AN EMULATOR FOR THE E-MACHINE

by

Michael Leigh Birch

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

June 1990

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of material in this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

In the Master's thesis, "The E-machine: Supporting the Teaching of Program Execution Dynamics", Samuel D. Patton, presented the design of a virtual computer, called the E-machine, that was developed as the first component of a project to develop a comprehensive program animation environment for teaching and learning programming and other concepts fundamental to computer science. To support program animation activities in an easy and natural fashion, the E-machine has many unique features, including the capability of reverse execution. This thesis represents the next step in the program animation project. The E-machine is refined and an E-machine emulator is presented. The emulator is written in standard C and should thus be portable to many different computer types.

CHAPTER 1

INTRODUCTION

This thesis represents the second step in the development of a comprehensive program animation system intended to support the teaching and learning of programming and other concepts fundamental to computer science. The cornerstone of the program animation system is a virtual computer, called the E-machine (short for Education Machine), that incorporates many special features that will support program animation. Chief among these is the capability to execute programs in reverse. The E-machine was originally defined in the thesis, "The E-machine: Supporting the Teaching of Program Execution Dynamics," by Sam Patton [Patton 89] as the first step of the program animation project. In this thesis the E-machine is refined and an emulator for the E-machine, written in C, is given.

Preview

The thesis is organized into six chapters and one appendix. This is Chapter 1, which is intended to give an overview of the thesis, including structure, as well as explain terminology and notation that will be used throughout the thesis.

Chapter 2 is essentially a copy of Chapter 5 of Patton's thesis. It is included for the sake of clarity, as it describes the E-machine design in its entirety, with certain modifications made in this thesis, and must be available in updated form for further work on the program animation project. Major modifications to the E-machine design are marked by a leading asterisk (*) throughout Chapter 2. Note that no attempt is made to describe any differences between the modifications and the original design, so if the reader is interested in the differences, the two theses should be compared.

Chapters 3 and 4 describe the E-machine emulator. Chapter 3 explains the emulator design. This includes information about the logical components of the emulator program as well as information necessary to interface with the emulator to provide animation. Chapter 4 gives a demonstration of the operation of the emulator. A primitive interface is used to highlight the features of the emulator and how they apply to program animation. This chapter is not intended to describe the program animator or its user interface (they have yet to be developed), only to demonstrate the capabilities of the emulator.

Chapter 5 is a guide for compiler writers developing high level language translators for the E-machine. Chapters 3 and 4 together should contain sufficient information for designing a program animator based on the E-machine, but the E-machine design presented in Chapter 2 was not felt to be sufficient information for compiler writers. Chapter 5 thus gives additional insight into the design of the E-machine, which should make the compiler writer's job much easier.

Chapter 6 presents the status of the program animation project and expectations for future directions for the project. The finished product has not yet been completely defined, so only highlights of what some of the features might be are included in this chapter.

The code for the emulator, in its entirety, is included in Appendix A, along with a make file used to compile the emulator. As already noted, most of Chapter 2 is the work of Sam Patton, although the current author was involved in discussions of the E-machine design from the beginning. The remaining chapters represent original material and form the core of this thesis.

Terminology and Background

Due to the nature of this thesis, there is an abundance of new terminology used throughout. Most of it is explained at the appropriate time or is anticipated to be familiar to anyone that might read the

thesis. There is, however, a pseudo assembly language that is used throughout the thesis that deserves attention at this point. Note that the assembly language used is neither strictly defined or implemented in an assembler, it is merely a tool to present necessary information.

The rules for the language are simple. The language is made up of instructions, composed of four fields, each of which appears separately on a single line. The first field of an instruction is an opcode mnemonic, which denotes the operation of the instruction. The second field is a flag marking the instruction as critical or noncritical. The nature of this flag is explained in detail in Chapter 2. The third field denotes the data type to be used in the instruction and the fourth field is the operand field containing either a number or an addressing mode. Addressing modes and their formats are discussed in Chapter 2.

The mnemonic field is separated from the others by one or more spaces, and the remaining fields are separated by commas. The critical flag is a single letter, either c (for critical) or n (for noncritical). The data type is a single capital letter, I, R, C, A, or B, standing for Integer, Real, Character, Address, or Boolean respectively.

