NOTICE

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

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

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

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

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

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED
THE INTEGRATION OF CONTROL STRUCTURES
IN MACHINE ARCHITECTURES

by

Luc A. Lepine

A thesis presented to the School of Graduate Studies and Research
of the University of Ottawa
in partial fulfillment of the requirements for the degree
of Master in Computer Science
in the Department of Computer Science

© Luc A. Lepine, Ottawa, Canada, 1984.
ACKNOWLEDGEMENTS

I wish to thank Peter Hickey for the technical assistance he provided, Dr. Robert Probert, Dr. Stan Matwin and Dr. Jacques Raymond for the useful suggestions they provided when reviewing this paper.
ABSTRACT

This thesis attempts to describe the development and installation of machine instructions which implement the following control structures:

1. If-Then-Else
2. While-Do
3. Repeat-Until

The description includes the syntax, function, restrictions, implementation and use of the new instructions in their chosen environments. The environments are an old but still commonly used machine architecture and a more recently developed pseudo machine architecture for a popular programming language.
CONTENTS

ACKNOWLEDGEMENTS ........................................ iii
ABSTRACT .................................................. iv

Chapter ........................................ Page
I.  INTRODUCTION ........................................ 1

II. HISTORY AND DESIGN ................................ 4
   A Definition of a Control Structure ............... 5
   A Brief History of Control Structures ............ 5
   Properties and Traits ................................ 7
   The If-Then-Else structure ......................... 8
   The While-Do structure .............................. 9
   The Repeat-Until structure .......................... 9
   Design ............................................. 10

III. IMPLEMENTATIONS ................................... 16

   Implementation on the PERQ ......................... 17
   The New Instructions ................................ 19
   Opcodes ............................................ 22
   Compilation and Usage ................................ 26
   Decompilation ...................................... 26
   Altered Routines ................................... 27
   Altered Control Block(s) ............................ 28
   Interrupts .......................................... 28
   Examples and Testing ................................ 30

   Implementation on the AMDAHL ..................... 31
   The New Instructions ................................ 32
   Opcodes ............................................ 34
   Compilation and Usage ................................ 38
   Decompilation ...................................... 38
   Altered Control Block(s) ............................ 39
   Altered Routines .................................... 40
   Interrupts .......................................... 41
   Examples and Testing ................................ 42
   Observations ........................................ 42

IV. CONCLUSIONS ......................................... 45
<table>
<thead>
<tr>
<th>Appendix</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. QC0DE DEFINITIONS</td>
<td>50</td>
</tr>
<tr>
<td>B. PERQ MICROCODE SOURCE</td>
<td>59</td>
</tr>
<tr>
<td>C. INSTRUCTION TEST SAMPLE</td>
<td>107</td>
</tr>
<tr>
<td>D. AMDAHL INSTRUCTION EMULATING ROUTINE</td>
<td>139</td>
</tr>
<tr>
<td>E. TEST PROGRAMS FOR AMDAHL V7A</td>
<td>149</td>
</tr>
<tr>
<td>F. DISASSEMBLED I.B.M. OBJECT CODE</td>
<td>160</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>163</td>
</tr>
</tbody>
</table>
Chapter I
INTRODUCTION

Control structures have had many forms. The earliest form of control structures used the GOTO or simple branch instruction. Although it has many forms (i.e. in FORTRAN the computed, assigned and unconditional GOTO statements) it has been utilised to give structure to the execution of programs. These instructions are used explicitly, and are generated by pre-processors and compilers to implement more sophisticated control structures, such as If-Then-Else and Case. A major problem with this implementation of a control structure is that programs relying heavily on this type of instruction tend to be cryptic and unreadable[5], requiring a lot of effort, even on the author's part, to determine what had actually been done.

The development of the If-Then-Else, While-Do and Repeat-Until control structures followed a trend towards the use of more general, understandable and expressive control structures. Their use in Algol-60[8], PL360[14], Pascal[13] and more recently developed languages such as Modula-2[12] and their use in various assembler languages in the form of macros, as in Hibal[11] and Macros[10], show the versatility and general acceptance of these particular control structures.
Other control structures have been designed and are being designed. The trend is towards allowing a simple implementation of some basic programming mechanisms. The result has been the development of such control structures as the CASE statement and the If-Then-ElsIF ... Else End, although the latter is more of a syntactic solution to the problem of heavily nested IF statements. The Case statement has been implemented as an instruction in certain machine environments, such as P-code. However, the basic structures have not yet been implemented in many architectures. The implementation of the more generally recognised control structures using branch instructions still leaves some aspects of control structures relatively untouched by machine architecture. This thesis attempts to describe the implementation of machine instructions for the three most prominent control structures, If-Then-Else, While-Do and Repeat-Ultil.

It has been found by Mills[21] that these control structures have limitations as to the class of problems they can be used in. This has led to the development of other, more general control structures. The paper by Parnas[16] has shown one such control structure. Among its benefits is the apparent removal of the need for temporary variables used in the performance of the control structure and the ability to express non-deterministic algorithms.

The intent of this research has been to:
1. Move the control structures If-Then-Else, While-Do and Repeat-Until into two existing machine architectures. The I.B.M. System/370 and the PERQ Systems PERQ machine which runs an operating system using Pascal, which uses a P-Code based architecture.

2. To determine if they add any legibility to program code.

3. To document the limitations and restrictions in the implementation of these control structures and explain why they occur.

The paper first contains a brief history of control structures and the basic design of each control structure. Then, it proceeds to describe for both machine architectures, which instructions were added, how they were implemented, their syntax, function and use. Any alterations to the operating systems involved is also documented.

The appendixes at the end of the paper contain a list of the definition of each of the Q-Code op-codes found in the PERQ microcode, the listing of the interpreter micro-code for the PERQ and various source files and corresponding disassembled listings. They also include the routine which performs the instruction handling for the AMDAHL V7A, a System/370 compatible machine, two tests which verify the proper functioning of the new instructions on the AMDAHL, and a disassembled listing of one of the tests.
Chapter II
HISTORY AND DESIGN

There are very few prominent control structures. Some have been added since the first implementation of the three most prominent control structures:

1. If-Then-Else
2. While-Do, and
3. Repeat-Until

Other control structures include the Select or Case statement, the For or Do statement and the If-Then-Else-If-...-Else-End type statement.

Some of these other control structures have been implemented on certain machine architectures, to a point. For example, the For and Do statements are a commonly recognised structure found in Pascal, Fortran and PL/1. On I.B.M[17] machines, they have been implemented using the branch on index high [BXH] and branch on index less than or equal [BXLE] instructions so as to limit the number of instructions required to perform a loop with a positive or negative counter.

What we wish to do is to demonstrate how to implement the more commonly used control structures on any available machine, to determine what use they could be, and any restrictions that would apply.
2.1 A DEFINITION OF A CONTROL STRUCTURE

The implementation of a control structure depends on what properties it is given. We need to define a control structure as an entity which is used to control the flow of execution. The discussion is limited to machines which operate in a sequential manner. Because it is defined as a structure, it is a building block which can be used to map the logic of a program. To be of any use it must be available in any valid programming context.

To permit any control structure, the definition must also allow for any method of implementation. It is an abstract concept which permits the user to build a program in a logical manner. This in effect enforces the use of structured programming techniques.

2.2 A BRIEF HISTORY OF CONTROL STRUCTURES

Control structures have been around since the beginning of programming. It is simply a method by which the logical structure of a program is implemented. The structure usually resides in the reader's mind because the implementation is usually done using branch instructions. However, it has been generally recognised that the use of control structures in a programming environment is more suitable than implementing a program than using explicit branch instructions[5].

As previously indicated, the simple branch instruction was, and still is, the way most control structures are im-
implemented on most machines. The reason for this is that it is usually the only way to alter the program counter and change the flow of execution. Some minor changes have been made towards allowing different uses for branch instructions, for example the Branch And Link (BAL) instruction in IBM/370 architecture also places the current value of the program counter in a register before branching, but they still have restrictions on their use in structured programming (i.e. the BAL instruction is unconditional).

The most familiar control structures today are the If-Then-Else and While-Do structures. These were apparently not the first control structures. The Do statement precedes these structures by a couple of years¹. Since then other control structures, some mentioned above, have been implemented. The purpose of these control structures was to make the task of implementing programs as easy as possible.

The easiest way to make something simple is to have it short and recognisable. This allows the concept involved in the decision to be extracted and analysed quickly. The concepts of If-Then-Else and While-Do are very easy to understand and remove many branch instructions. Removal of temporary variables also makes it easy to understand high level languages, which is why they are so attractive for general purpose programming.

¹ We assume Fortran-II was 'created' circa 1958 and Algo1-60 was 'created' circa 1960.
In more recent years, not only has work been done in high level programming languages but in certain low level programming languages, and even machine code. On I.B.M. machines, the assemblers all have very powerful macro processing facilities. These facilities allow the user to code a 'macro' which has the ability to test the 'type' of a parameter, and generate the appropriate code to the implement the most prominent control structures, i.e. If-Then-Else While-Do[11,10]. On certain micro-code driven machines, such as the PERQ\textsuperscript{3}, these control structures are available at the micro-code level, and are used in the micro-operating system to interpret the Q-code, a P-code dialect, which runs the Pascal based operating system.

2.3 PROPERTIES AND TRAITS

The three control structures which are of interest all have similar basic properties. Some other properties are traits of their specific function. All the control structures are performed under control of a boolean expression. They all perform some form of branching operation and they all return control to the instruction syntactically following the structure when their task has been completed.

\textsuperscript{3} The PERQ is manufactured by PERQ Systems Inc., formerly the Three Rivers Computer Corporation, of Pittsburgh Pennsylvania.
What follows is a simple description of the three control structures we are studying. The block A represents the boolean expression which sets the branch condition. The blocks B and C represent the object of the control structures.

2.3.1 The If-Then-Else structure
The If-Then-Else structure has two unique properties. It does not itself perform a loop and it may optionally have a null Else clause.

The common implementation of this control structure on a machine architecture is as follows:

![Diagram](image)

**Figure 1:** Layout of If A Then B (Else C) structure
2.3.2 The While-Do structure

The While-Do instruction performs a loop while the tested condition is true. If the condition is originally false, no statement inside the structure will be executed.

The common implementation of this control structure is:

![Figure 2: Layout of While A Do B structure](image)

2.3.3 The Repeat-Until structure

The Repeat-Until structure is basically a While-Do structure with the statements internal to the While-Do structure being repeated once in front of the structure[21]. This means that the statements are executed at least once, then the condition is applied. Note that the condition used is the logical complement of that used in the If-Then-Else and While-Do structures.
The common implementation of this structure is as follows:

![Diagram of structure]

**Figure 3:** Layout of Repeat B Until A structure

2.4 **DESIGN**

To implement these control structures on a machine architecture, they should each be designed as a single instruction. They may include extra instructions to set up the correct environment but they must be self-contained.

To make each control structure an instruction, its basic properties must be encoded in the instruction. The basic properties of each instruction are:

1. They rely on the evaluation of a boolean expression.
2. They all contain at least one conditional branch instruction. They may also contain unconditional branch instructions.
3. When the condition allows the instruction to fall through, i.e. the statements inside the control structure are no longer to be executed, the next instruction executed is usually the next textual statement. This is not always the case in certain programming languages. For example, in some dialects of Pascal, using P-Code, there is an Exit-GoTo statement or instruction. In Fortran-IV there is the indexed RETURN statement for exiting SUBROUTINES. These operations allow the ability to return to locations other than the one syntactically following the calling statement.

4. They contain one or two blocks of statements to be executed. Each block may contain zero or more instructions.

We should leave the basic machine architecture unchanged to allow all previously-running programs to continue to run successfully with the new control structures. To this end we must break down each structure into its individual components. This will allow us to implement these structures as single instructions. What we obtain is a general design for each of the control structures.

<table>
<thead>
<tr>
<th>If-Then-Else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Expression &lt;- not part of instruction</td>
</tr>
<tr>
<td>Op-Code &lt;- start of instruction</td>
</tr>
<tr>
<td>Address 1 &lt;- Address of routine</td>
</tr>
</tbody>
</table>
2. **If-Then**

- **Address** 2
  - holding Then clause
- **<-- Address of routine**
  - holding Else clause

3. **While-Do**

- **Marker**
  - possible instruction
  - marking position of
  - boolean expression
- **Boolean Expression**
  - not part of instruction
- **<-- start of instruction**
- **Address**
  - address of routine
  - holding statements to be executed.

4. **Repeat-Until**

- **Marker**
  - possible instruction
  - marking position of
  - boolean expression
- **Routine**
  - statement(s) performing
  - required actions inside
loop

Boolean Expression <- as indicated
Op-Code <- instruction

This design requires that we make some basic assumptions on how these control structures function.

1. There must exist sufficient information such that when the control structure is exited, the syntactically next sequential instruction can be executed.

2. The boolean expression to be evaluated cannot be included in the machine instruction. These expressions can be complex and can contain functions and other non-simple expressions.

3. In keeping with the design of the control structures, the boolean expressions will be calculated using facilities provided by the current architecture. The result of the boolean expression will be either the value true or false. The new instructions will test this result, and will conditionally invoke the appropriate code block or procedure. If we require that this expression be re-evaluated we must

4. Maintain a pointer to the expression so that it can be re-evaluated.

5. The instructions implementing these structures cannot perform two or more functions. For example, the evaluation of the boolean expression is a complex task which may require the evaluation of functions, array
references etc.... This would be very difficult to implement in most architectures, i.e. I.B.M. System/370, and therefore not practical.

This is not always the case. There has been work done[7] where the concepts of Top Down Structured Programming have been implemented at the machine level. The technique used three simple structures, DO, WHILE and IF-THEN-ELSE to control the flow of execution of a program. The boolean expression in this case was evaluated by processing the list provided to the instruction. The entire evaluation was performed by the instruction. This is very difficult to do in this case because the addresses used to point to the parameters are constants. We wish to allow any parameter to be used, however it is passed to the architecture, and therefore should allow for variable addressing.

6. Because the instructions cannot perform two or more functions, they will not be interruptable i.e. they will not be cut off when an interrupt becomes pending. Interrupts will be handled as specified by the principles of operation of the machines involved. The invoked procedures are not part of the instructions but of the entire control structure and inherit the masks present when the structure is invoked.
7. The information passed to an instruction (its parameters) must be passed using the normal conventions followed in the architecture. This usually means that specific locations in storage or arguments available as part of the instruction itself can be used to pass information to the instruction. The return address is maintained through the procedure call process.

8. Because the block of statements to be executed are separate from the actual decision made to use them, the block is located outside the instruction itself and is invoked as a procedure. The end of the block uses the normal procedure termination to exit to the appropriate return point.

9. The final implementation is machine dependent.

The one or more blocks which contain the statements to be executed are logically separated using the definitions of the programming languages, although recognising the bounds of a block is not always a trivial problem. This feature allows to implement each block as a procedure which is called by the instruction and has been featured in the definition of the SPLM [20 p. 138] architecture, which has not actually been implemented.
Chapter III
IMPLEMENTATIONS

The redesigned control structures were implemented on two machines. The first is the PERQ. It is a micro-programmable machine which has a host Q-code architecture and Pascal based operating system. With the micro operating system source it was possible to add and alter certain instructions to implement these new control structures.

The second machine was an AMDAHL V7A. This is an I.B.M. System/370 compatible CPU. It is compatible in that it basically handles the same machine instructions, except where machine diagnostics are involved. None of this directly affects a normal user. This machine is not micro-programmable. The implementation is done using what is termed macro-code. We use certain properties in the machine architecture to obtain information for the instruction, then perform that instruction using a special program inside the operating system itself. This approach is primarily used to test the design and implementation of an instruction before actually installing it in a machine, either by altering the hardware or the micro-program running the machine.
3.1 IMPLEMENTATION ON THE PERQ

All three control structures were added to the PERQ's micro-operating system. This was done by adding one instruction for each of the control structures and an instruction to mark the beginning of the boolean expression for the While-Do and Repeat-Until instructions. Note that the boolean expression is not part of the control instruction but it must immediately precede the instruction. Various other routines internal to the micro-operating system were altered to minimise the amount of code that had to be added to install the new instructions. One control block was altered to contain the pointer to the beginning of the boolean expression, when required.

Various resources were required by the new routines which are not immediately available to a normal user. Since the routines were implemented in micro-code, three resources were required.

1. Sufficient writeable control storage (WCS) to install the new instruction were required. This type of memory is for all intents and purposes read only.

2. Two work areas, one to hold a flag, the other to hold a counter which would be used as an offset from the code base. The code base is the location in real storage of the start of the segment which contains the procedure currently being executed.
3. The actual micro-operating system itself. Various routines were altered slightly to permit their use as subroutines, or to alter their behaviour such that they would perform the required extra tasks to initialise the execution of various instructions in the correct manner. For example, the routine which performs the instruction Load Variable Routine Descriptor (LVRD)\(^3\) was altered so that a flag would be set indicating that the termination of the instruction was to result in the return to the calling routine, and not execution of the next instruction.

The work areas were taken from the pool of 256 20\(^4\) bit registers available in the micro-engine. The name of the register which contains the flag is CR0. The name of the register which contains the loop pointer is NOTEL.

This specific implementation is very dependent on the PERQ's hardware characteristics. The use of the work areas and storage addressability are unique to the PERQ. The expansion of the ACB and changing the behaviour of a procedure call could be extended to Q-Code and P-Code.

\(^3\) The LVRD instruction is used to place the generic address of a procedure on top of the PERQ expression stack. This address can be used as a parameter in procedures or for system 'hacking', as in this case. Q-code has three other forms of addresses used to refer to procedures.

\(^4\) The PERQ's registers are 20 bits wide, so they can be used to address and calculate addresses for 1 megabyte of real storage using single precision arithmetic.
3.1.1 The New Instructions

The implementation of the new instructions requires the addition of some new features to the QCode architecture. At initialisation of the micro-operating system, the work areas are initialised to zero, to indicate that all is to proceed as it would have done in the unmodified system.

The two work areas play critical roles in the performance of each machine instruction. CR0, the flag register, is used to signal the modified routines that they are to take a slightly different approach to their tasks. When CR0 is zero, all functions as it would in the unmodified system. When it is one, each routine will behave in a slightly different way. The two routines which use this flag do so in the following manner:

1. The LVRD instruction is called as a subroutine by the new instructions. When the control flag is one, it will use the RETURN micro-instruction instead of proceeding to decode the next instruction.

2. The routine which builds the activation control block (ACB) for procedure calls will use the value in the NOTE1 register instead of the offset of the next instruction as the default return address.

In this way the amount of duplicated code is reduced.

The NOTE1 register is used to indicate a loop boundary. It is set during initialisation to zero. It is stored when a procedure is invoked in the ACB, then set to zero for the
new routine. It is restored when a procedure terminates, and the value is obtained from the ACB. The register is explicitly set by a NOTE instruction. The value stored is the offset from the current code base to the instruction immediately following the NOTE instruction.

Errors can occur in the invocation of the new instructions. There are three classes of errors.

1. An invalid boolean expression. This occurs when the location which is supposed to contain the value true or false does not. The Pascal compiler always places the result of a boolean expression on the top of a 16 element expression stack and this is where these instructions look for the result of the expression.

2. An invalid loop address. For the While-Do and Repeat-Until control structures, the loop boundary must be set to indicate where to go to when a loop is required. This value is found in NOTE1. The value in NOTE1 is undefined when it contains zero. This is because the NOTE instruction cannot set the value in NOTE1 to zero.

3. An invalid procedure address. In the If-Then-Else and While-Do instructions, the address of a procedure to be invoked is required. Because of compiler restrictions, this was supplied by placing the appropriate NOOP or LVRD instructions after the op-code. If these instructions are not found after the op-code, an error is detected.
Because it is possible to detect errors, the built-in micro-code error routine is called with the appropriate parameters to signal the host operating system that an error has occurred. Because of the restricted number of error indicators, those deemed most appropriate were selected to indicate what type of error had occurred.
3.1.1.1 Opcodes

B'01010100'

Instruction

Note

Op-Code X'54'

Operands None.

Function To update the Note register. The value placed in
the Note register is the offset from the code
base, the start of the segment containing this
routine, of the next sequential instruction in
this segment. The value zero can never be placed
into the Note register by this instruction.

Errors None.
Instruction

If-Then-Else

Op-Code X'51'

Operands Two. Both Procedure addresses. The first is the address of the Then clause, the second is the address of the Else clause. The address is created by using either a LVRD instruction (Load Variable Routine Descriptor) or a NOOP instruction. If it is a NOOP instruction, no branch is taken if that clause is invoked.

Function This instruction takes the result of a boolean expression and conditionally performs a two way procedure call. When control is returned to the caller through the return instruction, the next sequential instruction is executed. The result of the boolean expression must be on the top of the expression stack (TOS).

Errors

1. Expression out of range
2. Invalid routine address
Instruction

While-Do

Op-Code X'52'

Operands The address of the routine to be invoked if the expression on top of the expression stack contains the value true. This address takes the form of either a LVRD instruction, or a NOOP instruction. If the address is a NOOP instruction, no branch is taken. The return address of the invoked routine will be that of the instruction following the last instruction executed in this routine.

Errors 1. Invalid expression
2. Invalid loop address
3. Invalid routine address
B'01010011'

Instruction
Repeat-U til

Op-Code X'53'

Operands None.

Function If the expression on top of the expression stack is false, the next instruction executed will be that following the last Note instruction executed in this procedure.

