Writeln ('     ** Loadhp = ',LoadhpT,'(',LoadhpP,')');
Writeln ('         * LoadhpS : ',LoadhpSC,' * LoadhpIS : ',LoadhpISC,' * LoadhpL : ',LoadhpLC,
        ' * LoadhpIL : ',LoadhpILC);
Writeln ('     ** Loadp = ',LoadpT,'(',LoadpP,')');
Writeln ('         * LoadpS : ',LoadpSC,' * LoadpIS : ',LoadpISC,' * LoadpL : ',LoadpLC,
        ' * LoadpIL : ',LoadpILC);
Writeln ('     ** Loaddp = ',LoaddpT,'(',LoaddpP,')');
Writeln ('         * LoaddpS : ',LoaddpSC,' * LoaddpIS : ',LoaddpISC,' * LoaddpL : ',LoaddpLC,
        ' * LoaddpIL : ',LoaddpILC);
Writeln ( '    ** Laddrp = ',LaddrpT,'(',LaddrpP,')');
Writeln ('         * LaddrpS ',LaddrpSC,' * LaddrpIS : ',LaddrpISC,' * LaddrpL : ',LaddrpLC,
        ' * LaddrpIL : ',LaddrpILC);
Writeln;
```

```
Writeln (' *** STORE INSTRUCTIONS              =',StoreInstC,' (',StoreInstP,' )');
Writeln ('      ** StoreB = ',StorebT,'(',StorebP,')');
Writeln ('         * StorebS : ',StorebSC,' * StorebIS : ',StorebISC,' * StorebL : ',StorebLC,
       ' * StorebIL : ',StorebILC);
Writeln ('      ** StoreH = ',StorehT,'(',StorehP,')');
Writeln ('         * StorehS : ',StorehSC,' * StorehIS : ',StorehISC,' * StorehL : ',StorehLC,
       ' * StorehIL : ',StorehILC);
Writeln ('      ** Store  = ',StoreT,'(',StoreP,')');
Writeln ('         * StoreS : ',StoreSC,' * StoreIS : ',StoreISC,' * StoreL : ',StoreLC,
       ' * StoreIL : ',StoreILC);
Writeln;
Writeln;
Writeln ('**** INTEGER ARITHMETIC INSTRUCTIONS  =',IntegerArithmeticC,' (',IntegerP,' )');
Writeln (' *** SINGLE-PRECISION                 =',SingleIntegerC,' (',SingleIntegerP,' )');
Writeln ('      ** Add    = ',AddT,'(',AddP,')');
Writeln ('         * Add Res. : ',AddC,' * Add I. : ',AddIC);
Writeln ('      ** Div    = ',DivC,'(',DivP,')');
Writeln ('      ** Neg    = ',NegC,'(',NegP,')');
Writeln ('      ** Mpy    = ',MpyT,'(',MpyP,')');
Writeln ('         * Mpy Res. : ',MpyC,' * Mpy I. : ',MpyIC);
Writeln ('      ** Rem    = ',RemC,'(',RemP,')');
Writeln ('      ** Sub    = ',SubT,'(',SubP,')');
Writeln ('         * Sub Res. : ',SubC,' * Sub I.',SubIC);
Writeln;
Writeln (' *** EXTENDED-PRECISION               =',ExtendedIntegerC,' (',ExtendedIntegerP,' )');
Writeln;
Writeln;
Writeln ('**** REAL ARITHMETIC INSTRUCTIONS     =',RealArithmeticC,' (',RealP,' )');
Writeln (' *** SINGLE-PRECISION                 =',SingleRealC,' (',SingleRealP,' )');
Writeln ('      ** Rneg   = ',RnegC,'(',RnegP,')');
Writeln ('      ** Radd   = ',RaddC,'(',RaddP,')');
Writeln ('      ** Rsub   = ',RsubC,'(',RsubP,')');
Writeln ('      ** Rmpy   = ',RmpyC,'(',RmpyP,')');
Writeln ('      ** Rdiv   = ',RdivC,'(',RdivP,')');
Writeln ('      ** Float  = ',FloatC,'(',FloatP,')');
Writeln ('      ** Fixr   = ',FixrC,'(',FixrP,')');
Writeln ('      ** Fixt   = ',FixtC,'(',FixtP,')');
Writeln ('      ** Makerd = ',MakerdC,'(',MakerdP,')');
Writeln;
Writeln (' *** DOUBLE-PRECISION                 =',DoubleRealC,' (',DoubleRealP,' )');
Writeln;
Writeln;
Writeln ('**** BIT MANIPULATION INSTRUCTIONS    =',BitManipulationC,' (',BitManipulationP,' )');
Writeln ('      ** Asl    = ',AslT,'(',AslP,')');
Writeln ('         * Asl Res. : ',AslC,'     * Asl I : ',AslIC);
Writeln ('      ** Asr    = ',AsrT,'(',AsrP,')');
Writeln ('         * Asr Res. : ',AsrC,'     * Asr I : ',AsrIC);
Writeln ('      ** Cbit   = ',CbitC,'(',CbitP,')');
Writeln ('      ** Csl    = ',CslT,'(',CslP,')');
Writeln ('         * Csl Res. : ',CslC,'     * Csl I : ',CslIC);
Writeln ('      ** Dlsl   = ',DlslT,'(',DlslP,')');
Writeln ('         * Dlsl Res.: ',DlslC,'     * Dlsl I : ',DlslIC);
Writeln ('      ** Dlsr   = ',DlsrT,'(',DlsrP,')');
Writeln ('         * Dlsr Res.: ',DlsrC,'     * Dlsr I : ',DlsrIC);
Writeln ('      ** Lsl    = ',LslT,'(',LslP,')');
Writeln ('         * Lsl Res. : ',LslC,'     * Lsl I : ',LslIC);
```