Note that not all of the instructions use all of the fields. Every instruction will have the mnemonic field, but any or all of the remaining fields may be omitted. Because of the easily discernible differences between fields, if a field is not appropriate for an instruction, it is merely left out. Note also, that fields 2 and 3 are left out completely in some examples, when they are not pertinent to the point being made and would only serve to confuse the issue. Anyone familiar with assembly languages in general should be able to understand the pseudo assembly language form used here without difficulty.

CHAPTER 2

THE E-MACHINE

This chapter is included for continuity and was taken virtually verbatim from Chapter 5 of Patton's thesis [Patton 89]. There have been some changes in the design of the E-machine which differ from the design presented in Patton's thesis; these differences are noted by a leading asterisk (*). The changes in design have been incorporated into the text and there is no discussion of the differences from the original design. Patton's thesis should be read for a complete background on the E-machine.

The Education Machine, or E-machine, is a virtual computer with its own machine language, called E-code. The task of the E-machine is to execute E-code translations of high level language (e.g., Pascal) programs. The real purpose of the E-machine, however, is to support a program animation system, as described more fully in [Ross 90], [Birch 90] and in Patton's thesis (there it was called a "dynamic display system"). This chapter focuses on the design of the E-machine, highlighting its special capabilities for supporting program animation activities.

Design Considerations

The part played by the E-machine in a program animation system is central to its design. The E-machine operates as follows. It is first loaded with a compiled E-code translation of a particular high level language source program. It then awaits a call from a driver program (the animator); this call causes a packet of E-code instructions corresponding to one high level language statement to be executed by the E-machine. Afterwards, control is returned to the animator, which performs the necessary animation activities before calling the E-machine again to have the next packet of E-code instructions executed. The E-machine thus acts as a dedicated microprocessor whose only purpose is to wait for a signal

from the animator and then execute a prescribed set of instructions based upon that signal. This definition of how the E-machine is to be used allows constraints to be placed upon its design that make the design process somewhat simpler.

As already noted, the E-machine is a virtual computer. The concept of a virtual computer is central to many computer science applications. Compilers and interpreters are the most common examples of systems designed around a virtual computer. The design of a virtual computer must take into account the purpose of the application. This helps to define and give structure and logic to the virtual computer. In the case of the E-machine, its purpose is to enable program execution dynamics of high level programming languages to be displayed easily by a program animator. This goal places some considerations upon the E-machine's design. Most importantly, the E-machine must:

1) Have structures for easy implementation of high level programming language constructs;
2) Incorporate a simple method for implementing functions, procedures, and parameters;
3) Be able to execute either forward or backward.

The driving force in the design of the E-machine is the requirement for backward, or reverse, execution. What does it mean for a computer to run in reverse? What does it mean for a high level language program to execute in reverse? As will be seen, these two questions have very similar and related answers, but they are not the same.

In a computer (virtual or real), the program counter, registers, main memory, and other status information can all be thought of as variables that change as the computer executes instructions. These variables can be collectively thought of as the "state" of the computer. If one knows the current state of a computer, one knows everything necessary for properly carrying out the next instruction to achieve the proper next state. In normal computers, however, the current state does

not contain enough information to reset the computer to a prior state. That is, most computers do not keep track of their history of execution. However, the computer's execution history is precisely what must be accessed in order to execute in reverse. How can this information be retained? The previous states must be recoverable. That is, given the present state of the computer, there must be a mechanism for changing this state to an arbitrary past state.

The brute force approach to solving this problem is to store each current state of the computer just before each new instruction is executed (all instructions change the state of a computer). Then, when the computer is to be restored to some prior state, all that has to be done is to load the computer with that state and the operation is done. With this method, the computer can be restored to an arbitrary prior state in one step.

The brute force method is unnecessarily powerful and also very inefficient. For example, this approach would require that all of main memory be stored with each state, even though at most one memory location would have changed from state to state as single instructions were executed. A better approach would be to have the computer save the minimal amount of information necessary to recover just the previous state from the current state in a given reversal step. The computer could then be restored to an arbitrary prior state by doing the reversal one state at a time until the desired prior state were obtained. For the purpose of the E-machine, this approach is sufficient.