Errors
1. Invalid expression
2. Invalid loop address
3.1.1.2 Compilation and Usage

Because the basic language on the PERQ is Pascal, there exists no assembler for the Q-code. Instead, the designers of the Pascal compiler allowed for the explicit inclusion of various instructions. They are either generated by the compiler to satisfy a user directive, as in the case of the MakeVRD directive which generates a complete LVRD instruction, or inserted completely by the user. Any combination of instructions is valid. The Pascal compiler does not optimise the code. See appendix C for examples of how this is done.

3.1.1.3 Decompilation

To look at the generated code, the routine QDIS was used to disassemble the generated object code. The object code generated by the Pascal compiler on the PERQ is reentrant and contains no data other than that contained in the machine instructions themselves.

QDIS is a table driven program, the table being stored as a data file somewhere on a system file structured device. The table was edited to insert the definition of the four new instructions. The results can be seen in appendix C, after the source of each program. You should note that the location of the control structures is relatively easy to spot.
3.1.2 Altered Routines

Various routines were altered to either provide their function as subroutines, or slightly alter their function so that they would not have to be copied, then modified to obtain the required function.

The PERQ's micro-code initialisation code was altered so that both the control register and the Note register would be initialised to zero.

The routine which performs the LVRD instruction was modified so that if the control register is not zero, the routine would return control to the caller using a 'return' micro-instruction, instead of executing the next instruction.

The routine which creates the ACB was altered so that the value in the Note register would be used as the default return address, not the address of the next instruction, when the control register is not zero. In all cases, this routine also stores the value of the Note register into the ACB, then sets the Note register to zero for the new procedure. This routine resets the register to zero and does not return control to the calling micro-code routine.

The routine which handles the exit of all functions and procedures was altered to restore the value of Note. The value in Note is always restored when a procedure returns to that lexical level in the program.
3.1.3 Altered Control Block(s)

One system control was altered. The Activation Control Block (ACB) was expanded by 1 word. This area contains the current value of the Note register when that procedure is invoked. This does not affect the addressability of any variable in the invoked procedure since the new top of the memory stack is set to the address immediately after the ACB, resulting in no control problems.

3.1.4 Interrupts

The PERQS operating system has a paging facility. The unit of space paged is a segment. There are two types of segments:

1. The data segment. This segment can be up to 64k words long.
2. The code segment. This segment can only be up to 32k words long.

All addressing is primarily based on segments.

In the implementation of the new instructions, only procedure addresses are used. These addresses are interpreted by the micro-code with the help of the operating system into real storage addresses. If the procedure is not in core, the micro-operating system will generate a page fault. The result is that there is no special requirements for system interrupts in the implementation.
The completion of I/O does not interrupt the execution of the micro-code. A flag is set indicating a pending I/O operation. This flag can be tested in the micro-code using the 'IntrPend' flag. With this type of control, handling I/O interrupts is simply done by adding the qualifier 'If IntrPend Call(Vector)'. This causes the micro-operating system to go to a routine which vectors to the correct I/O handling routine. The placement of this qualifier need only be tempered by two restrictions:

1. It uses the address field to perform a call operation, so no other operation using the address field can be used in the same micro-instruction.

2. The condition flags set by the current arithmetic operation in the ALU will be lost if the branch occurs because the next sequential instruction will not be seeing the result of the previous instruction when it tests the condition flags.

As dictated by the design, the instructions will not be interrupted by I/O interruptions or page faults. The mask which dictates when interrupts can be handled still applies and if an interrupt is to be signalled to the operating system, it will be done using the normal paths and at the appropriate times.

Because there are error conditions which can be detected by the new instructions, the routine used to generate exceptions is used to signal to the operating system that an er-
ror has occurred. Because there are only a certain number of predefined error codes used by the micro-code routines to indicate various programming errors, the codes which most closely resembled the meaning of the new errors were used to indicate them.

3.1.5 **Examples and Testing**

Appendix C contains a number of test programs which were run to verify the proper functioning of the new instructions. To insert the new instructions, intrinsics or compiler directives were used to directly place the required instructions into the object code. The object code is also included in some smaller examples to show how they were placed.

The testing on the PERQ is supposed to exercise all paths that could be taken by any of the new instructions. These paths are those generated by conditions which require execution of the statements associated with true and false looping conditions and all the possible exceptions that might be generated by the implementation of the new instructions. No problems were found during testing.
3.2 IMPLEMENTATION ON THE AMDAHL

Only two of the three control structures were implemented as machine instructions on the AMDAHL V7A. The If-Then-Else and While-Do control structures were implemented but the Repeat-Until was not. The reason for this is that the latter control structure, when designed for implementation using IBM/370 architecture, would be implemented as a simple conditional branch instruction, which already exists in IBM/370 architecture.

The actual implementation was done using macro-code. This term refers to the fact that the instructions are not directly implemented in the machine's architecture, but are executed using other machine instructions. The new instruction causes a program check which indicates that the operation is invalid. A specially written routine determines that it is a new instruction, and performs the required changes to the user's registers and program counter to carry out the execution of these new instructions.

This method is used by IBM and those who manufacture IBM/370 compatible CPUs. It allows the design and testing of machine instructions without having to alter the hardware or implement the instructions in micro-code. The mechanism allows the use of the same logic required in the hardware. But using hardware controls, it is possible to easily trace the execution of the pseudo instruction without interfering with the actual function of the machine.
3.2.1 The New Instructions

The specially written routine takes the form of a program written in assembler, and executed on the user's operating system, the Conversational Monitoring System (CMS) in this case. This technique has been used on other operating systems, and is extensively used in another form in VM/370, the host operating system for CMS.

The routine remains permanently in storage while the new instruction set is supposed to work. Another special routine loads this program into storage in such a way that it will not be removed from storage under any normal or abnormal circumstances.

To effect the proper execution of the program we use some common facts about the System/370 machine architecture in creating a short routine to determine the length of an undefined machine instruction. This architecture uses the first 2 bits of the op-code to indicate the actual length of the machine instruction. The binary patterns 00, 01, 10, and 11 correspond to 2, 4, 4, and 6 byte op-codes, even for undefined machine instructions. The result is that it is very simple to pick up all the required information from the Pro-

---

VM/370 is an operating system that runs on machines that support the I.B.M. System/370 architecture. This operating system presents the user with an image of a real machine, with some extensions to allow it to communicate with VM/370. This image allows the user to execute all the functions that a real machine can but allowing for the sharing of various limited resources, i.e. CPU time, disk space, unit record devices (printers, punches etc...), tapes etc....
gram Status Word (PSW), storage and registers, and perform
the instructions.

Since this architecture is consistent, any addresses used
in an instruction takes the form of a base displacement ad-
dress. This relocatable address requires a general purpose
register and an offset to create the absolute address for
use by an instruction. This type of address is called an
'scon'. The address is used to determine the location of
any subroutine to be executed. Subroutine linkage conven-
tions in this architecture are also consistent. The regis-
ter which is to contain the return address is also pointed
to by the instruction.

The Branch Condition is determined in the same way it is
for Branch on Condition instructions in this architecture.
The branch mask is used to test the Condition Code (CC) bits
found in the PSW. The mask allows the program to select and
test any combination of conditions, and perform a branch if
the test if successful. This same technique is used to
determine the branch condition in the new instructions.
3.2.1.1 Opcodes

| B'1110100' | ml | r1 | d1(b1) | d2(b2) |

Instruction

If-Then-Else

Mnemonic IFTE

Op-Code X'F4'

Operands Two relocatable address constants. The first is the address of the routine to be evoked if the branch condition is true[D1(B1)]. The second is the address of the routine to be evoked if the branch condition is false[D2(B2)].

A mask[M1] is used to determine the validity of the branch condition. For a full explanation see [17] for a description of the Branch on Condition instructions.

A link register[R1], which indicates which general purpose register is to hold the address of the area immediately after this instruction.

Function This instruction performs the function of the If-Then-Else structure at the machine level. Using the mask, the instruction determines whether the branch condition is true or false. Using this in-
formation, it takes either the first or second address and uses it to calculate the new value of the program counter. In all cases, the link register will contain the address of the field immediately after this instruction.

Errors: None.
Instruction
If-Then

Mnemonic  IFT

Op-Code  X'Al'

Operands  One relocatable address[Dl(B1)], a link register indicator[Rl] and a mask[Ml] to determine the branch condition.

Function  The branch mask is used to determine the branch condition. If the condition is true, the address field is used to determine the address of the next instruction to be executed, and the program counter is updated. If the branch condition is true, the indicated register is updated with the address of the area immediately following this instruction.

Errors  None.
Instruction

While-Do

Mnemonic

While

Op-Code

X'A0'

Operands

One relocatable address[D1(B1)], a link register indicator[R1] and a mask[M1] to determine the branch condition.

Function

The branch mask is used to determine the branch condition. If the condition is true, the address field is used to determine the address of the next instruction to be executed, and the program counter is updated. If the branch condition is true, the indicated register is updated with the address of this instruction.

Errors

None.
3.2.1.2 Compilation and Usage

To use these new instructions, they had to be explicitly included in the assembly of some program. Since there are many assemblers for I.B.M. System/370 architecture machines, we used one of them to generate the object code.

The nature of this architecture is that data and instructions are intermingled, there is no definite pattern which identifies one from the other. To use the new instructions, they merely have to be coded directly into an instruction stream. This was done using macros, one for each of the instructions, using the mnemonics given above.

The assembler is capable of generating any form of address constant the machine can use, be it absolute or relocatable. Using this ability, the instructions were written such that they are compatible with other machine instructions. The result is that the IFT and WHILE instructions are RS type instructions, the IFTE instruction is an SS type instruction.

3.2.1.3 Decompilation

As in the PERQ, there exists a routine which can decompose the object code generated by a compiler into a series of instructions. However, since System/370 architecture allows the user to imbed data into the object code, the information provided is not entirely consistent with the original source program. The program (HMASPZAP or 'super zap') will
display an op-code for each half-word of data in the object code if the value corresponds to a valid op-code. Altering the tables allows us to insert our new op-codes. The results can be seen in appendix F. The result is that locating the control structure is now easier, but the results must be verified with the actual program being decoded.

3.2.2 Altered Control Block(s)

To install the routine which handles the new instructions, an alteration to a field in the system nucleus was required. The field, the program check new psw, was replaced with information which points the hardware to a new interrupt handler for program checks. This handler is the second part of the program which performs the new instructions.

The first part of the program actually swaps the program check new psw with another value, and remembers its contents for use by the second part. It also restores the value when called to do so. This part is evoked by the CMS command 'NEWINST'.

The only other field used in the system nucleus was the general purpose register logout area (GPRLOG). This field is used when the 'system store' operation is performed by the hardware. Since this operation is only performed when a system dump will be performed, it is not hazardous to use this field to temporarily store information. Some operating system routines use this field for the same purpose.
3.2.3 Altered Routines

No system routines were altered to implement these new instructions. The new instructions are implemented by altering a field in the operating system which points to the routine which gains control when a programming error occurs. The hardware uses this field to pass control to the operating system when a programming error occurs. By replacing this field with one which points to the new routine which performs the new machine instructions, it is possible to suppress the error and perform the require function.

The new routine determines whether the interrupt is due to one of the new instructions. When this is not the case, the host operating system, obtains control to perform the normal recovery procedure. The steps which determine this are:

1. The program check old psw is examined, to determine which type of program check caused the interrupt. To be handled by the new routine it must be an undefined machine instruction.

2. Determine the supposed length of the instruction that was to be executed. This information can be determined from the instruction length field in the old psw.

3. The address of the instruction is then determined.

4. The op-code is checked. If it is one of the new instructions, a branch to the appropriate routine is made to emulate the instruction.
This method, sometimes referred to as 'macro-coding' the instructions, is usually used when new instruction logic and performance are to be checked. It is easier to install and verify the performance of the instructions in this case than to alter to hardware to execute the instructions.

3.2.4 **Interrupts**

Because of the method of implementation, no hardware interrupts will be fielded by the code handling the instructions. Interrupts handled by the software fall into two categories:

1. Those generated by the hardware, and
2. Those generated by the software.

Those interrupts generated by the hardware, I/O, external, and machine check interrupts are disabled during the execution of this code. Interrupts in the second category are not generated. Supervisor calls are explicitly generated by the 'SVC' instruction, which is not used in the emulating routine. Program checks, which are generated by program errors, are not generated if the emulating program is correct.

Errors can be generated by the new instructions, but the hardware will perform the checks far more quickly than the emulating program. These errors, having to do with storage addressing and protection, will be checked by the hardware when the instruction emulation has been completed. The appropriate errors will be generated by the hardware, so the tests have not been included in the emulating code.
3.2.5 **Examples and Testing**

Appendix E contains two sample programs which make use of the If-Then-Else, If-Then and While-Do instructions. The tests for these new instructions were designed to exercise all the conditions to which the new instructions could be exposed. Since there were no exceptions that could be generated that would not be detected by the hardware, only simple programs which exercised the consistency of the instructions were written. These tests found one problem. This problem was associated with the incorrect determination of the condition code. It was easily found and resolved using the instruction tracing mechanism available in the hardware coupled to some software in the host operating system.

### 3.3 Observations

There are some technical and aesthetic aspects to these new instructions on the AMDAHL which differ from the ones implemented on the PERQ.

1. The **IF-THEN** and **IF-THEN-ELSE** instructions are different on the AMDAHL. This occurs because there is no simple method to indicate to the hardware that the address is a no-op. The method which is used in some System/370 instructions to indicate a null address cannot be used for these new instructions. The resulting address is calculated as zero, which in this case is valid, and cannot be ignored.
2. The branch instruction is not so nearly restricted as it is in the PERQ. Because addressing is done using actual storage addresses and not offsets, as it is on the PERQ, a jump could be calculated to jump from one procedure to another. On the PERQ, this would be difficult, if not impossible in some circumstances, to do.

3. The control block addresses could be passed as parameters. For the PERQ the address was hand coded into the instruction, but using the relocatable address format in the System/370 does allow a parameter to be used. The PERQs instruction format could be altered to allow this type of function.

4. The nesting of procedures on the PERQ and AMDAHL are limited by the amount of storage available. For the PERQ, the total size of stack cannot exceed 64K. For the AMDAHL it is limited to amount of storage that can be addresses, 16384K (or 16 megabytes). In either case, the amount of nesting could get very unwieldy, as it is for heavily nested IF statements.

5. The evaluation of some boolean expressions may result in the generation of code which is far more complex than the control structure being invoked. If a compiler or pre-processor attempts to optimise the evaluation of a boolean expression, many branch instructions may be generated in an attempt to quickly
evaluate an expression. This would cause severe problems in decoding the resulting object code.

6. The actual implementation of these control structures results in the creation of conditional procedure calls, which are not available in either of the architectures studied.
Chapter IV
CONCLUSIONS

The result of attempting to implement various control structures in different architectures pointed out problems in some currently used machine architectures which restrict the implementation of new instructions. This can be observed in the context of machine architecture limitations and dependencies which usually fall into the following categories:

1. Address space size.
2. Real storage size. This does not have to be the same as address space size. For example, the IBM System/38 [20 pp 96-98] has a 48 bit addressing capability but only 31 bit real storage addressing. The user only sees the 48 bit architecture.
3. Hardware/software interface, i.e. interrupt vectors and I/O ports:
4. Interrupt mechanisms
5. Addressability constraints. These include (a) how references to storage are made, and (b) What storage is available at any one time.
6. Non-Uniform instruction sets, for example an instruction available for one data type but not available for another similar data type.
7. Limited instruction sets. i.e. no conditional procedure call instructions. The PERQ micro-code has a conditional procedure call feature but the micro-operating system (Q-code) did not.

Of all these dependencies, only the last three have any bearing on how difficult it is to implement control structures in a given machine architecture.

These dependencies manifest themselves in various ways:

1. How the boolean expression is evaluated.
2. How procedure addresses are passed to the instruction
3. The lack of suitable instructions to implement these control structures without explicit new instructions, i.e. a conditional procedure call.

For these reasons it was decided to break down the control structures. We split up the implementation of the control structures into a test, a conditional subroutine call, and a subroutine which contains the block of statements to be executed. Using procedure coding conventions in the appropriate architecture, it is possible to implement the control structures with very little effort. However, because users can program any way they wish, it is by no means certain that the control structures will be used. It is still up to the user to make a conscious effort to use these available control instructions.

In some machine architectures, these dependencies result in the inability to effectively implement some of these con-
control structures. For example, on the IBM/370 machine, the Repeat-Until structure is best implemented using a conditional branch instruction. Even if we break the block of statements inside the structure into a procedure call, the loop is still formed by a simple conditional branch instruction.

The advantages of properly implemented control structures outweigh the disadvantages. The major disadvantage in using this method of implementing control structures is that it severely restricts the use of the simple branch instruction. For example, in PASCAL the following program segment would be valid:

```
If A Then Begin
  1 : B
End
Else If C Then Begin
  D
End
Else GoTo 1
```

Although this is generally recognised as very poor programming style, it is still possible to code a program in this manner.

This expression would not be possible using the new instructions because the result would cross procedure boundaries. A GoTo statement can be implemented (as it is in PL/1) so that it can be allowed to cross procedure boundaries, both forward and backward, but the overhead is significant.
Another minor drawback is the overhead, compared to current implementations using branch instructions, involved in entering and exiting a block of code. This overhead is due to the procedure call and exit requirements. In some cases this is not severe because the actual events of entering and exiting a procedure are each contained in one instruction, i.e. in Q-code on the PERQ. The actual severity of this drawback is machine and implementation dependent.

The major advantage is the simplification of looking at the flow of the program at the object code level. Anybody who has tried to debug a high level language program at the machine level will appreciate the ability to associate machine instructions directly with program. This does not however remove all the problems involved with finding the statement in the program associated with the machine code.

Another advantage falls into the domain of assembler language programmers. The introduction of conditional procedure call instructions will allow the user to write modular programs using single machine instructions instead of using convoluted code to accomplish the same thing. For example, the normal way to execute a conditional subroutine call is:

```
Test
Branch to End If False (or true)
Subroutine Call
```

End:
With the new instruction it could be done in the following manner:

Test
Subroutine Call If True (or false)

Although this work does help in looking at some problems in using high level programming concepts at the machine level, some more work could be done in:

1. Conditional procedure calls. This type of instruction would be of great help to programmers who use assemblers.

2. Coupling Source and Object programs in the machine architecture. Although work has been done in many versions of Pascal[22, 23, 24] to provide debugging information when program problems occur, this is still not done with sufficient consistency to help debug user programs. For example Pascal 8000[22] only provides for trace back and variable dumps. This does not allow for break points and interactive debugging, where Pascal/VS[23] and Waterloo Pascal[24] do.

3. More uniform instruction sets would of great use in trying to debug programs i.e. one ADD instruction instead of many, each working on one data type. From the disassembly listings provided in the appendixes, it still is a great chore to follow the logic of a program. With these new instructions, it would be easier to locate control structures.
Appendix A

QCODE DEFINITIONS

The following is a list of the Qcode op-codes taken from the operating system definition. Included is a comparison of the p-code op-code list. Please note that there is not a one to one correspondence. Some functions available in p-code (i.e. sine, cosine etc... and various addressing operations) are not available in Q-Code and conversely some functions in Q-Code (i.e. the raster-op function, store writeable control storage as well as various addressing operations) are not available in P-Code. The four new instructions are also included in this list and have the values 80 through 83. The op-code values are in decimal.