```
Writeln ('     ** Lsr    = ',LsrT,'(',LsrP,')');
Writeln ('        * Lsr Res. : ',LsrC,'      * Lsr I : ',LsrIC);
Writeln ('     ** Sbit   = ',SbitC,'(',SbitP,')');
Writeln ('     ** Tbit   = ',TbitC,'(',TbitP,')');
Writeln;
Writeln;
Writeln ('**** LOGICAL INSTRUCTIONS               =',LogicC,' (',LogicP,' )');
Writeln ('     ** And    = ',AndT,'(',AndP,')');
Writeln ('        * And Res. : ',AndC,' * And I. : ',AndIC);
Writeln ('     ** Not    = ',NotT,'(',NotP,')');
Writeln ('     ** Or     = ',OrC,'(',OrP,')');
Writeln ('     ** Xor    = ',XorC,'(',XorP,')');
Writeln;
Writeln;
Writeln ('**** TEST INSTRUCTIONS                  =',TestT,' (',TestP,' )');
Writeln ('     ** TestGT  = ',TestGTC,
        '     ** TestLT  = ',TestLTC);
Writeln ('     ** TestEQ  = ',TestEQC,
        '     ** TestIGT = ',TestIGTC);
Writeln ('     ** TestILT = ',TestILTC,
        '     ** TestIEQ = ',TestIEQC);
Writeln ('     ** TestLE  = ',TestLEC,
        '     ** TestGE  = ',TestGEC);
Writeln ('     ** TestNE  = ',TestNEC,
        '     ** TestILE = ',TestILEC);
Writeln ('     ** TestIGE = ',TestIGEC,
        '     ** TestINE = ',TestINEC);
Writeln;
Writeln;
Writeln ('**** DATA MOVEMENT INSTRUCTIONS         =',MoveT,' (',MoveP,' )');
Writeln ('        * Move Res. : ',MoveC,' * Move I. : ',MoveIC);
Writeln;
Writeln;
Writeln ('**** COMPARISON INSTRUCTIONS            =',ComparisonC,' (',ComparisonP,' )');
Writeln ('     ** Dcomp   = ',DcompC,
        '     ** Drcomp  = ',DrcompC);
Writeln ('     ** Lcomp   = ',LcompC,
        '     ** Rcomp   = ',RcompC);
Writeln;
Writeln;
Writeln ('**** SIGN EXTEND INSTRUCTIONS           =',SignExtendC,' (',SignExtendP,' )');
Writeln ('     ** Seb     = ',SebC,
        '     ** Seh     = ',SehC);
Writeln;
Writeln;
Writeln ('**** BRANCH AND CALL INSTRUCTIONS       =',BranchCallC,' (',BranchCallP,' )');
Writeln (' *** BRANCH INSTRUCTIONS                =',BrT,'(',BrP,')');
Writeln ('     ** BrGTS  = ',BrGTSC,
        '     ** BrGTL  = ',BrGTLC);
Writeln ('     ** BrEQS  = ',BrEQSC,
        '     ** BrEQL  = ',BrEQLC);
Writeln ('     ** BrIGTS = ',BrIGTSC,
        '     ** BrIGTL = ',BrIGTLC);
Writeln ('     ** BrILTS = ',BrILTSC,
        '     ** BrILTL = ',BrILTLC);
Writeln ('     ** BrIEQS = ',BrIEQSC,
```