Backing up one state at a time is a much simpler proposition than backing up to an arbitrary state in one step. Rather than storing the entire state of the computer at each step, it is only necessary to store the difference between the previous state and the current state. For example, suppose the instruction

```
pop   V2
```

pops the top value of the evaluation stack and places the value into variable register 2. No other registers would have been changed by executing this instruction, so the only changes to the state of the computer (in most computer models) would be to the value in V2, the program counter, and perhaps some status information. Saving these changes rather than the entire state of the computer takes much less memory, and in a real computer, memory is a valuable commodity. Therefore the E-machine was designed with this method of backing up in mind.

A natural question to ask at this point is whether it is possible to do even better: could the previous state be constructed directly from the current state without relying on some saved portion of the execution history? The answer is no. Consider an assignment instruction: an assignment instruction destroys the value in the register or memory location receiving the assignment; the value being destroyed must therefore be saved in order for backup to be possible.

One other aspect of program animation influenced the design of the E-machine. The animator is meant to work with high level language programs. This led to an important observation: the E-machine actually has to be able to reverse only high level language statements in one reversal step, not each individual low level E-code instruction involved in the translation of some high level language statement. In particular, the state of the E-machine has to be restored to the state it was in prior to the execution of the group of E-code instructions that are the translation of the corresponding high level language statement.

This observation led to further efficiencies in the design of the E-machine and to the incorporation of two classes of E-machine code instructions, critical and noncritical. As will be explained further later, an E-machine instruction is classified as critical if it destroys information essential to backing up through a high level language statement; it is classified as noncritical otherwise. In the translation of a high level language statement into E-code, a number of E-machine

instructions will be used only for dealing with intermediate values. For example, in a high level language arithmetic assignment statement, a number of intermediate values are likely to be needed in computing the arithmetic value on the right side of the assignment statement before this value can be assigned to the variable on the left. However, the only value that needs to be restored as far as the high level programming language is concerned upon backing up through this assignment statement is the original value of the variable on the left. The intermediate values computed by various E-code instructions are of no consequence. Hence, such instructions can be classified as noncritical and their effects ignored for backup purposes.

A particular E-code instruction can be classified as either critical or noncritical in different circumstances. Different high level languages will often have quite different statement sets, and what needs to be remembered for backup purposes may differ substantially from one language to another. It will be the responsibility of the compilers for each high level language to produce the correct E-code (involving critical and noncritical instructions) for allowing backup.

E-machine System Overview

With these considerations for backing up in mind it is now possible to describe the architecture of the E-machine in more detail. Figure 1 depicts the logical structure of the E-machine. After some deliberation, a stack-based architecture was chosen over other possibilities for its inherent simplicity. As can be seen, however, there are a number of components not found in real stack-based computers.