These definitions are reprinted with the permission of PERQ systems.
<table>
<thead>
<tr>
<th>Q-Code</th>
<th>Op-Code</th>
<th>P-Code</th>
<th>Op-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC0</td>
<td>0</td>
<td>SLDC</td>
<td>0</td>
</tr>
<tr>
<td>LDC1</td>
<td>1</td>
<td>SLDC</td>
<td>1</td>
</tr>
<tr>
<td>LDC2</td>
<td>2</td>
<td>SLDC</td>
<td>2</td>
</tr>
<tr>
<td>LDC3</td>
<td>3</td>
<td>SLDC</td>
<td>3</td>
</tr>
<tr>
<td>LDC4</td>
<td>4</td>
<td>SLDC</td>
<td>4</td>
</tr>
<tr>
<td>LDC5</td>
<td>5</td>
<td>SLDC</td>
<td>5</td>
</tr>
<tr>
<td>LDC6</td>
<td>6</td>
<td>SLDC</td>
<td>6</td>
</tr>
<tr>
<td>LDC7</td>
<td>7</td>
<td>SLDC</td>
<td>7</td>
</tr>
<tr>
<td>LDC8</td>
<td>8</td>
<td>SLDC</td>
<td>8</td>
</tr>
<tr>
<td>LDC9</td>
<td>9</td>
<td>SLDC</td>
<td>9</td>
</tr>
<tr>
<td>LDC10</td>
<td>10</td>
<td>SLDC</td>
<td>10</td>
</tr>
<tr>
<td>LDC11</td>
<td>11</td>
<td>SLDC</td>
<td>11</td>
</tr>
<tr>
<td>LDC12</td>
<td>12</td>
<td>SLDC</td>
<td>12</td>
</tr>
<tr>
<td>LDC13</td>
<td>13</td>
<td>SLDC</td>
<td>13</td>
</tr>
<tr>
<td>LDC14</td>
<td>14</td>
<td>SLDC</td>
<td>14</td>
</tr>
<tr>
<td>LDC15</td>
<td>15</td>
<td>SLDC</td>
<td>15</td>
</tr>
<tr>
<td>LDCMO</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDCB</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDCW</td>
<td>18</td>
<td>LDCI</td>
<td>199</td>
</tr>
<tr>
<td>LSA</td>
<td>19</td>
<td>LCA</td>
<td>166</td>
</tr>
<tr>
<td>ROTSHI</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STIND</td>
<td>21</td>
<td>STO</td>
<td>154</td>
</tr>
<tr>
<td>LDCN</td>
<td>22</td>
<td>LDCN</td>
<td>159</td>
</tr>
<tr>
<td>LDB</td>
<td>23</td>
<td>LDB</td>
<td>190</td>
</tr>
<tr>
<td>STB</td>
<td>24</td>
<td>STB</td>
<td>191</td>
</tr>
<tr>
<td>LDCH</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDP</td>
<td>26</td>
<td>LDP</td>
<td>186</td>
</tr>
<tr>
<td>STPF</td>
<td>27</td>
<td>STP</td>
<td>187</td>
</tr>
<tr>
<td>STCH</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXGO</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QAND</td>
<td>30</td>
<td>LAND</td>
<td>132</td>
</tr>
<tr>
<td>QOR</td>
<td>31</td>
<td>LOR</td>
<td>141</td>
</tr>
<tr>
<td>QNOT</td>
<td>32</td>
<td>LNOT</td>
<td>147</td>
</tr>
<tr>
<td>EQUBool</td>
<td>33</td>
<td>EQUBOOL</td>
<td>175</td>
</tr>
<tr>
<td>NEQBool</td>
<td>34</td>
<td>NEQBOOL</td>
<td>183</td>
</tr>
<tr>
<td>LEQBool</td>
<td>35</td>
<td>LEQBOOL</td>
<td>180</td>
</tr>
<tr>
<td>LESBool</td>
<td>36</td>
<td>LESBOOL</td>
<td>181</td>
</tr>
<tr>
<td>GEQBool</td>
<td>37</td>
<td>GEQBOOL</td>
<td>176</td>
</tr>
<tr>
<td>GTRBool</td>
<td>38</td>
<td>GTRBOOL</td>
<td>177</td>
</tr>
<tr>
<td>EQUI</td>
<td>39</td>
<td>EQUI</td>
<td>195</td>
</tr>
<tr>
<td>NEQI</td>
<td>40</td>
<td>NEQI</td>
<td>203</td>
</tr>
<tr>
<td>LEQI</td>
<td>41</td>
<td>LEQI</td>
<td>200</td>
</tr>
<tr>
<td>LESI</td>
<td>42</td>
<td>LESI</td>
<td>201</td>
</tr>
<tr>
<td>GEQI</td>
<td>43</td>
<td>GEQI</td>
<td>196</td>
</tr>
<tr>
<td>GTRI</td>
<td>44</td>
<td>GTRI</td>
<td>197</td>
</tr>
<tr>
<td>UNDF45</td>
<td>45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF46</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF47</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF48</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF49</td>
<td>49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF50</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUStr</td>
<td>51</td>
<td>EQUSTR</td>
<td>175 4</td>
</tr>
</tbody>
</table>

- 51 -
<table>
<thead>
<tr>
<th>NEQStr</th>
<th>52</th>
<th>NEQSTR</th>
<th>183</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQStr</td>
<td>53</td>
<td>LEQSTR</td>
<td>180</td>
<td>4</td>
</tr>
<tr>
<td>LESStr</td>
<td>54</td>
<td>LESSTR</td>
<td>181</td>
<td>4</td>
</tr>
<tr>
<td>GEQStr</td>
<td>55</td>
<td>GEQSTR</td>
<td>176</td>
<td>4</td>
</tr>
<tr>
<td>GTRStr</td>
<td>56</td>
<td>GTRSTR</td>
<td>177</td>
<td>4</td>
</tr>
<tr>
<td>EQUByt</td>
<td>57</td>
<td>EQUBYT</td>
<td>175</td>
<td>10</td>
</tr>
<tr>
<td>NEQByt</td>
<td>58</td>
<td>NEQBYT</td>
<td>183</td>
<td>10</td>
</tr>
<tr>
<td>LEQByt</td>
<td>59</td>
<td>LEQBYT</td>
<td>180</td>
<td>10</td>
</tr>
<tr>
<td>LESByt</td>
<td>60</td>
<td>LESBYT</td>
<td>181</td>
<td>10</td>
</tr>
<tr>
<td>GEOByt</td>
<td>61</td>
<td>GEOBYT</td>
<td>176</td>
<td>10</td>
</tr>
<tr>
<td>GTRByt</td>
<td>62</td>
<td>GTRBYT</td>
<td>177</td>
<td>10</td>
</tr>
<tr>
<td>EQUPowr</td>
<td>63</td>
<td>EQUPOWR</td>
<td>175</td>
<td>8</td>
</tr>
<tr>
<td>NEQPowr</td>
<td>64</td>
<td>NEQPOWR</td>
<td>183</td>
<td>8</td>
</tr>
<tr>
<td>LEQPowr</td>
<td>65</td>
<td>LEQPOWR</td>
<td>180</td>
<td>8</td>
</tr>
<tr>
<td>SGS</td>
<td>66</td>
<td>SGS</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>GEQPowr</td>
<td>67</td>
<td>GEQPOWR</td>
<td>176</td>
<td>8</td>
</tr>
<tr>
<td>SRS</td>
<td>68</td>
<td>SRS</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>EQUWord</td>
<td>69</td>
<td>EQUWORD</td>
<td>175</td>
<td>12</td>
</tr>
<tr>
<td>NEQWord</td>
<td>70</td>
<td>NEQWORD</td>
<td>183</td>
<td>12</td>
</tr>
<tr>
<td>ABI</td>
<td>71</td>
<td>ABI</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>ADI</td>
<td>72</td>
<td>ADI</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>NGI</td>
<td>73</td>
<td>NGI</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>SBI</td>
<td>74</td>
<td>SBI</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>75</td>
<td>MPI</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>DVI</td>
<td>76</td>
<td>DVI</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>MODI</td>
<td>77</td>
<td>MODI</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>CHK</td>
<td>78</td>
<td>CHK</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>UNDF79</td>
<td>79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AIF</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWHILE</td>
<td>81</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREPEAT</td>
<td>82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANOTE</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF84</td>
<td>84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF85</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF86</td>
<td>86</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNDF87</td>
<td>87</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INN</td>
<td>88</td>
<td>INN</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>UNI</td>
<td>89</td>
<td>UNI</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>QINT</td>
<td>90</td>
<td>INT</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>DIF</td>
<td>91</td>
<td>DIF</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>EXITT</td>
<td>92</td>
<td>EXIT</td>
<td>158</td>
<td>4</td>
</tr>
<tr>
<td>NOOP</td>
<td>93</td>
<td>NOP</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>REPL</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPL2</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMS</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MES</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LVRD</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSSN</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XJP</td>
<td>100</td>
<td>XJP</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>PSW</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RASTOP</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRT10</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBLK</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTOFF</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTON</td>
<td>106</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDLB</td>
<td>107</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDLW</td>
<td>108</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL0</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL1</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL2</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL3</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL4</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL5</td>
<td>114</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL6</td>
<td>115</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL7</td>
<td>116</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL8</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL9</td>
<td>118</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL10</td>
<td>119</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL11</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL12</td>
<td>121</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL13</td>
<td>122</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL14</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDL15</td>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLAB</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLAW</td>
<td>126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL8</td>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STLW</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL0</td>
<td>129</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL1</td>
<td>130</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL2</td>
<td>131</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL3</td>
<td>132</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL4</td>
<td>133</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL5</td>
<td>134</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL6</td>
<td>135</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STL7</td>
<td>136</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDOB</td>
<td>137</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDOW</td>
<td>138</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO0</td>
<td>139</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO1</td>
<td>140</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO2</td>
<td>141</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO3</td>
<td>142</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO4</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO5</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO6</td>
<td>145</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO7</td>
<td>146</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO8</td>
<td>147</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO9</td>
<td>148</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO10</td>
<td>149</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO11</td>
<td>150</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO12</td>
<td>151</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO13</td>
<td>152</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO14</td>
<td>153</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDO15</td>
<td>154</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAB</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAW</td>
<td>156</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOB</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOW</td>
<td>158</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST00</td>
<td>159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRO</td>
<td>171</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**- 53 -**
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>STO1</td>
<td>160</td>
</tr>
<tr>
<td>STO2</td>
<td>161</td>
</tr>
<tr>
<td>STO3</td>
<td>162</td>
</tr>
<tr>
<td>STO4</td>
<td>163</td>
</tr>
<tr>
<td>STO5</td>
<td>164</td>
</tr>
<tr>
<td>STO6</td>
<td>165</td>
</tr>
<tr>
<td>STO7</td>
<td>166</td>
</tr>
<tr>
<td>MVBB</td>
<td>167</td>
</tr>
<tr>
<td>MVBW</td>
<td>168</td>
</tr>
<tr>
<td>MOVB</td>
<td>169</td>
</tr>
<tr>
<td>MOVW</td>
<td>170</td>
</tr>
<tr>
<td>INDB</td>
<td>171</td>
</tr>
<tr>
<td>INDW</td>
<td>172</td>
</tr>
<tr>
<td>LDIND</td>
<td>173</td>
</tr>
<tr>
<td>IND0</td>
<td>173</td>
</tr>
<tr>
<td>IND1</td>
<td>174</td>
</tr>
<tr>
<td>IND2</td>
<td>175</td>
</tr>
<tr>
<td>IND3</td>
<td>176</td>
</tr>
<tr>
<td>IND4</td>
<td>177</td>
</tr>
<tr>
<td>IND5</td>
<td>178</td>
</tr>
<tr>
<td>IND6</td>
<td>179</td>
</tr>
<tr>
<td>IND7</td>
<td>180</td>
</tr>
<tr>
<td>LGAWW</td>
<td>181</td>
</tr>
<tr>
<td>STMW</td>
<td>182</td>
</tr>
<tr>
<td>STDW</td>
<td>183</td>
</tr>
<tr>
<td>SAS</td>
<td>184</td>
</tr>
<tr>
<td>ADJ</td>
<td>185</td>
</tr>
<tr>
<td>CALLL</td>
<td>186</td>
</tr>
<tr>
<td>CALLV</td>
<td>187</td>
</tr>
<tr>
<td>ATPB</td>
<td>188</td>
</tr>
<tr>
<td>ATPW</td>
<td>189</td>
</tr>
<tr>
<td>WCS</td>
<td>190</td>
</tr>
<tr>
<td>JCS</td>
<td>191</td>
</tr>
<tr>
<td>LDGB</td>
<td>192</td>
</tr>
<tr>
<td>LDGW</td>
<td>193</td>
</tr>
<tr>
<td>LGAB</td>
<td>194</td>
</tr>
<tr>
<td>LGAW</td>
<td>195</td>
</tr>
<tr>
<td>STGB</td>
<td>196</td>
</tr>
<tr>
<td>STGW</td>
<td>197</td>
</tr>
<tr>
<td>UNDF198</td>
<td>198</td>
</tr>
<tr>
<td>UNDF199</td>
<td>199</td>
</tr>
<tr>
<td>RET</td>
<td>200</td>
</tr>
<tr>
<td>MMS2</td>
<td>201</td>
</tr>
<tr>
<td>MES2</td>
<td>202</td>
</tr>
<tr>
<td>LDTP</td>
<td>203</td>
</tr>
<tr>
<td>JMPB</td>
<td>204</td>
</tr>
<tr>
<td>JMPW</td>
<td>205</td>
</tr>
<tr>
<td>JFB</td>
<td>206</td>
</tr>
<tr>
<td>JFW</td>
<td>207</td>
</tr>
<tr>
<td>JTB</td>
<td>208</td>
</tr>
<tr>
<td>JTW</td>
<td>209</td>
</tr>
<tr>
<td>JEQB</td>
<td>210</td>
</tr>
<tr>
<td>JEQW</td>
<td>211</td>
</tr>
<tr>
<td>JNEB</td>
<td>212</td>
</tr>
</tbody>
</table>

- 54 -
JNEW  213  EFJ  211
IXP   214  IXP  192
LDIB  215  LOD  182
LDIW  216
LIAB  217  LDA  178
LIAW  218
STIB  219  STR  184
STIW  220
IXAB  221
IXAW  222  IXA  164
IXA1  223
IXA2  224
IXA3  225
IXA4  226
TLATE0 227
TLATE1 228
TLATE2 229
EXCH  230
EXCH2 231
INCB  232
INCW  233  INC  162
CALLXB 234
CALLXW 235  CXP  205
LDMC  236  LDC  179
LDDC  237
LDMW  238  LDM  188
LDDW  239
STLATE 240
LINE  241
ENABLE 242
QRAISE 243
LDAP  244
UNDF245 245
UNDF246 246
UNDF247 247
UNDF248 248
UNDF249 249
ROPS  250  { See below for 2nd byte }
INCDDS 251
LOPS  252  { See below for 2nd byte }
KOPS  253  { See below for 2nd byte }
BREAK 254
REFILLOP 255

{ -------------------------------------------------
{ Long Operations - Second byte of LOPS opcode
{ -------------------------------------------------} }
| SBL  | 4 |
| MPL  | 5 |
| DVL  | 6 |
| MODL | 7 |
| ABL  | 8 |
| EQULong | 9 |
| NEQLong | 10 |
| LEQLong | 11 |
| LESLong | 12 |
| GEQLong | 13 |
| GTRLong | 14 |
| LUNUSED | 15 |

```
{ Real Operations - Second byte of ROPS opcode }

| TNC  | 0 |
| FLT  | 1 |
| ADR  | 2 |
| NGR  | 3 |
| SBR  | 4 |
| MPR  | 5 |
| DVR  | 6 |
| RND  | 7 |
| ABR  | 8 |
| EQUReal | 9 |
| NEQReal | 10 |
| LEQReal | 11 |
| LESReal | 12 |
| GEQReal | 13 |
| GTRReal | 14 |
| LUNUSED | 15 |
```

```
{ Spice Kernel Operations - Second byte of KOPS opcode }

| KBLOCK | 15 |
| KUNBLOCK | 14 |
| KSLEEP | 13 |
| KWAKEUP | 12 |
| KREMOVEFROMQUEUE | 11 |
| KADDTOQUEUE | 10 |
| KRESUMEMICROSTATE | 9 |
| KCLOCKTICK | 8 |
| KINITQUEUES | 7 |
| KINTRSRV | 6 |
| KINTROFF | 5 |
| KINTRON | 4 |
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>KSETSOFT</td>
<td>3</td>
</tr>
<tr>
<td>KCLEARSOFT</td>
<td>2</td>
</tr>
<tr>
<td>KCURPROCESS</td>
<td>1</td>
</tr>
<tr>
<td>KUNUSED</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix B
PERQ MICROCODE SOURCE

The implementation of the four new machine instructions required one new routine which is comprised of 54 micro-instructions and 20 micro-instructions added or altered in various old routines. This comprises about 2.5% of 2880 micro-instructions in the micro-operating system. If some parts of the code had simply been duplicated, about 31 more micro-instructions would have been required, bringing the total to 105 micro-instructions added or modified, or 3.6% of the micro-operating system. It is not possible to implement all the new instructions without changing some parts of the micro-operating systems, specifically the common CALL and RETURN routines and the initialisation of the micro-operating system.

The routines that were modified comprise the basic CALL and RETURN mechanisms, the LVRD instruction which was modified to behave as a subroutine instead of a simple machine instruction and the initialisation of the micro-operating system.
Certain components of the PERQ® interpreter micro-code are contained in this appendix. Only those portions which were heavily modified to provide support for the new instructions are included. Those portions which were modified are marked by a '|' to the left of the text. Those portions which were added are marked by a '*' to the left of the text. Certain alterations are not marked because the only change was to add a label to those instructions.

The only changed code not included is the removal of the invalid instruction routine(s), for the new instructions. There is a lot of micro-code, both interpreter and other material, which is not included to keep the appendix short. You will find at the end of this appendix, a list of pages which contain the routines which were either added or modified to implement the new instructions.

N.B. The object code is in octal.

The listing in this appendix is reprinted with the permission of PERQ systems.
Horst Mauersberg, Brian Rosen, Miles Barel
J. P. Strait
rewritten
ca. 1 Jan 80.

Copyright (c) Three Rivers Computer Corporation, 1980.

Abstract:

13 Jan 82  V2.5  W. Hansen
change to: in StkOv (in Perq.Routine.1) to help ProgPlace

31 Dec 81  V2.4  M. Kristofic
Added floating point.

9 Sep 81  V2.3  J. Strait
Fix bugs in double precision—see change history in double precision

14 May 81  V2.2  G. Robertson.
1. Moved IO up to 4400, expanded space for Perq to 2.25K.
2. Added double precision arithmetic operations.
3. Added Spice kernel operations.
4. Added ROM and Line as part of interpreter.

14 Mar 81  V2.1  J. Strait.
1. Begin installing exception handling microcode.
2. Make sure that the SL and RA from the main program are zero, and
   if SL of procedures inside the main program are also zero. This is
   for stack searches for exceptions.
3. Minor bug corrections to stack overflow processing.
4. Bug correction to external calls.

21 Nov 80  V2.0  J. Strait.
Start file.

xxxWord: Multiple word comparisons
**Opcode JMPB.**

**Abstract:**
JMPB is a two byte unconditional jump instruction.

**Instruction:**
JMPB Offset,

**Environment:**
old PC = Byte address + 1 of the Offset operand, PC = UPC + 2

**Result:**
new PC = old PC + Offset,

**Calls:**
AdjustPC.

---

**Opcode JFB, JTB.**

**Abstract:**
JFB and JTB are two byte conditional jump instructions which jump if the value on the top of the expression stack is false or true respectively.

**Instruction:**
JXB Offset

**Environment:**
old PC = Byte address + 1 of the Offset operand, PC = UPC + 2
(tos) = Boolean value.

**Result:**
Stack popped.
If condition met then new PC = old PC + Offset.
If condition not met then new PC = old PC.

**Calls:**
AdjustPC.
Opcape JEQB, JNEB.

Abstract:
JEQB, JNEB are two byte conditional jump instructions which jump if the two values on the top of the expression stack are equal or not equal respectively.

Instruction:
JxB Offset

Environment:
old PC = byte address + 1 of the Offset operand. PC = UCP + 2
(Tos) = Value0.

Jxxx, XJP - Jumps.

File: Perq QCode.4 Perq Microcode Page 7

(Tos-1) = Value1.

Result:
Stack popped twice.
If condition met then new PC = old PC + Offset.
If condition not met then new PC = old PC.

Calls:
AdjustPC.

JMPB1:
Opcode(JMPb), JmpOffset := NextOp;
    if BytesSign Goto(JMPB2); 1 byte offset
    if backward jump
    Goto(AdjuslPC);

JMPB2: JmpOffset := JmpOffset or(377, 377), Goto(AdjuslPC); sign ext

-63-
Opcode JMPW.

Abstract:
JMPW is a three byte unconditional jump instruction.

Instruction:
JMPW LowByteOffset HighByteOffset

Environment:
old PC = Byte address + 1 of the HighByteOffset operand.
PC = UPC + 2 + BPC.

Result:
nPC = old PC + Offset.

Calls:
AdjustPC.

Opcode JFW, JIW.

Abstract:
JFW and JIW are three byte conditional jump instructions which
if the value on the top of the expression stack is false or tr
respectively.

Instruction:
JFW LowByteOffset HighByteOffset

Environment:
old PC = Byte address + 1 of the HighByteOffset operand.
PC = UPC + 2 + BPC.
(Tos) = Boolean value.

Result:
Stack popped.
If condition met then new PC = old PC + Offset.
If condition not met then new PC = old PC.

Calls:
AdjustPC, PuntByte.

Opcode JEQW, JNEW.
Abstract:
JEGW, JNEW are three byte conditional jump instructions which
if the two values on the top of the expression stack are equal
not equal respectively.

Instruction:
JxW LowByteOffset HighByteOffset

Environment:
Jxxx, XJP - Jumps.

File: Perq.QCode.4 Perq Microcode Page 9

old PC = Byte address + 1 of the HighByteOffset operand.
PC = UPC * 2 + BPC.

(Tos) = Value0
(Tos-1) = Value1.

Result:
Stack popped twice.
If condition met then new PC = old PC + Offset.
If condition not met then new PC = old PC.

Calls:
 AdjustPC, PuntByte.

JMPW1:
Opcodes(JMPW1), JmpOffset := NextOp:

1170 310 52 0 1 0 1 0 0 3 17 164 4 3 3513
1171 311 31 0 1 0 1 0 0 3 17 164 4 3 3513
1172 312 31 0 6 0 0 0 0 2 0 160 0 16
1173 313 52 5 2 0 0 1 0 7 3 10 272 0 3 3503

1174 300 0 1 7 1 0 0 4 0 5 306 0 3
1175 71 0 0 6 0 0 0 0 3 17 67 15 3 1170
1176 72 0 0 1 0 0 0 0 3 17 164 4 3 3513
1177 73 0 0 6 0 0 0 0 3 17 325 0 3 1156

1200 270 0 1 7 1 0 0 4 0 5 276 0 3
1201 101 0 0 6 0 0 0 0 3 17 67 10 3 1170
1202 102 0 0 1 0 0 0 0 3 17 164 4 3 3513
1203 103 0 0 6 0 0 0 0 3 17 325 0 3 1156

1204 260 30 0 7 0 1 0 0 0 5 330 0 3
1205 47 0 30 7 0 0 0 12 0 5 306 0 3 1175
1206 250 30 0 7 0 1 0 0 0 5 315 0 3
1207 62 0 30 7 0 0 0 12 0 5 276 0 3 1201

Jxxx, XJP - Jumps.
File: Perq.QCode.4 Perq Microcode Page 10
Opcode XJP

Abstract:
XJP is a variable length instruction that implements the Pascal case statement. It is an n-way branch which chooses the target based on an integer value in some range Low..High. Three word follow the XJP in the code stream, and they must be word aligned. A noise byte is added when the XJP opcode is in a low order by The three words are the minimum index (Low), the maximum index (High), and the self-relative address of the code to be executed when the case selector is outside the range Low..High. A word aligned jump table immediately follows the maximum index, and each word in the table contains a self-relative address.

Instruction:

| XJP | Low                |
|     | High               |
| JumpTable: | Address for Low case |
|         | Address for Low+1 case |
|         | Address for High case |

Environment:

(Tos) = Index = Case selector.

Result:

Stack popped.
If (Index < Low) or (Index > High) then
new PC = JumpTable byte address + 2 * (High-Low).
Otherwise new PC = JumpTable byte address + JumpTable(Index -

Calls:
RefillJmp, VectSrv.

Opcode(XJP), UState, Field(0,4):