```
        '      ** BrIEQL = ',BrIEQLC);
Writeln ('      ** BrLES  = ',BrLESC,
        '      ** BrLEL  = ',BrLELC);
Writeln ('      ** BrNES  = ',BrNESC,
        '      ** BrNEL  = ',BrNELC);
Writeln ('      ** BrS    = ',BrSC,
        '      ** BrL    = ',BrLC);
Writeln ('      ** BrILES = ',BrILESC,
        '      ** BrILEL = ',BrILELC);
Writeln ('      ** BrIGES = ',BrIGESC,
        '      ** BrIGEL = ',BrIGELC);
Writeln ('      ** BrINES = ',BrINESC,
        '      ** BrINEL = ',BrINELC);
Writeln;
Writeln (' *** CALL INSTRUCTIONS                    =',CallT,'(',CallP,')');
Writeln ('      ** CallS  = ',CallSC,
        '      ** CallL  = ',CallLC);
Writeln ('      ** Callr  = ',CallrC);
Writeln;
Writeln (' *** LOOP INSTRUCTIONS                    =',LoopT,'(',LoopP,')');
Writeln ('      ** LoopS  = ',LoopSC,
        '      ** LoopL  = ',LoopLC);
Writeln;
Writeln (' *** RETURN INSTRUCTIONS                  =',RetC,'(',RetP,')');
Writeln;
Writeln;
Writeln ('************************ TOTAL INSTRUCTIONS ....',TotalInstructionC);
Writeln;
Writeln;
Writeln;
Writeln ('**********************************************************************');
Writeln ('************************ ANALYSIS RESULT ************************');
Writeln ('**********************************************************************');
Writeln;
Writeln ('####     Number of Reg. to Reg. format instructions    : ',RegToRegForMatC);
Writeln ('           % in total Memory Reference instructions    : ',RegToRegFormatP);
Writeln;
Writeln ('  ##     Number of Short displacement instructions     : ',ShortOffsetC);
Writeln ('           % in total Memory Reference instructions    : ',ShortOffsetP);
Writeln;
Writeln ('  ##     Number of Long displacement instructions      : ',LongOffsetC);
Writeln ('           % in total Memory Reference instructions    : ',LongOffsetP);
Writeln;
Writeln;
Writeln ('  ##     Number of DIRECT MEMORY ACCESS instructions   : ',DirectC);
Writeln ('           % in total Memory Reference instructions    : ',DirectP);
Writeln;
Writeln ('  ##     Number of INDEXED MEMORY ACCESS instructions  : ',IndexC);
Writeln ('           % in total Memory Reference instructions    : ',IndexP);
Writeln;
Writeln;
Writeln ('####     Number of total BRANCH instructions           : ',BrT);
Writeln ('                    % in Total Instructions            : ',BrP);
Writeln;
Writeln ('  ##     Number of Conditional Branch instructions     : ',ConditionalBrC);
Writeln ('                   % in total Branch instructions      : ',ConditionalBrP);
```

```
Writeln;
Writeln ('  ##    Number of Unconditional Branch instructions   : ',UnconditionalBrC);
Writeln ('                 % in total Branch instructions   : ',UnconditionalBrP);
Writeln;
Writeln;
Writeln ('####    Total # of code space access instructions    : ',CodeSpaceAccessC);
Writeln ('           % in total memory access instructions    : ',CodeSpaceAccessP);
Writeln;
Writeln ('####    Total # of data space access instructions    : ',DataSpaceAccessC);
Writeln ('           % in total memory access instructions    : ',DataSpaceAccessP);
Writeln;
Writeln ('####    Average length of loop displacement in byte   : ',LoopLength);
Writeln;
Writeln ('####    Average length of branch displacement in byte : ',BrLength);
Writeln;
Writeln ('####    Average length of call displacement in byte   : ',CallLength);
end;


(*************************************)
(* Procedure to reset all counters.*)
(*************************************)
Procedure ResetVariables;

begin

TotalLoop := 0; TotalBr := 0;   TotalCall := 0;