```
1210 1154 0 0 5 0 0 0 0 0 2 0 374 0 16
1211 1155 30 1 0 1 0 1 0 14 1 0 341 0 16
1212 1156 16 16 0 0 1 0 10 14 1 0 16 350 0 3
1213 1027 16 3 6 1 0 14 3 15 266 0 3
1214 1111 31 0 3 0 1 0 0 3 17 236 2 1 3331
1215 1112 16 2 6 1 0 0 16 1 16 272 0 3
1216 1105 32 31 7 0 1 0 16 2 0 341 0 16
1217 1106 31 0 7 0 1 0 0 0 5 275 12 3 1225
1220 1107 32 60 6 0 1 0 4 3 15 314 0 3
1221 1063 0 31 3 0 0 0 16 3 15 306 0 3
1222 1071 16 32 6 0 0 0 14 1 16 276.12 3 1224
1223 1072 16 32 6 0 1 0 14 3 15 301 0 3 1227
```

- read BPC
  - round up to word bounda
  - fetch Low
  - fetch High
  - JumpTable word address
  - if IntrPend Call(VectSrv)
  - if Index < Low
  - if Index > High
  - if Index is good off
  - ensure tmp2 is good off
  - offset within JumpTable
  - if Lss Goto(Case2)
  - if Index < Low
  - if Index > High
  - if Lss Goto(Case1)
  - word address of JumpTab
Case Index is out of range. Jump to default.

Jxxx, XJP - Jumps.

File: Perm.OCODE.4 Perm Microcode Page 11

Case1: Md1. If IntrpEnd Call(VectSrv):
        allows mem to finish
        fetch address for default
RightShift(I);

Case2: UPC := UPC - 1. Fetch;

Case3: tmp2 := Md1;
        byte offset
        if jumping backward
        if Jumping backward
        UPC := Shift + UPC. LeftShift(I):
        add word offset to UPC
        byte in word
        word in quad
BPC := tmp1;

Case4: tmp1 := Shift and .7. If IntrapEnd Call(VectSrv):
        word in quad
        word in
        word in
        byte in
        byte in
        set BPC

UPC := UPC + tmp2. Goto(RefillJmp);

UPC := UPC and not 3. Goto(RefillJmp);

UPC is a quad address

Byte offset is negative, sign extension is necessary.

Title CALLx. LVRD, RET, EXIT, EXGO - Calls and returns.

CALLx, LVRD, RET, EXIT, EXGO - Calls and returns.

File: Perm.OCODE.4 Perm Microcode Page 12

Opcode CALLL.

-----------------------------------------------

Abstract:
        CALLL is a two byte routine call instruction. It is used to call
        routines in the current code segment.
Instruction:
        CALLL RoutineNumber

Result:
        New activation record built on memory stack.
        Code state registers saved in new ACB.
        Expression stack saved in new ACB.
        Code state registers updated.

Calls:
        CllSib, RefillJmp, VectSrv.

-----------------------------------------------

Opcode(CALLL). tmp := NextDp:

new RN
tmp7 := GP;
tmp5 := CS, if IntrPend Call(VectSrv);
new GP = old GP
new CS = old CS
Instruction is two byte
new CB = old CB
set up ACB etc.
enter the routine
Goto(RefillJmp);
CALLX, LVRD, RET, EXIT, EXGO - Calls and returns.
File: Perq.QCode.4 Perq Microcode Page 13

Opcode CALLXB.

Abstract:
CALLXB is a three byte routine call instruction. It is used to
routines in an external code segment. The external segment is
identified by an ISN (internal segment number) which is an ind
into the XST (external segment table). The XST maps an ISN in
an XSN (external segment number) and an XGP (external global p

Instruction:
CALLXB ISN RoutineNumber

Result:
New activation record built on memory stack.
Code state registers saved in new ACB.
Expression stack saved in new ACB.
Code state registers updated.

Calls:
Cl1Sub, XSTMap, ChkSeg, RefillJmp, SegFault, VectSrv.

Opcode CALLXW.

Abstract:
CALLXW is a four byte routine call instruction. It is used to
routines in an external code segment. The external segment is
identified by an ISN (internal segment number) which is an ind
into the XST (external segment table). The XST maps an ISN in
an XSN (external segment number) and an XGP (external global p

Instruction:
CALLXW LowByteISN HighByteISN RoutineNumber

Result:
New activation record built on memory stack.
Code state registers saved in new ACB.
Expression stack saved in new ACB.
Code state registers updated.
Opcode(CALLXB). tmp := NextOp;
  tmp1 := 2;
  CallExternal: Call(XSTMap);
  tmp5 := tmp, LoadS(Stk0v);
  Call(ChkSeg);
  tmp6 := tmp, if Odd Goto(SegFault1)
  tmp := NextOp;
  tmp1 := tmp1 + 1;
  CALLX, LVRD, RET, EXIT, EXGO - Calls and returns.
  File: Perg.QCode.4 Perg Microcode Page 14

Opcode(CALLXW). tmp1 := 3. Call(WordParm):
  tmp := Shift + tmp, Goto(CallExternal):
  CALLX, LVRD, RET, EXIT, EXGO - Calls and returns.
  File: Perg.QCode.4 Perg Microcode Page 15

Opcode LVRD.

Abstract:
LVRD is a five byte instruction that builds a variable routine descriptor. This descriptor may be used later in a CALLV (call variable) instruction. The external segment is identified by ISN (internal segment number). An ISN of zero identifies the current segment. The ISN is an index into the XST (external segment table). The XST maps an ISN into an XSN (external segment number) and an XGP (external global pointer).

Instruction:
LVRD LowByteISN HighByteISNRoutineNumber LexicalLevel

Result:
Stack pushed four times.
(Tos..Tos-3) = Variable routine descriptor
(Tos) = XSN = External segment number.
(Tos-1) = XGP = Global link as offset from stack base.
(Tos-2) = new RN = Routine number.
(Tos-3) = new SL = Static link as offset from stack base.

Calls:
WordParm, XSTMap, GetLL, GetSL.

GoLvrD: Opcode(LVRD), Call(WordParm);
tmp := Shift + tmp, Call(XSTMap);  | ISN

tmp2 := NextOp;  | new RN

lop0 := 0;  | SL of top-level routine

lop3 := NextOp - 2;  | new LL - 2

tmp0 := CB, if Led Goto(LVRD2);  | if calling top-level routine

Nop:  | allow placer to do page esca

tmp0 := RN, Call(GetLL);  | get current LL

lop3 := lop3 + 2;  | new LL

lop1 := MD1 - lop3;  | current LL - new LL (typical)

tmp10 := AP, if Gex Call(GetSL);  | if not calling deeper

lop1 := MD1

lop10 := MD1, if Gex Call(GetSL);  | if calling deeper, new SL -

static link

LVRD1:  | routine number

tmp10 := SB, Push;  | global link

lop10 := SB, Push;  | system segment number

tmp0 := 0  | If CRO is zero, normal instr

tmp0 := 0

If Neg Return:

NextInst(0);

1304 1137 40 0 6 1 1 0 1 3 15 241 0 3 1275

LVRD2: tmp10 := 0, Goto(LVRD1);

Perq.Micro = Perq Q-Code Interpreter microcode
VA PA X Y A B W H AL F SF Z CN JP VT
CALLX, LVRD, RET, EXIT, EXGO - Calls and returns.
File: Perq.QCode.4 Perq Microcode Page 16

Opcode CALLV.

Abstract:
CALLV is a one byte routine call instruction. It is used to call routines described by variable routine descriptors.

Instruction:
CALLV

Environment:
(Tos, Tos-3) = Variable routine descriptor:
(Tos) = XSN = External segment number.
(Tos-1) = XGP = Global link as offset from stack base.
(Tos-2) = new RN = Routine number.
(Tos-3) = new SL = Static link as offset from stack base.

Result:
New activation record built on memory stack.
Code state registers saved in new ACB.
Expression stack saved in new ACB after popping four words.
Code state registers updated.

Calls:
ClIV, ChkSeg, RefillJmp, SegFault.
tmp5 := Tos.  if Odd Goto(Call1V2);

if Odd Goto(Call1V2);

tmp6 := tmp, Pop, Load5(Call1V1);

if Odd Goto(Call1V2);

n = tmp, Pop, Load5(Call1V1);

tmp7 := Tos + SB, Pop;

if Odd Goto(Call1V2);

tmp8 := Tos + SB, Pop;

if Odd Goto(Call1V2);

Hold, tmp := Tos, Pop, LeftShift(3);

if Odd Goto(Call1V2);

Hold, tmp := Tos + SB, Pop;

if Odd Goto(Call1V2);

Hold, tmp := Tos + SB, Pop.

if Odd Goto(Call1V2);

Goto(RefillJmp);

if Odd Goto(Call1V2);

! Stack overflow.

Call1V1: Tos := tmp, Push;

! restore expression stack.

Tos := Gp, SB, Push;

Tos := CS, Push, Goto(StkDv);

! Segment fault.

Call1V2: tmp11 := 1, Goto(SegFault1);

! Instruction is one byte

Perq.Micro - Perq O-Code interpreter microcode

CALLx, LVRD, RET, EXIT, EXGO - Calls and returns.

VA PA X Y AB W ALF SF Z CN JP VT

File: Perq.OCode.4 Perq Microcode Page 17

Opcode RET.

Abstract:

RET is a one byte instruction used to return from a routine.

the return address from the ACB is zero, the program counter is

set to the exit point of the routine that is being returned to.

This is used by the EXIT and EXGO opcodes.

Instruction:

RET

Result:

Code state registers restored from old ACB.

Expression stack restored from old ACB.

Old activation record popped from memory stack.

Function result (if any) left on top of memory stack.

Calls:

RetExit, RefillJmp, VectSr, RestoreStack.

 Opcode(RET), AP + ACBRS, Fetch;

Md1 := CS;

tmp5 := Md1, if Neg Goto(Return4); ! if returning to another
Return1: Call(RestoreStack): ! restore expression stack
AP := ACBGL, Fetch;
GP := Md1 + SB, if IntrPend Call(VectSrv); ! global pointer
AP := ACBTL, Fetch;
TP := Md1 + SB;
AP := ACBR, Fetch;
RN := Md1; ! routine number
AP := ACBRA, Fetch;
tmp := Md1, RightShift(1); ! return address
UPC := Shift and not 3, if Eq1 Goto(Return2); ! if return add
UPC := UPC + CD, if IntrPend Call(VectSrv);

! ----- The following line is a good place to set a breakpoint. RN, C
! ----- UPC have been restored (though BPC hasn't).
BPC := tmp and 7, Goto(Return3):

! In the middle of an exit sequence.

Return2: Call(ReturnExit):

Return3: AP := ACBDL, Fetch; ! activation pointer
AP := Md1 + SB;
AP := ACBDL, Fetch;
LP := Md1 + SB;
Goto(Return1Jump); ! enter routine

! Cross segment return.
CALLs, LVRD, RET, EXIT, EXGO - Calls and returns.
File: PerqQCode, Page 18

Return4: tmp5 + tmp5, Fetch2; ! check residence
tmp1 := CB; ! save old CB just in case
CB := Md1 + 376;
CB := Mdxx or CB, if Odd Goto(Return5); ! if not resident
CS := tmp5, Goto(Return1); ! code segment

! Segment fault,

Return5: CB := tmp1; ! restore CB
tmp11 := 1, Goto(SegFault1);

*******
Opcode(RET), AP := 4, Fetch4;
Tos := Md1, Push;
tmp5 := Md1;
tmp6 := Md1;
tmp7 := Md1;
tmp5 := CS;
tmp10 := CB, if Neg Goto(Return6); ! t3 if cross segment

Return1: TP := Tos + SB, if IntrPend Call(VectSrv);
RN := tmp7; !
AP := Md1, Fetch2;
tmp4 := AP, Pop;

- 72 -
BPC := tmp6 and 7;
AP := Mdl + SB;
GP := Mdl + SB;
tmp5, RightShift(1);
UPC := Shift and not 3.

if Eqi Goto(Return2):
    UPC := UPC + CB;
    Goto(Return3);

In the middle of an exit sequence.

Return2: Call(ReturnExit);

Return3: AP + ACBLP, Fetch;
LP := Mdl + SB;
tmp4 + ACBStackSize, Fetch;
\text{t3 address+1 of las}
tmp4 := tmp4 + tmp3,
\text{t3 address+1 of las}
if Eqi Goto(ResetNote); \text{if stack empty}
else keep address

Note1 := tmp4;

Return4: tmp4 := tmp4 - 1, Fetch;
tmp3 := tmp3 - 1, Push.
\text{if IntrPend Goto(Return5):}
Tos := Mdl, if Gtr Goto(Return4):
\text{t1}
tmp4 := Note1;
\text{copy address of spe}
ResetNote: tmp4, Fetch;
Note1 := Mdl;
\text{restore the special}
Goto(Refill);\text{and jump}

Perq, Micro - Perq Q-Code interpreter macrocode
\text{CALLX, LVRO, RET, EXIT, EXGO - Calls and returns.}
File: Perq, QCode.4 - Perq Microcode Page 19

Interrupt while restoring the stack.

Return5: tmp4 := tmp4 + 1, Call(Vec[Srv]);
\text{-Pop, tmp3 := tmp3 + 1, Goto(Return4):
\text{Cross segment return.}

(******
Return6: tmp5 + tmp5, Fetch2;
\text{check residence}
CB := Mdl and not 376;
\text{CB := Mdx or CB, if Odd Goto(Return7): if not resident}
CS := tmp5. Goto(Return1);
\text{code segment}
******)

Return6: tmp := tmp5, Call(ChkSeg);
\text{find new code base}
\text{CB := tmp, if Odd Goto(Return7): if not resident}
CS := tmp5, Goto(Return1);
\text{code segment}

Segment fault.
that the RET opcode treats a return address of zero as an exit request. The ACB that returns to the target routine has its return address set to the target address.

The assumption is made that the segment and routine numbers do specify the current procedure.

**Instruction:**
- EXGO LowByteISN HighByteISN:
  - RoutineNumber
  - LowByteAddress HighByteAddress

**Result:**
- PC (UPC, BPC) = Exit point of current procedure.
- ACBs modified as described in abstract.

**Calls:**
- ExSub, RetExit, RefillJump, WordParm.

---

OpExGO, Call(ExSub):
- Call(WordParm):
  - tmp := Shift + tmp;
  - tmp2 := ACBRA, Store;
  - tmp, Call(ReExit);
  - Goto(RefillJump);

CALLx, LVRD, RET, EXIT, EXGO - Calls and returns.

File: Perq.Micro Perq Microcode Page 22

$NOLIST
Floating point arithmetic operators.

File: Perq.Micro Perq Microcode Page 23

$LIST
Floating point arithmetic operators.

File: Perq.Routine.1 Perq Microcode Page 24

$Include Perq.Routine.1

! 13 Jan 82 WJH change to ; in StkOV

$Title Addressing routines.

File: Perq.Routine.1 Perq Microcode Page 25

Addressing routines.

!!! Abstract:
ChkSeg checks a segment number for residency and returns its base address (if resident).

Environment:
  tmp = Segment number.

Result:
  tmp = Base address.
  tmp and ALU result are Odd if non-resident.

[**]**
ChkSeg: tmp + tmp, Fetch2;
  tmp := Md1 and not 378;
  tmp := Md0 or tmp.
Return;
        ! fetch segment table entry
        ! base address and reside

    *****)
ChkSeg: tmp + tmp, Fetch2;
  tmp12 := tmp;
  Tos := tmp := Md1. Push;
  tmp := Md0 or tmp.
  If Odd Goto(ChkSeg1);  ! if not resident
  If not resident
  tmp := tmp and not 377;
  tmp12 + tmp12. Store;
ChkSeg1: Tos or 4, Pop. Return;
        ! clear the flags
        ! set RecentlyUsed in SAT

Perq Micro - Perq Q-Code Interpreter microcode
VA PA X Y A B W H AL F SF Z CN JP VF

Addressing routines.
File: Perq Routine 1 Perq Microcode Page 26

Routine GetGP.

Abstract:
  Get the global pointer for an external segment. The input is the
  internal segment number, and the output is a global pointer.
  An internal segment number of zero is used to mean the current
  segment.

Environment:
  tmp2 = Internal segment number.

Result:
  tmp2 = External global pointer.

Calls:
  VectSrv.

2700 2636 0 0 6 0 0 0 0 0 3 17 236 0 1 3331
GetGP2: Call(VectSrv);
        ! serve an interrupt
        ! let placer make two gro
2701 2637 0 0 6 0 0 0 0 3 17 372 0 3 -
          GetGP: tmp2, if Intrapend Goto(GetGP2);
          tmp2 := tmp2 + tmp2, if Neg Goto(GetGP1);
2702 5 32 0 6 0 0 0 0 3 12 141 2 3 2700
2703 6 32 32 6 0 1 0 14 3 12 136 10 3 2700
tmp2 := GP, Return;  
  ! same segment as current 

GetGP1: GP = tmp2, Fetch:
    tmp2 := Mdl + SB, Return;

Addressing routines.
File: Perq.Routine.1  Perq Microcode Page 27

Routine GetLL.

Abstract:
GetLL gets the lexical level of a routine given its number and 
base address. The code segment must be resident.

Environment:
    tmp4 = Code base address as a physical address.
    tmp10 = Routine number.

Result:
    tmp1 = Address of LL field.
    LL word of the routine descriptor fetched--it may be read on M

Calls:
    VectSrv.

GetLL: tmp4, Fetch;
    tmp10, LeftShift(3);
    tmp1 := Shift + tmp4;
    tmp1 := tmp1 + RDLL;
    tmp1 := Mdl + tmp1, If IntrPend Call(VectSrv);
    tmp1 := Offset of dictionary entry
    tmp1 := Offset of LL field
    tmp1 := Fetch, Return;

Addressing routines.
File: Perq.Routine.1  Perq Microcode Page 28

Routine GetLP.

Abstract:
Get the local pointer for another activation record. The input
is an offset in static nesting, and the output is a local poin

Environment:
    tmp2 = AP.
    tmp3 = Offset in static nesting.

Result:
The memory word containing the desired local pointer is fetched in the instruction which returns.

**Calls:**

- VectSrv

---

### Details:

**GetLP1:**

GetLP1: \( \text{tmp3} := \text{tmp3} + 1 \), Call(VectSrv);
- `GetLP:`
  - `GetLP1:`
  - `GetLP2:`
  - `GetLP3:`

---

### Addressing routines.

**File:** Perq.Routine.1  Perq Microcode  Page  29

**Routine GetSL:**

**Abstract:**

GetSL gets the static link of a procedure that is a lexical level away. Thus specifying zero gets the static link for a procedure that is at the same level as the starting one.

**Environment:**

- \( \text{tmp10} = \text{Starting AP} \)

**Result:**

- \( \text{tmp10} = \text{Desired SL} \)

**Calls:**

- VectSrv

---

### Details:

**GetSL:**

GetSL: \( \text{tmp10}, \text{Fetch} \), \( \text{tmp10} := \text{tmp10} + 1 \); \( \text{tmp10} := \text{Mdl} + \text{SB} \), \( \text{if} \ \text{GetGoto} \) Goto\( \text{GetSL} \); \( \text{if not there yet} \) \( \text{Return} \);

---

### Addressing routines.

**File:** Perq.Routine.1  Perq Microcode  Page  30

**Routine XSTMap:**

**Abstract:**

XSTMap maps an internal segment number (ISN) into an external segment number (SSH) and an external global pointer (XGP). An
ISN of zero is taken to mean the current segment.

Environment:
  tmp = ISN.

Result:
  tmp = XSN.
  tmp7 = XGP.

Calls:
  VectSrv.

```
xSTMap: tmp, LeftShift(1);
  tmp := Shift, if Eq1 Goto(xSTMap1);  // if current segment dest
  GP := tmp, Fetch2;
  tmp7 := Md1 + SB;
  tmp := Md1, Return;
  XGP

! ISN = 0 means XSN = CS, XGP = GP.
```

```
xSTMap1: tmp := CS, if IntrPend Call(VectSrv);
  tmp7 := GP, Return;
```

$Title Boolean routines.

Perq Micro - Perq O-Code Interpreter microcode

```
Perq Micro - Perq O-Code Interpreter microcode
VA PA X Y A B W H AL F SF Z CN JP VT
```

Boolean routines.
File: Perq.Routine.1 Perq Microcode Page 31

```
Routine SetFalse.

Abstract:
SetFalse sets the top of the expression stack to false. SetFalse is not called, it is jumped to. It exits via a NextInst.
```

```
Environment:
(Tos) = Anything.

Result:
(Tos) = False.
```

```
2735 2670 0 0 6 1 0 0 1 0 3 377 0 2
SetFalse: Tos := 0, NextInst(0);
```

Perq Micro - Perq O-Code Interpreter microcode

```
Perq Micro - Perq O-Code Interpreter microcode
VA PA X Y A B W H AL F SF Z CN JP VT
```

Boolean routines.
File: Perq.Routine.1 Perq Microcode Page 32

- 79 -
Routine SetTrue.

Abstract:
SetTrue sets the top of the expression stack to false. SetTrue is not called, it is jumped to. It exits via a NextInst.

Environment:
(Tos) = Anything.

Result:
(Tos) = True.

SetTrue: Tos := 1, NextInst(0);

$Title Byte array and string routines.

PerPage, Micro - Perq Q-Code Interpreter microcode

Byte array and string routines.

File: Perq.Routine.1 Perq Microcode Page 33

Routine GetStringIndex.

Abstract:
GetStringIndex gets an index into a string variable and checks against the dynamic length of the string. If the index is out of range, GetStringIndex causes an ErrInxCase error. If the index is in range, GetStringIndex fetches the correct word and returns.

Environment:
(Tos) = Byte offset.
(Tos-1) = Word address as offset from stack base.

Result:
Stack popped.
tmp4 = Physical word address of the beginning of the string.
tmpl = Word offset within the string.
Word containing character fetched and readable on Mdi.

Calls:
ChkOvr, VectSrv.

GetStringIndex: tmp := Tos and All0nes, Pop, RightShift(1); ! by
mpl := Shift, if IntrPend Call(VectSrv);! word offset
Tos + SB. Fetch:
Routine GetSrcDst.

Abstract:
GetSrcDst gets a source and destination byte pointer from the expression stack. It also pre-fetches the first bytes referred by the two byte pointers. This is done to get the length byte, for string operations. The result conditions of GetSrcDst are precisely the environment for GetSrc, GetDst, and PutDst.

Environment:
(Tos) = Source byte offset.
(Tos-1) = Source address as offset from stack base.
(Tos-2) = Destination byte offset.
(Tos-3) = Destination address as offset from stack base.

Result:
Stack popped four times.
Src = Source word physical address (Address + ByteOffset div 2
Srclsb = Least significant bit of the source byte address.
SrcWord = First source word.
SrcByte = First source byte.
Dst = Destination word physical address (Address + ByteOffset
Dstlsb = Least significant bit of the destination byte address.
DstWord = First destination word.
DstByte = First destination byte.

Calls:
VecSrv.

GetSrcDst: tmp3 := Tos and All1Ones, Pop, RightShift(1);  \ get  Src byte
Src := Shift + SB, If InPendCall(VecSrv);
Src := Tos + Src, Fetch;  \ Src word address
Srclsb := tmp3, Pop;  \ Src least significant bit
if Odd'Gotod(GetS02);  \ if first byte is upper
SrcWord := Md1;  \ 1st Src word
SrcByte := Md1 and 377;
GetS01: tmp3 := Tos and All1Ones, Pop, RightShift(1);  \ get  Dst byte
Dst := Shift + SB, If InPendCall(VecSrv);
Dst := Tos + Dst, Fetch;  \ Dst word address
Dstlsb := tmp3, Pop;  \ Dst least significant bit
if Odd'Gotod(GetS031);  \ if first byte is upper
DstWord := Md1;  \ 1st Dst word
- DstByte := Mdl and 377. Return; 1st Dst byte
- GetSD2: SrcWord := Mdl, Field(0,10); 1st Src word
- SrcByte := Shift, Goto(GetSD1); 1st Src byte
- GetSD3: DstWord := Mdl, Field(0,10); 1st Dst word
- DstByte := Shift, Return; 1st Dst byte

Byte array and string routines.
File: Perq.Routine.1 Perq Microcode Page 35

! Routine GetSrc.

! Abstract:
GetSrc gets the next source byte from a byte array or a string

! Environment:
Src = Source word physical address.
SrcLsb := Least significant bit of the source byte address.
SrcWord := Current source word.

! Result:
SrcWord = Current source word.
SrcByte := Current source byte.
Source address (Src and SrcLsb) advanced to next byte.

! Calls:
VectSrv.

GetSrc := SrcLsb;
if Odd Goto(GetSrc2);
if Upper Byte Goto(GetSrc);
if IntrPend Goto(GetSrc3);
GetSrc1 := SrcFetch;
SrcLsb := 1, if IntrPend Goto(GetSrc3);
SrcWord := Mdl;
SrcByte := Mdl and 377, Return;
GetSrc2 := RightShift(10);
SrcByte := Shift and 377;
Src := Src + 1, if IntrPend Call(VectSrv);
SrcLsb := 0, Return;

Serve an interrupt.

GetSrc3 := Call(VectSrv);
Goto(GetSrc1);

Byte array and string routines.
File: Perq.Routine.1 Perq Microcode Page 36

- 82 -
Routine GetDst.

Abstract:
GetDst gets the next destination byte from a byte array or a string.

Environment:
Dst = Destination word physical address.
DstLsb = Least significant bit of the destination byte address
DstWord = Current destination word.

Result:
DstWord = Current destination word with current byte removed.
DstByte = Current destination byte.
Destination address (Dst and DstLsb) advanced to next byte.

Calls:
VectSrv.

GetDst: DstLsb:
if Odd goto(GetDst2);
if upper byte
get a new word
GetDst1: Dst. Fetch:
DstLsb := 1, if IntrPend goto(GetDst3);
next byte is upper
DstWord := Mdi and not 377;
current word
DstByte := Mdi and 377. Return;
current byte
GetDst2: DstWord, RightShift(10):
get upper byte from curr
Dstbyte := Shift and 377;
current byte
DstWord := DstWord and 377;
current word
Dst := Dst + 1, if IntrPend Call(VectSrv), advance to next
DstLsb := 0. Return;
next byte is lower byte
Serve an interrupt.

GetDst3: Call(VectSrv):
Goto(GetDst1);

Perq.Micro - Perq O-Code Interpreter microcode Byte array and string routines.

Routine PutsRec.
Result:
DstWord = Current destination word with source byte added.

Calls:
- VectSrv.

PutDst:
- DstLab:
  If Odd Goto(PutDst1):
  SrcByte. LeftShift(10):
  DstWord := Shift or DstWord,
  If InPend Call(VectSrv);
  Dst - 1, Store:
  DstWord, Return;

PutDst1:
- DstWord := DstWord or SrcByte, Return;

PutDst:
- DstWord := DstWord or SrcByte, Return;

Byte array and string routines.

Routine BytCmp.

Abstract:
BytCmp compares two byte arrays. After the call to BytCmp, the
Neq, Leq, Lss, Geq, and Gtr condition codes can be used to che
the results of the comparison.

Environment:
- Next byte in the opcode/operand stream = Length.
  If Length <> 0:
    (Tos) = Byte offset for ByteArray0.
    (Tos-1) = Word address of ByteArray0.
    (Tos-2) = Byte offset for ByteArray1.
    (Tos-3) = Word address of ByteArray1.
  If Length = 0:
    (Tos) = Length of byte arrays.
    (Tos-1) = Byte offset for ByteArray0.
    (Tos-2) = Word address of ByteArray0.
    (Tos-3) = Byte offset for ByteArray1.
    (Tos-4) = Word address of ByteArray1.

Result:
Length removed from opcode/operand stream.
- If Length <> 0:
  Stack popped four times.
- If Length = 0:
  Stack popped five times.
  ALU result := ByteArray0 compared to ByteArray1.

Calls:
GetSrcDst, GetSrc, GetDst, VectSrv.

BytCmp: tmp2 := NextOp;
     | get length
     | if non-zero length byte
     | get length, from express
     | get byte pointers
     | enter comparison loop
BytCmp2: if Lss Goto(BytCmp4).
     | if done and all equal
     | dstbyte - srcbyte; i.e., compare bytes
     | if done and not equal
BytCmp3: Call(GetSrcDst);
     | get next byte
     | get next byte
     | count byte
     | Byte arrays are equal.
BytCmp4: 0, Return;
     | return Eq1
     | Byte arrays are not equal, return with condition codes set for
     | byte comparison.
BytCmp5: dstByte - srcByte, Return;

Perm. Micro - Perm Q-Code Interpreter microcode
VA    PA    X    Y    A    B    W    H    AL    F    SF    Z    CN    JP    VT

Byte array and string routines.
File: Perm.Route1.1 Perm Microcode Page 39

Routine StrCmp.

Abstract:
StrCmp compares two strings. After the call to StrCmp, the Eq
Ned, Leq, Lss, Geq, and Gtr condition codes can be used to che
the results of the comparison. The strings must be word align

Environment:
(Tos) = Byte offset for String0.
(Tos-1) = Word address of String0.
(Tos-2) = Byte offset for String1.
(Tos-3) = Word address, pf, String1.

Result:
Stack popped three times.
ALU result = String0 compared with String1.

Calls:
GetSrcDst, GetSrc, GetDst, VectSrv.

3030 3061 32 0 1 0 1 0 0 3 17 164 4 3 3513
3031 3062 0 0 6 0 0 0 0 3 11 321 10 3 3033
3032 3063 32 0 7 0 1 0 0 5 321 0 3 3033
3033 3066 0 0 6 0 0 0 0 3 12 100 0 1 2746
3034 3057 0 0 6 0 0 0 0 3 12 17 0 3 3033
3035 2756 66 56 6 0 0 0 0 16 3 12 14 12 3 3042
3036 2757 0 0 6 0 0 0 0 3 12 13 10 3 3043
3037 2760 0 0 6 0 0 0 0 3 12 61 0 1 2770
3040 2761 0 0 6 0 0 0 0 3 12 45 0 1 3004
3041 2762 32 1 6 1 1 0 16 3 12 21 0 3 3035
3042 2763 0 0 6 1 0 0 1 0 0 0 0 12
3043 2764 66 56 6 0 0 0 16 0 0 0 0 12
3044 2765 0 0 6 0 0 0 0 3 12 100 0 1 2746

StrCmp: Call(GetSrcDst);
     | get string pointers
3045 2766 55 1 6 1 1 0 1 2 3 12 10 0 3 -
3046 2767 65 1 8 1 1 0 1 2 3 11 310 0 3 -
3047 3067 66 56 6 0 0 0 1 6 0 4 312 0 3 -
3050 3065 30 56 6 0 1 0 1 3 12 7 17 3 3052
3051 3066 30 66 6 0 1 0 1 3 12 7 0 3 -

SclSib := SclSib xor 1;  ; skip length bytes
DstSib := DstSib xor 1;
Tos := dstByte - srcByte. Push;  ; compare lengths, save d
tmp := srcByte, if G eq Goto(StrCmp1);  ; if src is shorter
tmp := dstByte;

Compare the strings, tmp is length of shorter string.

StrCmp1: if EqI Goto(StrCmp3);  ; if strings equal up to
Call(GetSrc);  ; get next character
Call(GetDst);  ; get next character
dstByte - srcByte, if IntPend Goto(StrCmp5);

StrCmp2: if Neq Goto(StrCmp4);  ; if characters are not e
tmp := tmp - 1, Goto(StrCmp1);  ; count the character

Strings are equal up to length of shorter string, return with
condition codes set for length comparison.

StrCmp3: Tos. Return;

Strings are not equal, return with condition codes set for cha
comparison.

StrCmp4: dstByte - srcByte. Return;

Serve an Interrupt.

StrCmp5: Call(VectSrv);  ; dstbyte - srcbyte. Goto(StrCmp2):

Perq Micro - Perq Q-Code Interpreter microcode
VA PA X Y ABWHAL PSF Z CN JP VT
Byte array and string routines.
File: Perq.Routine.1 Perq Microcode Page 40

$Title Call and return routines.

Perq Micro - Perq Q-Code Interpreter microcode
VA PA X Y ABW HAL PSF Z CN JP VT
Call and return routines.
File: Perq.Routine.1 Perq Microcode Page 41

Routine Cl1Sub.

Abstract:
Cl1Sub is a common routine used by the call opcodes to build t
activation record and set up the new pointers (GP, AP, TP, etc

Environment:
tmp := New routine number.
tmp5 := New code segment number.
tmp6 := New code base.
tmp7 := New global pointer.
tmp11 := Length of call instruction (for PCB backup).
S = Routine to jump to on stack overflow.

Result:
New activation record built.
Code state pointers saved in ACB.
Expression stack saved in ACB.
Code state pointers updated.

Calls:
GetLL, GetSL, SvStk, Vectsrv, S.

---------------------------------------------------------------------

Routine C11V.

Abstract:
C11V is a common routine used to make a call to a routine desc
by a variable routine descriptor. C11V builds the new activat
record and set up the new pointers (GP, AP, TP, etc.).

Environment:
tmp = New routine number.
tmp5 = New code segment number.
tmp6 = New code base.
tmp7 = New global pointer.
tmp10 = New static link.
tmp11 = Length of call instruction (for PDBbackup).
S = Routine to jump to on stack overflow.
First half of routine dictionary entry fetched and readable on

Result:
New activation record built.
Code state pointers saved in ACB.
Expression stack saved in ACB.
Code state pointers updated.

Calls:
SvStk, Vectsrv, S.

---------------------------------------------------------------------

C11Sub:
Call and return routines.
File: Perq.Routine.1 Perq Microcode Page 42

Get the new static link.

\[
\begin{align*}
\text{tmp4} & := \text{tmp6}; \\
\text{tmp10} & := \text{tmp}, \text{Call(GetLL)}; \\
\text{tmp2} & := \text{Md1} - 2; \\
\text{tmp3} & := \text{tmp1}, \text{if Leq Goto(C11Sub3)}; \\
\text{tmp4} & := \text{CB}; \\
\end{align*}
\]

new code base (for GetL
get new lexical level
save address of LL in R
if calling a top-level
current codebase
3136  3141  17  4  6  1  1  0  1  4  1  1  7  2  0  0  3  -  
3137  3137  0  0  6  1  0  0  1  3  1  1  235  0  3  -  
3140  3142  6  36  6  0  1  0  1  3  1  1  234  0  3  -  
3141  3143  3  32  6  0  1  0  1  3  1  1  230  0  3  -  

TP := TP + 4, Store;  
O;  
CB := tmp0;  
AP := tmp2;  

Save the expression stack.  
tmp := not O, if IntrPend Call(VectSrv);  1 - 1;  
TP := TP + 1, Call(SaveStack);  
TP := tmp, Store;  
tmp, if IntrPend Call(VectSrv);  
TP := TP + 1, Store;  
Note1;  
Note1 := 0;  
GP := tmp7;  

Set up new PC.  

The following instruction is a good place to set a breakpoint.  
RN, and UPC have their new values, although BPC doesn't yet.  

BP0 := tmp4 and 7;  
tmp4 and now 7, RightShift(1);  
UPC := Shift + CB, Return;  
quad program counter  

Set SL for top-level routines: LL <= 2.  

C11Sub3: tmp10 := SB, Goto(C11Sub1);  
static link is not used  

Signal a stack overflow. Restore Both stacks.  

Make sure abort goes ok  

Perq Micro - Perq Q-Code Interpreter microcode  
VA PA X Y A B W H AL F SF Z CN JP VT  

Call and return routines.  
File: Perq.Routine.1  Perq Microcode Page 44  

Routine SaveStack.  

Abstract:  
SaveStack saves the expression stack in the activation control  

Environment:  
TP = First word address of the saved stack (length word + 1).  
tmp = -1.  

Result:  
Expression stack pushed onto memory stack.  
tmp = Number of saved words.
Call and return routines.

File: Perq.Routine.1  Perq Microcode  Page 45

----------

Routine RestoreStack.

Abstract:
RestoreStack restores the expression stack from the ACB prior returning from a routine.

Environment:
Expression stack empty.

Result:
Expression stack restored.

Calls:
Vector.

RestoreStack: AP + ACBStackSize, Fetch:
  tmp1 := AP;
  tmp1 := tmp1 + ACBSaveStack;
  tmp2 := Md1;
  tmp1 := tmp1 + tmp2, if Eq1 Return;  \(\) number of saved words 0
  tmp1 := tmp1 + tmp2, if Eq1 Return;  \(\) if no words to restore
Restore1: tmp1 := tmp1 - 1, Fetch:
  tmp2 := tmp2 - 1, Push, if IntrPend Goto(Restore2);
  TOS := Md1, if Gtr Goto(Restore1);  \(\) if more words to restore
  Return;

  Serve an interrupt.

Restore2: tmp1 := tmp1 + 1, Call(VectSrv);
  Pop, tmp2 := tmp2 + 1, Goto(Restore1);

----------

Call and return routines.

File: Perq.Routine.1  Perq Microcode  Page 46

----------

Routine RetExit.

----------

Abstract:
RetExi is used when a RETURN instruction finds that the return address is zero. This means that the return address is the exit point of the routine.

Résult:
PC set to exit point of current routine.

Calls:
VectSrv.

RetExi: CB, Fetch:
Hold, RN, LeftShift(3);
Hold, tmp1 := Shift + CB;
Hold, tmp1 := tmp1 + RDEXIT;
Hold, Md1 + tmp1, Fetch;
BPC := Md1 and 7, RightShift(1);
Md1 and not 7;
UPC := Shifl + CB, Return;

fetch dictionary address
fetch exit word in dict
set BPC and UPC

Call and return routines.
File: Perq.Routine.1 Perq Microcode Page 47

Routine ExSub.

Abstract:
ExSub is used by EXIT and EXGO to get the target segment number from the opcode/operand stream and set return addresses to zero in the memory stack.

Environment:
Segment and routine numbers are in the opcode/operand stream;
tmp2 = Target AP.

Result:
Appropriate return addresses set to zero.

Calls:
WordParm, XSTMap, VectSrv.

ExSub: Call(WordParm);
tmp := Shift + tmp, Call(XSTMap);
tmp1 := NextOP;
tmp = CS;
tmp2 := AP, if Neq Goto(ExSub1);
tmp1 := RN;
if Eq1 Return;
if different code segment
if current routine
set a return address to
### Error processing routines.

File: Perq.Routine.1  Perq Microcode  Page 48

#### Routine ChkOvr.

**Abstract:**
ChkOvr signals a ErrlnxCase error. ChkOvr is not called, rather is jumped to. It exits to RunErrorO.

**Result:**
tmp2 = ErrlnxCase.

**Calls:**
RunError1.

#### ChkOvr:
tmp2 := ErrlnxCase. Goto(RunErrorO).

---

File: Perq.Routine.1  Perq Microcode  Page 49

#### Routine ChkStk.

**Abstract:**
ChkStk checks to be sure that N words can be pushed onto the stack without overflowing. If the words will not fit, ChkStk to address in the 2910's S register.

**Environment:**
tmp10 = Number of words.
S = Address of stack overflow handler.

**Result:**
tmp10 unchanged.
Calls:

- VectServ.

----------

3217  13  40  17  6  0  1  0  14  3  11  155  0  3  3221
3220  3221  0  0  6  0  0  0  0  3  17  236  0  1  3331
3221  3222  13  40  6  0  0  0  0  16  3  11  156  2  3  3220
3222  3223  40  17  6  0  1  0  16  3  11  152  5  3  3224
3223  3224  0  0  6  0  0  0  0  0  0  0  0  0  0  3224

ChkStk: tmp10 := tmp10 + TP, Goto(ChkStk2);
ChkStk1: Call(VectServ);
ChkStk2: SL := tmp10, If IntrPend Goto(ChkStk1);
ChkStk3: Call(VectServ);
ChkStk4: SL := tmp10, If IntrPend Goto(ChkStk1);
Return:

Serve an interrupt if there's room, so return.

No room, signal error by jumping to S.

ChkStk3: GotoS;

Perq.Micro - Perq Q-Code Interpreter microcode

Error processing routines.

File: PerqRoutine.1  Perq Microcode Page  50

Abstract:
ErrCall initiates the raising of an exception by calling routine RNRaise in segment ExcCS (procedure Raise in module Except).
The caller pushes parameters to the exception (if any) onto the memory stack before calling ErrCall. ErrCall is jumped to and exits to RefillJmp. If the Except module has not been initial.
is non-resident, or a stack overflow happens while calling Raise, transfer to busted.

Environment:
- tmp3 = Parametersize = Number of words of parameters.
- ExcCS = Segment number of the Except module.
- ExcGP = Global pointer of the Except module.
- tmp1 = Segment number of the exception.
- tmp2 = Routine number of the exception.

Result:
- Micro level call stack cleared.
- Four words pushed on the memory stack.
- Memory(TP - 0) = PStart = TP - 3 - Parametersize.
- Memory(TP - 1) = PEnd = TP - 3.
- Memory(TP - 2) = ER = tmp2.
- Memory(TP - 3) = ES = tmp1.
- tmp10 unchanged.
- ExcCS unchanged.
- ExcGP unchanged.
- tmp1 unchanged.