MoveC := 0;      NegC := 0;       AddC := 0;       SubC := 0;       MpyC := 0;
DivC := 0;       RemC := 0;       NotC := 0;       OrC := 0;        XorC := 0;
AndC := 0;       CbitC := 0;      SbitC := 0;      TbitC := 0;      ChkC := 0;
NopC := 0;       MoveIC := 0;     AddIC := 0;      SubIC := 0;      MpyIC := 0;
NotIC := 0;      AndIC := 0;      ChkIC := 0;      FixtC := 0;      FixrC := 0;
RnegC := 0;      RaddC := 0;      RsubC := 0;      RmpyC := 0;      RdivC := 0;
MakerdC := 0;    LcompC := 0;     FloatC := 0;     RcompC := 0;     EaddC := 0;
EsubC := 0;      EmpyC := 0;      EdivC := 0;      DfixtC := 0;     DfixrC := 0;
DrnegC := 0;     DraddC := 0;     DrSubC := 0;     DrdivC := 0;     DrmpyC := 0;
MakedrC := 0;    DcompC := 0;     DfloatC := 0;    DrcompC := 0;    TrapC := 0;
SusC := 0;       LusC := 0;       RumC := 0;       LdressC := 0;    TransC := 0;
DirtC := 0;      MaintC := 0;     ReadC := 0;      WriteC := 0;     TestGTC := 0;
TestLTC := 0;    TestEQC := 0;    CallrC := 0;     TestIGTC := 0;   TestILTC := 0;
TestIEQC := 0;   RetC := 0;       TestLEC := 0;    TestGEC := 0;    TestNEC := 0;
KcallC := 0;     TestILEC := 0;   TestIGEC := 0;   TestINEC := 0;   LslC := 0;
LsrC := 0;       AslC := 0;       AsrC := 0;       DlslC := 0;      DlsrC := 0;
CslC := 0;       SebC := 0;       LslIC := 0;      LsrIC := 0;      AslIC := 0;
AsrIC := 0;      DlsrIC := 0;     DlslIC := 0;     CslIC := 0;      SehC := 0;
BrGTSC := 0;     BrEQSC := 0;     CallSC := 0;     BrIGTSC := 0;    BrILTSC := 0;
BrIEQSC := 0;    LoopSC := 0;     BrLESC := 0;     BrNESC := 0;     BrSC := 0;
BrILESC := 0;    BrIGESC := 0;    BrINESC := 0;    BrGTLC := 0;     BrEQLC := 0;
CallLC := 0;     BrIGTLC := 0;    BrILTLC := 0;    BrIEQLC := 0;    LoopLC := 0;
BrLELC := 0;     BrNELC := 0;     BrLC := 0;       BrILELC := 0;    BrIGELC := 0;
BrINELC := 0;    StorebSC := 0;   StorebISC := 0;  StorehSC := 0;   StorehISC := 0;
StoreSc := 0;    StoreISC := 0;
StoredSC := 0;   StoredISC := 0;  StorebLC := 0;   StorebILC := 0;
StorehLC := 0;   StorehILC := 0;  StoreLC := 0;    StoreILC := 0;   StoredLC := 0;
```

```
StoredILC := 0; LoadbSC := 0;   LoadbISC := 0;  LoadhSC := 0;   LoadhISC := 0;
LoadSC := 0;    LoadISC := 0;   LoaddSC := 0;   LoaddISC := 0;  LaddrSc := 0;
LaddrISC := 0;  LoadbLC := 0;   LoadbILC := 0;  LoadhLC := 0;   LoadhILC := 0;
LoadLC := 0;    LoadILC := 0;   LoaddLC := 0;   LoaddILC := 0;  LaddrLC := 0;
LaddrILC := 0;  LoadbpSC := 0;  LoadbpISC := 0; LoadhpSC := 0;  LoadhpISC :=0;
LoadpSc := 0;   LoadpISC := 0;  LoaddpSc := 0;  LoaddpISC := 0; Laddrpsc := 0;
LaddrpISC := 0;
LoadbpLC := 0;  LoadbpILC := 0; LoadhpLC := 0;  LoadhpILC := 0; LoadpLC := 0;
LoadpILC := 0;  LoaddpLC := 0;  LoaddpILC := 0; LaddrpLC := 0;  LaddrpILC := 0;
end;



(***************************************************************************)
(***************************************************************************)
(*******************     MAIN  PROGRAM   ***********************************)
(***************************************************************************)


BEGIN {Main}

Writeln ('* Type the number of characters in File Name :');
Readln (MaxNumberOfName);
Writeln('* Please type the File Name each character at once     : ');
For i := 1 to MaxNumberOfName Do
 Readln(CharArray[i]);
Writeln;
Writeln ('* Thank you !');
Writeln ('* Do you need code print out ? (y/n) :');
Readln(Tempchar);
Writeln;
If TempChar = 'y'
   Then CodeListing := TRUE
   Else CodeListing := FALSE;
Writeln ('Start of analyzing ');
For i := 1 to MaxNumberOfName Do
  Writeln (CharArray[i]);
Writeln (' file.....');
FileName  := NewString (MaxNumberOfName);
For i :=1 to MaxNumberOfName Do
 FileName^.Chars[i] := CharArray[i];

(* Open target program "XX.o" *)
OpenFile (DataFile,FileName,'r');
ResetVariables;
FirstPass;
SecondPass;
ReadIntoCodeMemory;
Execution;
PrintStatistics;
CloseFile (DataFile)

END.{Main}
```

APPENDIX D

RIDGE 32 OPCODE MAP

Figure D-1 Ridge 32 Opcode Map