Calls:
- ChkSeg, Cl1Sub, RefillJmp, VectServ.
Design:
If the Except module has not been initialized, Execute = 0. Since segment 0 is guaranteed to be non-resident, we need not make a separate check for Execute = 0.

ErrCall: TP := TP + 1, Store:
  tmp1 := ThreeWayBranch(0);
  Tos := TP - SB, Push;
  TP := TP + 1, Store;
  tmp2 := ThreeWayBranch(0);
  TP := TP + 1, Store;
  Tos := tmp3 := ThreeWayBranch(0);
  TP := TP + 1, Store;
  tmp5 := Execute, ThreeWayBranch(0);
  tmp := tmp5, Call(ChkSeg);
  tmp6 := tmp, if Odd Goto(Busted);
  tmp := RRRaise, LoadS(Busted);
  tmp7 := Execute, ThreeWayBranch(0);
  tmp := tmp7 + SB, Call(Subr);
  Goto(Ret1).jmp;

Error processing routines.
File: Perq.Routine.1 Perq Microcode Page 51

Error processing routines.
File: Perq.Routine.1 Perq Microcode Page 52

Routine PCBackup:

Abstract:
PCBackup sets the Q-code program counter back N bytes. This is used when a recoverable error is detected (e.g. segment fault). The PC is set back in order that the instruction can be re-executed after the error condition is cleared.

Environment:
  tmp11 = Number of bytes.

Result:
  tmp = New program counter as byte offset from code base.
  UPC set back.
  BPC set back.

Calls:
  VectSrv.
PCBackup: tmp := UState and 17;
UPC, LeftShift(1);
tmp := Shift + tmp, If IntrPend Call(ExitSrv); i full byte PC
tmp := tmp1, RightShift(1); i new byte program counter
UPC := Shift and not 3;
BPC := tmp and 7, Return;

Error processing routines.
File: Perq.Routine.1 Perq Microcode Page 53

Routine RunError, RunErrorO.

Abstract:
RunError is called when the microcode wants to raise an except
The caller pushes parameters to the exception (if any) onto the memory stack before calling RunError. RunError is jumped to a exit to ErrCall. The variant of RunError that is called depends on how many words of parameters were pushed on the memory stack.
RunErrorO is called when 0 words were pushed. If parameters were pushed onto the memory stack, RunError can be called with Para.
in tmp3.

Environment:
tmp3 = ParameterSize = Number of words of parameters (If call
RunError, not RunErrorO.
ExcCS = Segment number of the Except module.
ExcGP = Global pointer of the Except module.
tmp2 = Error number = Routine number of the exception.

Result:
tmp3 unchanged.
ExcCS unchanged.
ExcGP unchanged.
tmp2 unchanged.
tmp1 = ExcCS.

Calls:
ErrCall.

RunErrorO: tmp3 := 0;
LocR.RunError), tmp1 := ExcCS. Goto(ErrCall):
Abstract:
SASerr causes the ErrStrLong error.

Result:
tmp2 := ErrStrLong.

Calls:
RunErrorO.

SASerr: tmp2 := ErrStrLong, Goto(RunErrorO);
Routine StkOV.

Abstract:
StkOV signals a stack overflow error.

Environment:
tmp11 = Amount to back up the program counter.

Result:
Program counter backed up.
Stack overflow error signalled.

Calls:
PCBackup, RunError0.

Routine StkOVPop.

Abstract:
StkOVPop signals a stack overflow error after popping the expr stack.

Environment:
tmp11 = Amount to back up the program counter.

Result:
Stack popped.
Program counter backed up.
Stack overflow error signalled.

Calls:
PCBackup, RunError0, SetQState.

StkOVPop: Pop;
StkOV: Call(PCBackup);

!must be separate line, fixo
emp := 100;
Call(SetQState);
SL := SL + StackLimit;
!add a little extra to w
mp2 := ErrStackOverflow, Goto(RunError0);
Routine UOP.

Abstract:
UOP signals an ErrUndfQcd error.

Result:
  * tmp2 = ErrUndfQcd.

Calls:
  RUNErrorO.

-------------------------------------------------------------
3277 3273 32 14 6 1 1 0 1 3 14 51 0 3 3253
-------------------------------------------------------------

UOP:  tmp2 := ErrUndfQcd, Goto(RUNErrorO);

-------------------------------------------------------------
Title Interrupts: Microcode level and Pascal level.

Interrupts: Microcode level and Pascal level.
File: Perq.Routine.1 Perq Microcode Page 58

Routine UserSrv.

Abstract:
UserSrv serves Pascal level interrupts by calling the appropriate Pascal level interrupt service routine.

Environment:
UserIntr bits 0..14 non-zero. That is, UserIntr > 0.

Result:
One interrupt served.

Calls:
  ChkSeg, C11V, RefillJmp, VectSrv, Stk0v.

Design:
It is assumed that:
1) The segment with the interrupt handler is always resident.
2) The table with variable routine descriptors (IntTab) is quadword aligned.

-------------------------------------------------------------
3300 3277 30 2 0 6 0 1 0 1 3 11 77 0 3
3301 3300 30 3 7 7 6 0 0 4 3 11 76 0 3
3302 3301 64 0 6 1 1 0 1 3 11 75 0 3

UserSrv: tmp := UserIntr;
  tmp and 377:
  dst := 0, Goto(UserSrv1);
  I initialize count
If Neg Goto(UserSrv1);

dst := 10, if IntrPend Call(VectSrv);

userIntr, RightShift(10);

tmp := Shift;

UserSrv1: if IntrPend Call(VectSrv):

tmp, RightShift(1);

tmp := Shift, if Ddi Goto(UserSrv2); if this bit is set.

dst := dst + 1, Goto(UserSrv1); update bit position cou

! Bit found, index into IntTab by the bit position to get the r0 descriptor for the interrupt handler.

UserSrv2: dst, LeftShift(2):

Shift + IntPtr, Fetch4;

UserIntr := UserIntr or 100000;

tmp1 := 0, Load5(Stk0v);

tmp5 := Md1;

tmp7 := Md1 + SB;

tmp10 := Md1 + SB;

tmp := tmp5, Call(ChkSeg);

tmp6 := tmp5, Fetch6;

if Ddi Goto(Busted); if not resident

Hold, tmp := tmp2, LeftShift(3);

Hold, tmp1 := Shift + tmp6;

Hold, Md1 + tmp1, Fetch4;

Call(CWTV);

dst, LeftShift(4);

Shift or 17, Shift0R;

clear bit where interru

UserIntr := Shift xor UserIntr, Goto(RefillJmp); enter rout

Interru: Microcode level and Pascal level.
File: Perq.Route.1 Perq Microcode Page 59

---------------------

Routine VectSrv.

---------------------

Abstract:

VectSrv serves micro level interrupts by vectoring into the IO microcode. VectSrv should be called.

Environment:

IntrPend true.

Result:

interrupt served.

Calls:

IO microcode.
$Include Perq.Thesis  ! Code used in Thesis

   Luc A. Lepine (258009), August 14th, 1986
   This is implemented in partial fulfillment of my thesis.

This implements the structured programming instructions using micro-
Using previously undefined op-codes and registers, these routines an-
modified micro-code source implement the following instructions:

   If  <Lrdr1> <Lrdr2>
   While <Lrdr>
   Repeat

All these instructions require the evaluation of a boolean-expressio
The result of this expression must be on the Top Of Stack when the
instruction is executed. If the IOS is not 'true' or 'false' then a
type error, exactly the same as the one for conditional branches, is
raised.

For simplicity sake, the following restrictions and conventions ap-

(1) An LVRD instruction is used for the procedure addresses. This
   us to check to make sure the correct type of information follo-
   op-code is a branch to be taken. For the <If> instruction,
   instruction may be replaced by a <No-op> instruction. This all
   proper implementation of null 'else' and 'then' clauses.

(2) The instruction <Note> is used to delimit the range of <While>
   <Repeat> instructions. Because a boolean-expression may contai-
   a function which changes variables, a <No-op> is allowed as th
   <Lrdr> for a <While> instruction.

(3) The value of true is 1, of false is 0. This is defined in
   set true/false in Perq.Routine.1

(4) Until the information created by a Note instruction can be pla
in an ACB when a procedure is called, the <Repeat> instruction be nested. This does not matter for the <If> instruction and t
<While> instruction places the information in the Acb as the return address.

(5) An ErrInxCase exception will be raised if the instruction(s) f
a While or If-Then-Else instruction is/are not the one(s) expe

(6) An ErrStringx exception will be raised if the Note instruction not been executed prior to the execution of a While or Repeat
instruction in a procedure.

---

This routine performs the IF operation. If TOS is true the 1st
instruction is used to obtain the address of the procedure. If
is false the 2nd instruction is used to obtain the address of
procedure. A branch to CALLV is then made to perform the rout
unless the instruction was a NOOP.

3705 1274 0 0 6 0 0 0 0 3 10 113 0 3 - OpCode(AIF), Nop:

3705 3664 0 0 6 0 0 0 0 3 10 103 0 1 3716
3707 3665 0 0 6 0 0 0 0 3 10 106 15 3 3713
3710 3666 0 0 6 0 0 0 0 3 10 63 0 1 3736
3711 3667 0 0 6 0 0 0 0 3 10 74 0 1 3725
3712 3670 0 0 6 0 0 0 0 3 10 55 0 3 3744
3713 3671 0 0 6 0 0 0 0 3 10 74 0 1 3725
3714 3672 0 0 6 0 0 0 0 3 10 63 0 1 3736
3715 3673 0 0 6 0 0 0 0 3 10 55 0 3 3744

3716 3674 155 1 7 1 10 4 3 10 102 0 3 - ChkBool: Work1 := Tos and 1;
3717 3675 0 155 7 0 0 0 0 16 3 10 101 0 3 - Tos - Work1:
3720 3676 0 0 7 0 0 0 0 0 0 0 15 12
3721 3677 0 12 6 1 10 1 3 14 51 0 3 3253

This routine determines whether the value on the top of stack
be a boolean value. If not, a range exception is raised else t
value of TOS is gated so that the instruction after the Call c
if it is true (1) or false (0) ('eqv' or 'eqv').

This routine determines if the value in register Note1 has bee
If not, an ErrStringx exception is raised to indicate that the
is out of range (i.e. zero). If Note1 has been set, then a ret
made to the caller: Note1 is undefined if it contains zero.

ChkNote: Note1: If Neq Return; tmp2 := ErrStrIdx, GoTo (RunErrorO):
This routine looks at the next instruction and skips by it.
The instruction must be either an LVRD or a NOOP, else an exception is raised.

DoNull: Work1 := NextOp;
Work1 := NOOP;
Work1 := LVRD, If EqI Return; ! If <No-op>, return {1 byte op-
tmp2 := ErrInxCast, If Neq GoTo(RunErrorO); ! Else must be <LV
Work1 := NextOp;
Work1 := NextOp;
Work1 := NextOp;
Return;
LVRD> is a 5 byte op-code

DoLVRD: Work3 := NextOp;
Work3 := NOOP;
Work3 := LVRD, If EqI Return; ! No-op is simple
 tmp2 := ErrInxCast, If Neq GoTo (RunErrorO);
 CRO := i, Call(GoLVRD); ! Into special mode, load the VR
 CRO := 0, Return; ! Back to normal, return to Call
This routine causes a CALLV instruction to be performed.
It does not return to the caller.

DoCALLV: Work3 := NOOP;
If Neq GoTo (GoCallV);
NextInst(0);
This instruction notes the offset into the code base of the in
following it. This can be used to alter the default return address found in an ACB for a WHILE instruction, and as the ob of a Repeat loop.

N.B. Because of its implementation, Note1 can never be set to zero by this instruction. Therefore if Note1 is zero, it is undefined.

| 3747 1260 151 17 5 1 10 4 3 10 52 0 3 | OpCode(ANote), Note1 := Ustate and 17; |
| 3750 3725 16 6 6 0 0 0 16 2 0 340 0 16 | UPC := CB, LeftShift(1); |
| 3751 3726 151 151 0 0 1 0 14 0 0 377 0 2 | Note1 := Shift + Note1; |
| | NextInst(); |

This instruction causes a branch to the indicated procedure if expression in TOS is true. The return address is taken from th register set by the NOTE instruction. A special flag is set to indicate to the CALLV routine that the special register is to:

FAnote := Cal (DOnull); |
| NextInst(); |
| That completes this ins |

OpCode(While), Call (ChkNote); |
| Make sure Note1 has been |
| Verify boolean value on |
| If EqI Goto (FAnote); |
| If false, treat LVRD as |
| Allows Placer to resol |
| Else have to do LVRD |
| CRO := 1, Goto (GoCallV); |
| Set flag, perform CALLV |

This routine causes a branch to the last NOTE instruction exec at this lexical level if the TOS is false.

DoRepeat:

JmpOffSet := Ustate and 17; |
| determine current |
| UPC := CB, LeftShift(1); |
| JmpOffSet := Shift + JmpOffSet; |
| JmpOffSet := JmpOffSet - Note1; |
| JmpOffSet := not JmpOffSet; |
| Negate |
| JmpOffSet := JmpOffSet + 1, |
| GoTo (AdjustPC); |
| and Branch |

OpCode(ARepeat), Call (ChkNote); |
| Make sure Note1 has been
Title Initialization.

Routine Init.

Abstract:
Init is the entrypoint and initialization of the Perq Q-code interpreter microcode.

Loc(2400).

Initialize constant registers.

    tmp := InitBlock;
    Where := not 0;
    AllOnes := 177777;
    SignBit := 100000;
    SignXtrnd := not 777777;
    C1777 := 1777;
    C4000 := 4000;
    CRO := 0;
    Note1 := 0;

    Initialize no interrupts and interrupts off.
    UserIntr := 10000;
    Interrupts turned off
    Z80 state registers for IO microcode.
    Z80State := 100000;
    Z80State := 0;
    Z80Status := 0;
    Z80WantOutput := 0;
    DpyImp := 6000;
    zero := 0, StackReset;

    128 lines of Off inVIS
initial code segment
initial stack segment
no exception module yet
main program dynamic
main program return row
main program return seg
main program return cod
main program return add
set SB and SL registers
initial program static lin
initial top pointer
initial global pointer
new codebase
fetch dictionary entry
set up ACB etc.
enter main program
$NoList
The following pages contain the routines which were added or modified to insert the new instructions:

70, 73, 73, 88, 89, 100 through 104 and 104
Appendix C

INSTRUCTION TEST SAMPLE

To properly evaluate the performance of the new instructions, a series of tests were written to attempt to exercise all of the required features. These tests were to exercise both correctly and incorrectly coded instructions. The tests are split into two groups:

1. One test to exercise all the valid sequences of instructions.
2. Eight other tests to exercise invalid instruction coding of parameters for these new instructions.

Please note that some apparent procedures are not procedures but intrinsics. Procedures like statements which cause the compiler to generate code other than procedure calls. The intrinsics used in the test are the following:

1. `MakeVRD`: This causes the compiler to generate a LVRD instruction and place into its object code at that particular point. The only parameter is the name of a procedure or function.
2. `LoadExpr`: This causes the compiler to evaluate any expression and leave the result where it normally would be found before the result is assigned to a variable, or used as a parameter. LoadExpr in this
case is used to load the result of a boolean expression. Boolean expression results are always placed on
the top of the expression stack.

3. **InlineByte**: This allows the compiler to insert a byte into the object code at that particular point. The byte to be inserted must be a constant whose value lies between 0 and 255.

Also included is the corresponding object code as disassembled by the QDIS program on the PERQ. The mnemonic table used by this program was altered to include the definition of the four new op-codes. The op-codes should be simple to locate because they all begin with the character '<'.

Note that the section of the listing for the program 'TEST.PAS', which occurs first in this appendix, has had the section containing the PERQ op-codes removed. An edited version is contained in appendix A.

What follows is a list of the programs and the results they generated.
Program Tests (Input, Output):

Luc Lepine (258009)

This program written to partially fulfill requirements for CSI7989

This program tests the various capabilities of the Structured
Instructions added in the Perq micro-code.

All forms of tests are included and documented. The basic
structure of the program is straight line, except for the
procedures required for testing purposes.

The Ocode definitions that are part of the listing
have been removed. An abbreviated form of this part of
the listing has been placed in Appendix A.

Var
  I : Integer;

Procedure Test(Number:Integer);

This procedure displays the current test

Begin
  Writeln;
  Write('Test No.', Number:3, ' - ')
End;

Procedure SayTrue;

This procedure is called to write 'True!' on the terminal

Begin
  Write ('True !')
End;

- 109 -
Procedure SayFalse:
  This procedure is called to write 'False!' on the terminal
  Begin
  Write('False!');
  End;
Procedure NextI:
  This procedure increments I, writes it out, then exits
  Begin
  I := I + 1;
  Write(I:3);
  End;
These procedures are used to demonstrate that the address
  capability is not affected by new instruction calls
Procedure Show(A: Integer):
  Begin
  Write(A:3);
  End;
Procedure CheckA:
  Begin
  Show(I);
  End;
Procedure CheckB:
  Begin
  Show(I-4);
  End;
Procedure CheckC:
  Begin
  Show(I);
  I := I - 1;
  End;
Procedure LastI:
  This procedure is used to test nesting of the while loop.
  To make sure the information generated by a NOTE is kept
  through procedure calls and restored to the correct value
  when exiting a routine.
  Var
  J : Integer;
Begin
  J := I;
  InLineByte(ADETE);  
  LoadExpr(I < 10);
  InLineByte(AWHILE);
  MakeVrd(NextI);
  ( While I < 10 Do )
  NextI;
  I := J - 1;
End;

Procedure TestExit;

This procedure tests the EXIT instruction and verifies that
an exit at a location other than the end of a procedure
will still result in the NOTE information being restored

Begin
  Write('Testing EXIT routine. ');
  InLineByte(ADETE);
  Exit(TestExit);
End;

The main program!

Begin

{ Test the IF structure }

  Test(1):  LoadExpr(True);
  InLineByte(AIF);  MakeVrd(SayTrue);  MakeVrd(SayFalse);
  Test(2):  LoadExpr(False);
  InLineByte(AIF);  MakeVrd(SayTrue);  MakeVrd(SayFalse);
  Test(3):  LoadExpr(True);
  InLineByte(AIF);  InLineByte(NOOP);  Makevrd(SayFalse);
  Test(4):  LoadExpr(True);
  InLineByte(AIF);  MakeVrd(SayTrue);  InLineByte(NOOP);
  Test(5):  LoadExpr(True);
  InLineByte(AIF);  InLineByte(NOOP);  InLineByte(NOOP);
  Test(6):  LoadExpr(False);
  InLineByte(AIF);  InLineByte(NOOP);  InLineByte(NOOP);

{ Test the Repeat structure }

  Test(7):
  I := 0;
  InLineByte(ADETE);
  ( Start of repeat loop )
  NextI;
  LoadExpr(I >= 10);
  InLineByte(ARPEATE);
{ Test the While structure }

Test(8):

I := 0;
InLineByte(ANOTE);
LoadExpr (I < 7);
InLineByte(AWHILE);
MakeVrd(NextI);

{ Start of While loop }
While I < 7 Do

{ Call NextI }

Test(9):

Written;

I := 9;
InLineByte(ANOTE);
LoadExpr (I >= 4);
InLineByte(AWHILE);
MakeVrd(LastI);

{ Start of While Loop }
While I > 4 Do

{ Call LastI }

Test(10):

Written ('Test Successfull !

{ Test exit by other than simple termination }

Test(11); I := 27;

{ Test data }

LoadExpr(True);
InLineByte(AIF); MakeVrd(CheckA); MakeVrd(CheckB);

Test(12); I := 39;

{ Test data }

LoadExpr(False);
InLineByte(AIF); MakeVrd(CheckA); MakeVrd(CheckB);

Test(13); I := 47;

{ Test data }

InLineByte(ANOTE);
LoadExpr(I = 47);
InLineByte(AWHILE); MakeVrd(CheckC);

Written;

Written ('Finis!')
Disassembling the program: TESTS
which was generated by compiling: Sys\:Lepine\:TestSys\:Test.PAS

QCode Version Number: 3
Size of Global Data Block: 33
Length of Identifiers: 6
Number of Imported Segments: 2
Block containing Import list: 3

File: Sys\:Lepine\:TestSys\:test.SEG Routine: TESTEXIT (10)

Lex Lev PS RPS LTS Enter Exit
2 0 0 0 188 228

0: LSSN
1: LOAB 16
3: MMS2
4: LSA
8: MMS2
9: Testing EXIT routine.
30: LDIO
31: MMS
32: CALLXB 2 5
35: <Note>
36: EXIT 0 10
40: RETURN

File: Sys\:Lepine\:TestSys\:test.SEG Routine: LASTI (9)

Lex Lev PS RPS LTS Enter Exit
2 0 0 0 160 186

0: LDOB 32
2: STLO
3: <Note>
4: LDOB 32
6: LDIC10
7: <While>
9: LVRD 0 4 2
14: LSSN
15: LOAB 16
17: MMS2
18: CALLXB 1 8
21: LDLO
22: LDIC1
23: SBI 32
24: STOB
26: RETURN

File: Sys\:Lepine\:TestSys\:test.SEG Routine: CHECKC (8)
Lex Lev | PS | RPS | LTS | Enter | Exit
--- | --- | --- | --- | --- | ---
2 | 0 | 0 | 0 | 148 | 159

0: LDOB | 32
2: MMS
3: CALL | 5
5: LDOB | 32
7: LDC1
8: SBI
9: STOB | 32
11: RETURN

File: Sys:Lepine>TestSys>test.SEG  Routine: CHECKB  (7)

Lex Lev | PS | RPS | LTS | Enter | Exit
--- | --- | --- | --- | --- | ---
2 | 0 | 0 | 0 | 140 | 147

0: LDOB | 32
2: LDC4
3: SBI
4: MMS
5: CALL | 5
7: RETURN

File: Sys:Lepine>TestSys>test.SEG  Routine: CHECKA  (6)

Lex Lev | PS | RPS | LTS | Enter | Exit
--- | --- | --- | --- | --- | ---
2 | 0 | 0 | 0 | 134 | 139

0: LDOB | 32
2: MMS
3: CALL | 5
5: RETURN

File: Sys:Lepine>TestSys>test.SEG  Routine: SHOW  (5)

Lex Lev | PS | RPS | LTS | Enter | Exit
--- | --- | --- | --- | --- | ---
2 | 1 | 1 | 0 | 122 | 133

0: LSSN | 16
1: LOAB
3: MMS2
4: LDLO
5: MMS
6: LDC3
7: MMS
8: CALLXB | 2
11: RETURN

File: Sys:Lepine>TestSys>test.SEG  Routine: NEXTI  (4)

Lex Lev | PS | RPS | LTS | Enter | Exit
--- | --- | --- | --- | --- | ---
2 | 0 | 0 | 0 | 102 | 120
<table>
<thead>
<tr>
<th>Lex</th>
<th>Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTS</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>230</td>
<td>549</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

File: Sys:Lepine>TestSys>test.SEG  Routine: TESTS  (0)
119: LDGB  32
121: LDGIO
122: GEQI
123: <Repeat
124: LDGB
125: MMS
126: CALL  1
128: LDCO
129: STOB  32
131: <Note>
132: LDGB  32
134: LDC7
135: LESI
136: <While>
137: LVRD  0  4  2
142: LDC9
143: MMS
144: CALL  1
146: LSSN
147: LQAB  16
149: MMS2
150: CALLXB  1  8
153: LDC9
154: STOB  32
156: <Note>
157: LDGB  32
159: LDC4
160: GEQI
161: <While>
162: LVRD  0  9  2
167: LDC10
168: MMS
169: CALL  1
171: LDC2
172: STOB  32
174: <Note>
175: CALL  10
177: LDGB  32
179: LDC1
180: ADI
181: STOB  32
183: LDGB  32
185: LDC4
186: EQU
187: <Repeat
188: LSSN
189: LQAB  16
191: MMS2
192: LSA  'Test Successfull !'
213: MMS2
214: LDCO
215: MMS
216: CALLXB  2  5
219: LSSN
Test No. 5 -
Test No. 6 -
Test No. 7 - 1 2 3 4 5 6 7 8 9 10
Test No. 8 - 1 2 3 4 5 6 7
Test No. 9 -
10
9 10
8 9 10
7 8 9 10
6 7 8 9 10
5 6 7 8 9 10

Test No. 10 - Testing EXIT routine. Testing EXIT routine. Test Successful!

Test No. 11 - 27
Test No. 12 - 35
Test No. 13 - 47
For brevity, only the program source is included in the rest of this appendix.

Program Test2:

(This program will test what happens when a NOTE instructions is not executed before a WHILE instruction)

Const
{$Include Perq.Qcodes.DFS}

Begin
WriteLn('Testing While clause.');
LoadExpr(False);
InLineByte(AWHILE)
End.

Disassembling the program: TEST2
which was generated by compiling: Sys:Lepine>TestSys>Test2.PAS

QCode Version Number: 3
Size of Global Data Block: 32
Length of Identifiers: 8
Number of Imported Segments: 2
Block containing import list: 2

File: Sys:Lepine>TestSys>Test2.SEG Routine: TEST2 (0)

<table>
<thead>
<tr>
<th>Lex Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTS</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSN</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>83</td>
</tr>
<tr>
<td>1</td>
<td>LOAB</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LDCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LDCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LDC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>LDCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>CALLXB</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>LSSN</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>LOAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>LSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>LDCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 121 -
Testing While clause.

String index out of range
Aborted at 79 in routine TEST2 (O) in TEST2.
Called from 169 in routine 0 in LOADER.
Called from 226 in routine 1 in SYSTEM.
Called from 568 in routine 0 in SYSTEM.
Program Test3;

{ This program will test what happens when a NOTE instructions is
  not executed before a Repeat Instruction }

Const
  {$Include Perq.Qcodes.DFS}

Begin
  WriteIn('Testing Repeat clause.');
  LoadExpr(False);
  InlineByte(AREPEAT)
End.

Disassembling the program: TEST3
which was generated by compiling: Sys:Lepine>TestSys>test3.PAS

QCode Version Number: 3
Size of Global Data Block: 32
Length of Identifiers: 8
Number of Imported Segments: 2
Block containing import list: 2

File: Sys:Lepine>TestSys>test3.SEG Routine: TEST3 (O)

<table>
<thead>
<tr>
<th>Lex Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTS</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 123 -
33: MMS
34: CALLXB  1  1
37: LSSN
38: LOAB  16
40: MMS2
41: LSA  'Testing Repeat clause.'
65: MMS2
66: LDCO
67: MMS
68: CALLXB  2  5
71: LSSN
72: LOAB  16
74: MMS2
75: CALLXB  1  8
78: LDCO
79: <Repeat
80: RETURN

Testing Repeat clause.

String index out of range
Aborted at  80  in routine  TEST3  (0)  in TEST3.
Called from  169  in routine  0  in LOADER.
Called from  226  in routine  1  in SYSTEM.
Called from  588  in routine  0  in SYSTEM.
Program Test4;

{ This test verifies that a Note instruction must be executed in the
same procedure to be valid, the value will not migrate forwards through
procedure calls }

Const
{$include Perq.Ocodes.DFS}

Procedure Test4A;
Begin
   Write('Procedure Test4A entered.';
   Load Expr(True);
   InLineByte(AWHILE); MakeVRD(Test4A);
End;

Begin
   Write('Testing nested control structures !');
   InLineByte(ANDOE);
   Test4A;
End.

Disassembling the program: TEST4
which was generated by compiling: Sys:Lepine>TestSys@test4.PAS

QCode Version Number: 3
Size of Global Data Block: 32
Length of Identifiers: 8
Number of Imported Segments: 2
Block containing Import list: 2

File: Sys:Lepine>TestSys>Test4.SEG Routine: TEST4A (1)
Lex Lev PS PPS LTS Enter Exit
2 0 0 0 4 56

0: LSSN
1: LOAB 16
3: MMS2
4: LSA 'Procedure Test4A entered.'
32: MMS2
33: LDCO
34: MMS
35: CALLXB 1 5
38: LSSN
39: LOAB 16
41: MMS2
42: CALLXB 2 8
45: LDC1
46: <WHILE>
47: LVRD
52: RETURN

- 125 -
Testing nested control structures!
Procedure Test4A entered.

String index out of range
Aborted at 47 in routine TEST4A (1) in TEST4.
Called from 94 in routine TEST4 (0) in TEST4.
Program Test5:

(This test verifies that the object of a While instruction must be
either a NOOP, or a LVRD instruction.
The instruction following the While instruction will be a RETURN)

Const
{$Include Perq.Qcodes.DFS$}

Begin
  WriteLn('Testing validation of instruction after While instruction ('):
  InLineByte(ANOTE);
  InLineByte(AWHILE);
End.

Disassembling the program: TEST5
which was generated by compiling: Sys:Lepine>TestSys>test5.PAS.

QCode Version Number: 3
Size of Global Data Block: 32
Length of Identifiers: 8
Number of Imported Segments: 2
Block containing Import list: 2

File: Sys:Lepine>TestSys>test5.SEG    Routine: TEST5    (0)

<table>
<thead>
<tr>
<th>Lex Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTR</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>121</td>
</tr>
</tbody>
</table>

0: LSSN
1: LOAD
3: MMS
4: LSA
7: MMS2
8: LDCO
9: MMS
10: LDC8
11: MMS
12: LDC1
13: MMS
14: LDCO
15: MMS
16: CALLXB
19: LSSN
20: LOAD
22: MMS2
23: LSA
25: MMS2
26: LDCO
27: MMS
28: LDC8
29: MMS
Testing validation of instruction after While instruction!

Expression out of range
Aborted at 118 in routine TEST5 (O) in TEST5.
Called from 169 in routine 0 in LOADER.
Called from 226 in routine 1 in SYSTEM.
Called from 568 in routine 0 in SYSTEM.
Program Test6;

{ This test verifies that the first instruction after an IF instruction must be a NOOP or a LVRO. }

Const
($Include Preq.Qcodes.DFS)

Begin
  Writeln('Test 6:');
  LoadExpr(True);
  InLineByte(AIF)
End.

Disassembling the program: TEST6
which was generated by compiling: Sys: Lepine>TestSys>test6.PAS

<table>
<thead>
<tr>
<th>QCCode Version Number</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Global Data Block</td>
<td>32</td>
</tr>
<tr>
<td>Length of Identifiers</td>
<td>8</td>
</tr>
<tr>
<td>Number of Imported Segments</td>
<td>2</td>
</tr>
<tr>
<td>Block containing Import list</td>
<td>2</td>
</tr>
</tbody>
</table>

File: Sys: Lepine>TestSys>test6.SEG  Routine: TEST6  (0)

<table>
<thead>
<tr>
<th>Lex Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTS</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSN</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>LDB</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>MMS2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>LSA</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>MMS2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>LDCO</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>LDCB</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>LDC1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>LDCO</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>CALLXB</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>LSN</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>LDB</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>MMS2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>LSA</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>MMS2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>LDCO</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>LDCB</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>LDC1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>MMS</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>LDC1</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Expression out of range
Aborted at 67 in routine TEST6 (0) in TEST6.
Called from 169 in routine 0 in LOADER.
Called from 226 in routine 1 in SYSTEM.
Called from 568 in routine 0 in SYSTEM.
Program Test7;

{ This test verifies that the second instruction after an IF instruction must be a NOOP or a LVRD. }

Const
{$Include Pegr.Qcodes.DFS}

Begin
WriteIn('Test 7 !');
LoadExpr(True);
InLineByte(AIF);
MakeVRD(Test7);
End.

Disassembling the program: TEST7
which was generated by compiling: Sys:\Lepine\TestSys\test7.PAS

QCod Version Number:  3 —
Size of Global Data Block:  32
Length of Identifiers:  8
Number of Imported Segments:  2
Block containing import list:  2

File: Sys:\Lepine\TestSys\test7.SEG    Routine: TEST7    (O)

Lex Lev PS RPS LTS Enter Exit
1 0 0 0 4 75

0: LSSN 0
1: LOAB
3: MMS2
4: LSA
7: MMS2
8: LDCO
9: MMS
10: LDC8
11: MMS
12: LDC1
13: MMS
14: LDCO
15: MMS
16: CALLXB 1 1
19: LSSN
20: LOAB 16
22: MMS2
23: LSA
25: MMS2
26: LDCO
27: MMS
28: LDC8
29: MMS
30: LDC1
Program TestB:

{ This test verifies that the second instruction after an IF instruction
  must be a NOOP or a LVRO.
}

Const
($Include Perq.Qcodes.DFS)

Begin
WriteLn('Test B !');
LoadExpr(False);
InLineByte(AIF);
MakeVRD(TestB);
InLineByte(NOOP)
End.

Disassembling the program: TESTB
which was generated by compiling: Sys:Leodine>TestSys>testB.PAS

<table>
<thead>
<tr>
<th>QCode Version Number</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Global Data Block</td>
<td>32</td>
</tr>
<tr>
<td>Length of Identifiers</td>
<td>8</td>
</tr>
<tr>
<td>Number of Imported Segments</td>
<td>2</td>
</tr>
<tr>
<td>Block containing import list</td>
<td>2</td>
</tr>
</tbody>
</table>

File: Sys:Leodine>TestSys>testB.SEG Routine: TESTB (O)

<table>
<thead>
<tr>
<th>Lex Lev</th>
<th>PS</th>
<th>RPS</th>
<th>LTS</th>
<th>Enter</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>LSSN</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>LDAB</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>LSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>LDG0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>LDCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>LDC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>LDC0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:</td>
<td>CALLXB</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:</td>
<td>LSSN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20:</td>
<td>LOAB</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>22:</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:</td>
<td>LSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25:</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26:</td>
<td>LDC0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28:</td>
<td>LDCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29:</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:</td>
<td>LDC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Code</td>
<td>Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>MWS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>LDCM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>MWS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>CALLXB</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>LSSN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>LOAD</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>LSA</td>
<td>&quot;Test 8!&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>LDCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>MMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>CALLXB</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>LSSN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>LOAD</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>MMS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>CALLXB</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>LDCO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>&lt;IF&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>LVRD</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>NOP</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>RETURN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test 8!
Program Test9;

( This test verifies that the value on the top of the expression stack
  must be boolean, i.e. 0 or 1 )

Const
  ($include Req.Qcodes.DFS)

Begin
  WriteInl('Test 9 !');
  LoadExpr(7);
  InLineByte(AIF);
  MakeVRD(Test9);
  InLineByte(NOOP)
End.

Disassembling the program: TEST9
which was generated by compiling: Sys: Lepine>TestSys>test9.PAS

QCode Version Number: 3
Size of Global Data Block: 32
Length of Identifiers: 8
Number of Imported Segments: 2
Block containing import list: 2

File: Sys: Lepine>TestSys>test9.SEG    Routine: TEST9    (O)

Lex Lev PS  RPS  LTS Enter Exit
  1  0  0  0  4  76

  0: LSSN
  1: LOAB  0
  3: MMS2
  4: LSA
  7: MMS2
  8: LDCC
  9: MMS
 10: LDCC
 11: MMS
 12: LDCC
 13: MMS
 14: LDCC
 15: MMS
 16: CALLXB  1  1
 19: LSSN
 20: LOAB 16
 22: MMS2
 23: LSA
 25: MMS2
 26: LDCC
 27: MMS
 28: LDCC
 29: MMS
 30: LDCC
Test 9 !
Expression out of range.
Aborted at 66 in routine TEST9 (0) in TEST9.
Called from 169 in routine 0 in LOADER.
Called from 226 in routine 1 in SYSTEM.
Called from 568 in routine 0 in SYSTEM.
Appendix D

AMDAHL INSTRUCTION EMULATING ROUTINE

This is the listing of the routine which performs the emulation of the three new instructions on the Amdahl V7A. The routine is written in standard IBM assembler, and was compiled using the H level assembler from IBM. This version of the assembler contains modifications obtained from the Stanford Linear Accelerating Centre (SLAC), but this has no effect on the object code produced by the compiler, or the performance of the routine.

The implementation of this routine requires no modifications to the CMS operating system. It is implemented by loading the routine in this appendix into permanent system storage, using a CMS command. The program is then called, specifying the 'START' parameter. This causes the program check interrupt vector to be altered by the program so that it obtains control when a program check occurs. Whenever any of the new instructions are executed, a program check specifying an invalid machine instruction occurs. The program obtains control, verifies it is a new instruction, performs it, then passes control back to the user program. If the interrupt is not caused by one of the new instructions, the program will pass control to CMS, so that it can handle the error.
The program occupies 630 bytes of storage, which is comprised of 81 instructions for the emulating program and a few constants and more instructions for the CMS command interface.

N.B. The object code is in hexadecimal.
EXTERNAL SYMBOL DICTIONARY

DMSSP ON MNT390
MEMBER(S) USED - NUCON

CMSLIC ON MNT390
MEMBER(S) USED - DMSLN

DMSLNC DMSLND DMSLNP DMSLNU DMSLNY DMSLNZ LINEDIT

DSECT STMT LENGTH DSECT STMT LENGTH DSECT STMT LENGTH

NEWINST

SOURCE STATEMENT

2
3 * THIS ROUTINE IMPLEMENTS USING 'MACRO-CODE' THE 'IF' AND 'WHILE'
4 * CONTROL STRUCTURES. THE IMPLEMENTATION Requires THAT THIS ROUTINE
5 * BE FIXED ANYWHERE IN THE CMS Nucleus AREA, AND THEN ENVO'ED TO
6 * TAKE CONTROL OF THE VIRTUAL MACHINE WHEN A PROGRAM CHECK OCCURS.
7 *
8 * THE FOLLOWING INSTRUCTIONS ARE IMPLEMENTED:
9 *
10 * INSTRUCTION
11 * OP-CODE
12 * INSTRUCTION LENGTH
13 * PARAMETERS: CC MASK
14 * LINK REG.
15 * SCON 1
16 * SCON 2
17 * FUNCTION
18 * THE BRANCH CONDITION TO TRUE OR FALSE. IF
19 * DETERMINE THE LOCATION OF THE ROUTINE TO
20 * BE EXECUTED. IF IT IS 'FALSE', THE SECOND
21 * ADDRESS IS USED. THE PROGRAM COUNTER IS
22 * UPDATED TO REFLECT THIS FACT. THE ADDRESS
23 * OF THE NEXT INSTRUCTION IS PLACED INTO
24 * THE LINK REGISTER.
25 *
26 * ERROR(S)

28 * INSTRUCTION
29 * OP-CODE
30 * INSTRUCTION LENGTH
31 * PARAMETERS: CC MASK
32 * LINK REG.
33 * SCON

SLAC V2.9  ASM H V 02  10.40  10/25/83
34 * FUNCTION
35 * BRANCH CONDITION IS TRUE. IF IT IS, THE
36 * ADDRESS IS USED TO UPDATE THE PROGRAM
37 * COUNTER TO EFFECT A BRANCH; THE LINK
38 * REGISTER TO ALTERED SO THAT THE ADDRESS
39 * OF THIS INSTRUCTION IS PLACED INTO IT.
40 * ERRORS
41 * IF-THEN
42 * OP-CODE
43 * X'41'
44 * INSTRUCTION LENGTH
45 * 4 BYTES
46 * PARAMETERS: CC MASK
47 * HIGH ORDER NIBBLE, SECOND BYTE
48 * LOW ORDER NIBBLE, SECOND BYTE
49 * RELOCATABLE ADDRESS CONSTANT
50 * THIS FUNCTION THE SAME WAY AS THE
51 * IF-THEN-ELSE INSTRUCTION. EXCEPT THAT
52 * THERE IS NO SECOND ADDRESS. AND NO ACTION
53 * IS TAKEN IF THE BRANCH CONDITION IS FALSE.
54 * THE NEXT SEQUENTIAL INSTRUCTION WILL BE
55 * PROCESSED.
56 * NONE.
57 * ERROR(S)
58 * (SLAC V2.9) ASM H V 02 10.40 10/25/83
59 * THE IMPLEMENTATION OF THE NEW INSTRUCTIONS AS FOLLOWS:
60 * (UNDEFINED OP-CODE)
61 * (1) THE NEW INSTRUCTION CAUSES A PROGRAM CHECK, CODE 1
62 * (2) THIS ROUTINE OBTAINS CONTROL AND DETERMINES IF THIS IS AN
63 * INSTRUCTION TO BE SIMULATED. IF NOT, THE CMS VERSION OF THE
64 * PGCMPSW FIELD IS LOADED, AND CONTROL PASSES TO CMS.
65 * (3) THE CORRECT ROUTINE IS CALLED BY REPLACING THE
66 * ADDRESS IN THE PGCMPSW, THEN RELOADING THE REGISTERS, THEN
67 * THE PGCMPSW TO TRANSFER CONTROL TO THE PROPER ROUTINE.
68 * NOTES: ANY LINKAGE IS PERFORMED USING THE BRANCH AND SAVE CONVENTION.
69 * ONLY THE ADDRESS IS PLACED INTO THE REGISTER.
70 * THIS ROUTINE SHOULD BE LOADED, USING THE PROGRAM 'RSLSIB',
71 * INTO CMS NUCLEUS STORAGE THEN ACTIVATED BY CALLING THE
72 * PROGRAM 'NEWINST' WITH THE 'START' OPTION. TO DEACTIVATE
73 * THE NEW INSTRUCTIONS, CALL 'NEWINST' WITH THE 'STOP' OPTION.
74 * THE 'BRANCH ON CONDITION' INSTRUCTIONS.
75 * ONLY THE BC MODE PSW IS ASSUMED HERE. CMS DOES NOT UNDERSTAND
76 * OR USE EC MODE PSWS, SO THIS IS A SAFE ASSUMPTION.
77 * NEWINST THE CMS PROGRAM
78 * (SLAC V2.9) ASM H V 02 10.40 10/25/83
79 * THIS ROUTINE MUST BE LOADED PERMANENTLY INTO STORAGE, OR THE CMS
80 * WILL FAIL ON THE FIRST PROGRAM CHECK.
81 * NEWINST START.
85  PRINT NOGEN  NO MACRO EXPANSIONS  USEFUL EQUATS

00000  B7  RO  EQU  0
00001  88  R1  EQU  1
00002  89  PARMREG  EQU  2
00003  90  R3  EQU  3
00004  91  R4  EQU  4
00005  92  R5  EQU  5
00006  93  R6  EQU  6
00007  94  R7  EQU  7
00008  95  R8  EQU  8
00009  96  SCON2  EQU  9
0000A  97  SCON1  EQU  10
0000B  98  OPCODE  EQU  11
0000C  99  BASEREG  EQU  12
0000D  100  R13  EQU  13
0000E  101  R14  EQU  14
0000F  102  R15  EQU  15
00010  104  IFTE  EQU  'X=F4'  IF-THEN-ELSE
00011  105  WHILE  EQU  'X=AO'  WHILE-DO
00012  106  IFT  EQU  'X=A1'  IF-THEN

00010  18CF

00012  4120  10CB

00016  95FF  2000  0000

0001A  4780  C072

00018  95FF  2008  0008

0001C  4780  C02C

00020  0507  C260  2000  00260  0000

00024  4780  C06C  000C

000092  47FO  C064  00064

0000BE  47FO  C064  00064  00064

NEWINST  Initialise the environment for the new instructions

ACTIVE USINGS - (NUCON,RO) (OPARM,R2) (OPCODE,R11) (NEWINST,R12)

LOC  OBJECT CODE  ADDR2  STMT  SOURCE STATEMENT

0000C2  DS  0068  C250  0068  00250  176  CLC  PGMPNSW,NEWPSW

0000CC  4780  COEC  00E0  00E0C  177  BE  SETRC

0000CD  0207  C248  00B8  0248  0068  178  MVC  OLDPSW,PGMPNSW

0000DD  0207  C250  0068  0250  179  MVC  PGMPNSW,NEWPSW

NEWINST  Terminate the environment for the new instructions

ACTIVE USINGS - (NUCON,RO) (OPARM,R2) (OPCODE,R11) (NEWINST,R12)

LOC  OBJECT CODE  ADDR2  STMT  SOURCE STATEMENT

(SLAC V2.9)  ASM H V 02  10.40  10/25/83

- 143 -
NEWINST INSTRUCTION EMULATION ROUTINE

ACTIVE USING - (NUCON, RO) (OPARM, R2) (OPCODE, R11)

LOC OBJECT CODE ADDR1 ADDR2

IDENTIFICATION STATEMENT

NEWO1450 0000DC 0000D 0068 C250 00068 00250
NEWO1460 0000E2 4770 COEC 000EC
NEWO1470 0000E6 D207 008B C248 00068 00248
NEWO1480
NEWO1500
NEWO1510
NEWO1530 PAGE 7

182 DO STOP DS OH ARE WE RUNNING ENABLED ?-
NEWO1450
183 CLC PGMNPSW,NEWPSW NO, IGNORE THE REQUEST
NEWO1460
184 BNE SETRCO YES, RE-INSERT THE CMS PGMNPSW
NEWO1470
185 MVC PGMNPSW,OLDPSW
NEWO1480
187 SETRCO LINEDIT TEXT = 'REQUEST COMPLETED DOT=NO'
NEWO1500
197 RETURN (14,12),T,RC=0 RETURN TO SYSTEM
NEWO1510
203 DROP BASEREG DROP ADDRESSABILITY
NEWO1530

205 THIS ROUTINE DETERMINES IF THE NEW INSTRUCTIONS SHOULD BE EMULATED.
NEWO1550
206
NEWO1560
207 THE LOGIC IS AS FOLLOWS:
NEWO1570
208
NEWO1580
209 THIS ROUTINE GAINS CONTROL ON A PROGRAM CHECK.
NEWO1590
210 IF NOT AN UNDEFINED INSTRUCTION CHECK, CONTROL PASSES TO CMS
NEWO1600
211 IF NOT ANY OF THE NEW INSTRUCTIONS, CONTROL PASSES TO CMS
NEWO1610
212 THE INSTRUCTION IS EMULATED, I.E. REGISTERS AND PSW ALTERED
NEWO1620
213 AS REQUIRED SO THAT NORMAL INSTRUCTION PROCESSING WILL
NEWO1630
214 CONTINUE AS IF THE HARDWARE HAD PERFORMED THE INSTRUCTION.
NEWO1640
215 CONTROL IS RETURNED TO THE INSTRUCTION FOLLOWING THE ONE THAT
NEWO1650
216 CAUSED THE ORIGINAL PROGRAM CHECK. CONTROL IS NOT PASSED TO
NEWO1660
217 CMS.
NEWO1670

219 PERFORM DS OH ENTRY POINT FOR PROGRAM CHECK
NEWO1690
220 STM RO,R15,GPRLG SAVE GENERAL REGISTERS
NEWO1700
221 BALR BASEREG,0 ESTABLISH ADDRESSABILITY
NEWO1710
222 USING *,BASEREG
NEWO1720
223 CLC +(1),PGMPSW"+3" IS THIS DUE TO AN ILLEGAL OP-CODE ?
NEWO1740
224 BNE PGMPSW C4 ASC INSTRUCTION LENGTH
NEWO1750
225 L R14,RO,OPCODE,PGMPSW+4 GET THE ADDRESS OF THE NEXT OP-CODE
NEWO1760
226 SR R14,R14 DETERMINE THE INSTRUCTION LENGTH
NEWO1770
227 IC R14,PGMPSW+4 &
NEWO1780
228 IC R14,LENGTH(R14)
NEWO1790
229 IC R14,LEN
NEWO1800
230 IC R14,LENGTH(R14)
NEWO1810
231 CLI *OPCODE,IFITE AND CALCULATE ADDRESS OF THE OP-CODE
NEWO1820
232 CLI *OPCODE,IFITE DETERMINE IF IT IS A VALID INSTRUCTION
NEWO1830
233 CLI *OPCODE,IFITE
NEWO1840
234 CLI *OPCODE,WHILE
NEWO1850
235 CLI *OPCODE,WHILE
NEWO1860
236 CLI *OPCODE,WHILE
NEWO1870
237 CLI *OPCODE,WHILE
NEWO1880
238 CLI *OPCODE,WHILE
NEWO1890
239 TOCMS MVC MCKPSW,OLDPSW PASS CONTROL TO CMS
NEWO1900
240 LM RO,R15,GPRLOG RESTORE GENERAL PURPOSE REGISTERS
NEWO1910
241 LPSW MCKPSW AND LOAD CMS VERSION OF PGMNPSW
NEWO1910

NEWO1930
NEWO1940
NEWO1950
NEWO1960

- 144 -
NEWINST IF-THEN-ELSE INSTRUCTION

ACTIVE USING: - (NUMON,RO) (OPARM,R2) (OPCODE,R11) (NEWINST+X'11C',R12)

LOC OBJECT CODE ADDR ADDR2 SIMT SOURCE STATEMENT

00001A 43EO BO01 00001 248 IC R14,OPCODE+1 OBTAIN BRANCH MASK
00001E 54EO C14C 00268 249 N R14,-(A\"X\"FO\") AND ONLY BRANCH MASK
0000172 0F08 002C 250 ICM R15,8,PGMOPS+4
0000176 0F08 002C 251 SPM R15 SET CC+ILC+PROGRAM MASK
000017B 44EO COA2 0018E 252 EX R14,TESTIFTE MERGE WITH IF-THEN-ELSE, BRANCH IFF TRUE
000017C 47FO C11E 0023A 253 B NOOP RETURN CONTROL TO USER PROGRAM

NEWINST WHILE INSTRUCTION

ACTIVE USING: - (NUMON,RO) (OPARM,R2) (OPCODE,R11) (NEWINST+X'11C',R12)

LOC OBJECT CODE ADDR ADDR2 SIMT SOURCE STATEMENT

00016A 43EO BO01 00001 248 IC R14,OPCODE+1 OBTAIN BRANCH MASK
00016E 54EO C14C 00268 249 N R14,-(A\"X\"FO\") AND ONLY BRANCH MASK
000172 0F08 002C 250 ICM R15,8,PGMOPS+4
000176 0F08 002C 251 SPM R15 SET CC+ILC+PROGRAM MASK
00017B 44EO COA2 0018E 252 EX R14,TESTIFTE MERGE WITH IF-THEN-ELSE, BRANCH IFF TRUE
00017C 47FO C11E 0023A 253 B NOOP RETURN CONTROL TO USER PROGRAM

NEWINST LOCAL SUBROUTINE(S)

ACTIVE USING: - (NUMON,RO) (OPARM,R2) (OPCODE,R11) (NEWINST+X'11C',R12)

LOC OBJECT CODE ADDR ADDR2 SIMT SOURCE STATEMENT

00020A 43EO BO01 00001 248 IC R14,OPCODE+1 OBTAIN BRANCH MASK
00020E 54EO C14C 00268 249 N R14,-(A\"X\"FO\") AND ONLY BRANCH MASK
0002172 0F08 002C 250 ICM R15,8,PGMOPS+4
0002176 0F08 002C 251 SPM R15 SET CC+ILC+PROGRAM MASK
000217B 44EO COA2 0018E 252 EX R14,TESTIFTE MERGE WITH IF-THEN-ELSE, BRANCH IFF TRUE
000217C 47FO C11E 0023A 253 B NOOP RETURN CONTROL TO USER PROGRAM
NEWINST LOCAL SUBROUTINE(S)

NEWINST+X '11C', R12

SET THE LINK REGISTER
R14 - RETURN ADDRESS
R15 - WORK REGISTER

NEWINST+X '11C', R12

IC R15, OPCODE+1 GET THE LINK REGISTER NUMBER

NEWINST+X '11C', R12

Newinst Constants and Dsects

NEWINST+X '11C', R12

PASS CONTROL TO USER PROGRAM

NEWINST+X '11C', R12

Restore Registers

NEWINST+X '11C', R12

Return Control to User Program

Page 12

Page 13
NEWINST

NO STATEMENTS FLAGGED IN THIS ASSEMBLY
OVERRIDING PARAMETERS- NONE
OPTIONS FOR THIS ASSEMBLY
OCPXA, NOLIBMAC, DXREF, LOOSE, NOFOLD, INTEGER, MSRC, UMAP, XL, NOOPCNTS, DISK, NOREL2, TERM, XREF(SHORT), NORL, ESD, ALIGN, BATCH, NOTEST, NORENT, LIST, OBJECT, NODECK, LINECOUNT(59), FLAG(0), SYSPARM(), PEXIT()

NO OVERRIDING DD NAMES
474K ALLOCATED TO BUFFER POOL. 137K WOULD BE REQUIRED FOR THIS TO BE AN INCORE ASSEMBLY.

200 CARDS FROM SYSIN
2277 CARDS FROM ?SLIB
0 SYSUT1 READS
425 LINES OUTPUT
15 CARDS OUTPUT
30 SYSUT1 WRITES
Appendix E
TEST PROGRAMS FOR AMDAHL V7A

There are two tests in this appendix which test out each of the three instructions implemented on the AMDAHL. These tests perform a bubble sort on an incore table of 20 elements, displaying the results on the user's console. The difference between the two tests is that the If-Then (IFT) and If-Then-Else (IFTE) instructions are interchanged. The IFT instruction executes the Else clause zero times while the IFTE instruction executes the Else clause more than zero times.
TEST

1. SYMBOL
   TYPE 1D ADDR LENGTH LD ID FLAGS
   TEST  SD 00001 000000 000110 00

EXTERNAL SYMBOL DICTIONARY

MACRO AND COPY CODE EXTERNAL SOURCE CROSS REFERENCE

MACRO DEFINITION FOUND WITHIN INPUT STREAM
MEMBER(S) USED - IFT IFTE WHILE

CMSLIB ON MNT390
MEMBER(S) USED - DMSLN DMSLNC DMSLND DMSLHP DMSLNU DMSLNY DMSLNZ LINEDIT REGEQU

OMACRO ON MNT390
MEMBER(S) USED - RETURN SAVE

TEST

ACTIVE USING - NONE
LOC. OBJECT CODE ADDR1 ADDR2 STMT SOURCE STATEMENT

2. THIS PROGRAM TESTS OUT THE 'IFT' AND 'WHILE' INSTRUCTIONS
3. THE PROGRAM IS A SIMPLE BUBBLE SORT ON 20 RANDOM ELEMENTS.

6
7 &LABEL IFTE &MASK,&LINK,&SCON1,&SCON2
8 &LABEL DC 'X'F4',AL.4(&MASK,&LINK),S(&SCON1,&SCON2)
9 MEND

11
12 &LABEL WHILE &MASK,&LINK,&SCON
13 &LABEL DC X'AO',AL.4(&MASK,&LINK),S(&SCON)
14 MEND

16
17 &LABEL IFT &MASK,&LINK,&SCON
18 &LABEL DC X'A1',AL.4(&MASK,&LINK),S(&SCON)
19 MEND

TEST

BUBBLE SORT PROGRAM
ACTIVE USING - NONE
LOC. OBJECT CODE ADDR1 ADDR2 STMT SOURCE STATEMENT

000000 34*R0 EQU 0
000001 35*R1 EQU 1
000002 36*R2 EQU 2
000003 37*R3 EQU 3
000004 38*R4 EQU 4
000005 39*R5 EQU 5

01-RGEO
01-RGEO
01-RGEO
01-RGEO
01-RGEO
00006  40+R6  EQU  6
00007  41+R7  EQU  7
00008  42+R8  EQU  8
00009  43+R9  EQU  9
0000A  44+R10  EQU  10
0000B  45+R11  EQU  11
0000C  46+R12  EQU  12
0000D  47+R13  EQU  13
0000E  48+R14  EQU  14
0000F  49+R15  EQU  15

51** FLOATING POINT REGISTERS
00000  53+F0  EQU  0
00002  54+F2  EQU  2
00004  55+F4  EQU  4
00006  56+F6  EQU  6

58** EXTENDED CONTROL REGISTERS
00000  60+C0  EQU  0
00001  61+C1  EQU  1
00002  62+C2  EQU  2
00003  63+C3  EQU  3
00004  64+C4  EQU  4
00005  65+C5  EQU  5
00006  66+C6  EQU  6
00007  67+C7  EQU  7
00008  68+C8  EQU  8
00009  69+C9  EQU  9
0000A  70+C10  EQU  10
0000B  71+C11  EQU  11
0000C  72+C12  EQU  12
0000D  73+C13  EQU  13
0000E  74+C14  EQU  14
0000F  75+C15  EQU  15

TEST  BUBBLE SORT PROGRAM
ACTIVE USINGS - NONE
LOC OBJECT CODE ADDR1 ADDR2 STMT SOURCE STATEMENT
  77+  PDP    PRINT
  70+  SAVE   (14,12),T.*
  80+  SAVE   B 10(O,15)
  81+  DC     AL(4)
  82+  DC     CL4,'TEST'
  83+  STM    14,12,12(13)
  84+  LR     R12,R15
  85+  USING   TEST,R12
  86+  DC     AL(14)
  87+  SR     R9,R9
  88+  BAL    R11,SORT
  89+  WHILE  FALSE,R11,SORT
  90+  DC     X,'AO',AL(4)(FALSE,R11),S(SORT)
  91+  LM     R4,R5=A(NUMBERS,COUNT)
  92+  DISPLAY LINEDIT TEXT=".....",DOT=NO,SUB=(DECA,(R4))
  93+  DISPLAY 05 0H
  94+  DISPLAY 05 0H
  95+  BAL    1.OMSA0005
  96+  DC     AL1(16,129)
  97+  DC     AL1(DMS0005)

FLAG BYTES
02-DMSNLN
02-DMSNLN
O00025 05
O00026  484B4B4B4B
O0002C 00005
O0002C 1804
O0002E 0ACB
O00030 000C1
O00032 4140 4004
O00036 4650 C01E
O0003A 4510 C066
O0003E 10B1
O00040 B1
O00041 23
O00042 C6C103E2C54C303
O00056 00066
O00066 1809
O00068 0ACB
O0006A 0FF0
O0006C 98EC 000C
O00070 92FF 000C 000C
O00074 41F0 0000
O00078 07FF

TEST SORT ROUTINE
ACTIVE USINGS - (TEST.R12)
LOC OBJECT CODE ADDR1 ADDR2

+DMST0005 DC AL(DMSL0005) LENGTH OF MESSAGE TEXT
100+DMST0005 EQU +DMST0005-1 TEXT LENGTH
101+DMSCO005 DS OH SUBSTITUTION CODE
103+ SVC 203
104+ DC H’-6’ DISPLAY ALL THE NUMBERS
106 LA R4,(R4) TIMES’,DOT=ND,
107 BCT R5,DISPLAY
108 LINEDIT TEXT—FALSE CLAUSE EXECUTED TIMES’,DOT=ND,
SUB=(DEC,(R9))
109+ BAL 1,DMSAO037
110+ DC AL(16,129) FLAG BYTES.
111+ DC AL(DMSAO037)
112+ DC AL(DMSL0005)
113+ DC AL(DMSL0005)
114+ DMSL0005 EQU +DMSL0005-1 LENGTH OF MESSAGE TEXT
115+ DMSCO037 DS OH TEXT LENGTH
116+ DMSCO037 EQU 129 SUBSTITUTION CODE
117+ LR 0,R9
118+ SVC 203
119+ DC H’-6’ RESTORE THE REGISTERS
120 RETURN
121 RETURN (14,12),T,RC=0 SET RETURN INDICATION
122+ LM 14,12,(13) O1-RETUR
123+ MVI 12,(13),X’FF’ LOAD RETURN CODE
124+ LA 15,(0,0) O1-RETUR
125+ BR 14 RETURN

NEW XTESO4000

SLAG V2.9 ASM V Haven 16.54 10/27/83

SOURCE STATEMENT
127 * THIS ROUTINE PERFORMS ONE PASS THROUGH A LIST.
128 * IF THE LIST IS NOT SORTED, IT RETURNS FALSE AND SWAPS THE
129 * OFFENDING ELEMENT.
130 * OFFENDING ELEMENT.
132 SORT DS OH ASSEM TRUE
133 SR R6,R6.
134 LM R4,R5,=A(NUMBERS,COUNT-1) ADDRESS & COUNT
135 NEXTNUM L R2,(R4) GET THE FIRST NUMBER
136 C R2,(R4) COMPAR
137 IFTE HIGH,R10,SWAP,NOSWAP IF HIGH SWAP ELSE DON’T
138+ DC X’F4’,AL(4,HIGH,R10),S(SWAP,NOSWAP)
139 LA R4,(R4) RUN THROUGH THE ENTIRE LOOP
140 BCT R5,NEXTNUM.
142 LTR R6,R6 SET THE CONDITION CODE
143 BR R11 RETURN TO CALLER

NUMBERS OF TIMES TEST IS FALSE
145 NOSWAP LA R9,(R9)
146 BR R10
148 SWAP LA R6,1
149 MVC (O(4,R4),4(R4)
150 ST R2,(R4)
151 BR R10

TEST WORKAREAS, CONSTANTS, ETC....

-152-
EXTERNAL SYMBOL DICTIONARY

MACRO DEFINITION FOUND WITHIN INPUT STREAM
MEMBER(S) USED - IFT IFTE WHILE

CMSLIB ON MNT390
MEMBER(S) USED - DMSLN DMSLNC DMSLND DMSLNP DMSLNU DMSLNY DMSLNZ LINEDIT REGEQU

OSMACRO ON MNT390
MEMBER(S) USED - RETURN SAVE

TEST TEST NEW INSTRUCTIONS
ACTIVE USING - NONE
LOC OBJECT CODE ADDR1 ADDR2 STMT SOURCES STATEMENT
2 * THIS PROGRAM TESTS OUT THE 'IFT' AND 'WHILE' INSTRUCTIONS
3 * THE PROGRAM IS A SIMPLE BUBBLE SORT ON 20 RANDOM ELEMENTS.
4
6 MACRO
7 &LABEL IFT &MASK,&LINK,&SCON1,&SCON2
8 &LABEL DC 'X'4',AL.4(&MASK,&LINK),S(&SCON1,&SCON2)
9 MEND
11 MACRO
12 &LABEL WHILE &MASK,&LINK,&S
13 &LABEL DC 'X'AO',AL.4(&MASK,&LINK),S(&S)
14 MEND
16 MACRO
17 &LABEL IFT &MASK,&LINK,&S
18 &LABEL DC 'X'A1',AL.4(&MASK,&LINK),S(&S)
19 MEND

TEST BUBBLE SORT PROGRAM
ACTIVE USING - NONE
LOC OBJECT CODE ADDR1 ADDR2 STMT SOURCES STATEMENT
21 TEST START
22 HIGH EQU 'X'02'
23 FALSE EQU 'X'7F'
24 HIGH EQU 'X'07'
25 REGEQU
26 PUSH - PRINT
28** SYMBOLIC REGISTER EQUATES
30** GENERAL PURPOSE REGISTERS
32**
34+R0 EQU 0
35+R1 EQU 1
36+R2 EQU 2
37+R3 EQU 3
38+R4 EQU 4
39+R5 EQU 5

.155
Appendix F

DISASSEMBLED I.B.M. OBJECT CODE

This appendix contains the disassembled version of the second test program found in the previous appendix. This information was generated using a modified version of the HMASPZAP utility available on the OS/VS1 operating system. It was modified to add the three op-codes IFT, IFTE and WHILE.
**CCHHR- 005A001310  RECORD LENGTH- 0110  MEMBER NAME TEST  CSECT NAME TEST**

<table>
<thead>
<tr>
<th>O000000 47F0 F00A D4E3 C5E2 E300 90EC D00C 1BCF 1B99 45B0 C077 A07B C07A 9845 C100 4510</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BC SRP SPM STM LR BAF FFBA 1A4D 4004 4650 C01E 4510 C066 108</td>
<td></td>
</tr>
</tbody>
</table>

**LPR SH SH SH LR SVE LA STH BCT BAL LPR**

*FALSE CLAUSE EXECUTED TIMES EXECUTED*

**O00060 E3C9 D4C5 E200 1809 D4C5 1B99 3E3F 1A4D 3D50 C400 4B48 4B4B 4B4B 4B4B**

**E7C5 C3E4 E3C5 C440 4848 4B4B 4B4B 4B4B**

**SH SH SH SH**

**NC LR SVE LM**

**MVI LA BCR LR LM**

**L020 4000 5920 4004 A12A C09E 4140 4004 4650 C080 1266 07FB 4190 9001 07FA 4160**

**MVC STH STH STH STH BCR**

**BC 0000 0004 0000 0008 0000 0002 FFFF FFEE**

**0000 0001 0000 0012 0000 0000 0000 00064**

**0000 0006E 0000 000E 0000 0007 0000 0009 0000 0015 FFFF FFEE 0000 0009 0000 07CE BCR**

**0000 0010 0000 00B0 0000 0014 0000 0080 0000 0013**

**HMA1131 COMPLETED DUMP REQUIREMENTS**

**HMA1001 HMASPZAP PROCESSING COMPLETED**
BIBLIOGRAPHY


5. GoTo Statement Considered Harmful Dijkstra, Edsger W.; Letter to the Editor, Communication of the Association for Computing Machinery (CACM), March 1968.