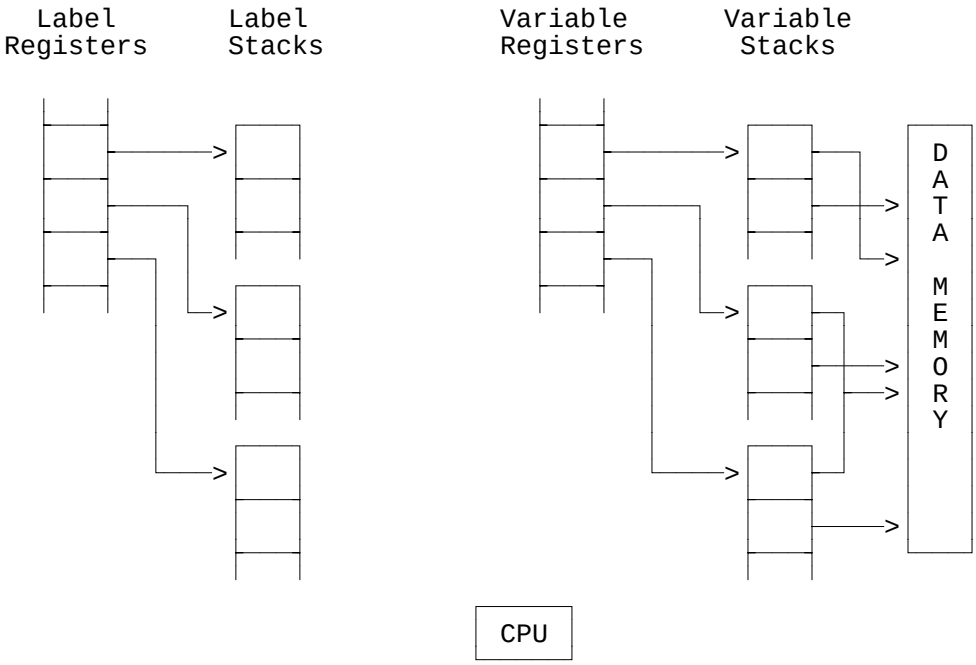


Figure 1
The E-machine

Program memory will contain the E-code program currently being executed by the E-machine. The program counter will contain the address in program memory of the current E-code instruction to be executed. The previous program counter, needed for backup purposes, will contain the address in program memory of the most recently executed E-code instruction.

*The packet register contains information about the next packet to be executed, or the packet that is currently executing, including the starting and ending line and column numbers of the original source program statement that is represented by the packet of E-code instructions about to be executed. Also included are the starting and ending program memory addresses for the packet, which are used internally to determine when execution of the packet is complete.

The variable registers are an unbounded number of registers that will be assigned to source program variables, constants, and parameters during compilation from the source program into E-code. Each identifier name representing memory in the source program will be assigned one variable register in the E-machine. As one can see in Figure 1, the variable registers only contain pointers to individual variable stacks, which in turn contain pointers into data memory, where the actual variable values are stored. The reason for this complex arrangement will become clearer as variables are discussed more thoroughly below.

The label registers are another unique component of the E-machine required for backup. There are also an unbounded number of these registers and, as described later, they are used to keep track of E-code label instructions in an E-code program for backup purposes. Each E-code label statement will be assigned a unique label register at compile time. A label register, in turn, points to a label stack that essentially maintains a history of previous instructions that caused a branch to the label represented by the label register in question.

The index register is found in real computers and serves the same purpose in the E-machine. Under normal circumstances, the data in a variable is accessed through the appropriate variable register. However, in the case of high level data structures, such as arrays and records, the address of an individual data value is not at the memory location directly accessible through a variable register. Rather, it is stored at a location offset from this memory location. When necessary, an offset value can be placed in the index register and the E-machine can then access the proper memory location as required (by any of the indexed addressing modes).

*The address register is provided to allow access to memory areas that are not directly accessible through variable registers. For example, a pointer in Pascal is a variable that contains a data address. Data at that address can be accessed using the variable indirect addressing mode (described later); however, if there are many levels of indirection, the address register must be loaded with a pointer value to continue accessing each level of indirection. The address register can used in place of variable registers for any of the addressing modes.

The evaluation stack pointer is also found in real computers. The evaluation stack pointer keeps track of the top of the evaluation stack. The evaluation stack is where the results of all arithmetic and logical operations and assignments are maintained. For example, in an arithmetic operation, the operands are pushed onto the stack and the operation is then performed on them. The operands are consumed by the operation and the result is pushed onto the top of the stack. Assignments are performed by popping the top value of the evaluation stack and placing it into a variable. The advantages of a stack architecture are well known; several popular computers use this design.

The return address stack (or call stack) pointer is a mechanism for implementing procedure and function calls. When a call is made to an E-machine subroutine, the program counter plus one is pushed onto the

return address stack. Then, when the E-machine executes a return from subroutine instruction, all it has to do is load the program counter with the top of the return address stack.

The save stack pointers point to the top and bottom of the save stack, which stores information required for backup that would otherwise be lost. Whenever some critical information (as determined by the execution of a critical instruction) is about to be destroyed, the required information is pushed onto the save stack. This ensures that when backing up, the instruction that most recently destroyed some critical information can be reversed by retrieving that critical information from the top of the save stack.

Finally, data memory represents the usual random access memory found on real computers, but in the E-machine it is only used for holding data values (it does not hold any of the program instructions). In real computers, a similar situation exists in some systems which provide for separate code and data segments in memory. On the E-machine, there is no bound on the available memory (or any of the stack memories). Implementations on real computers will naturally enforce some bounds, but for the academic (small program) environment envisioned for this system, no practical problems are expected to be encountered due to limited memory.

E-machine Instruction Set

The E-machine's instruction set is a quite small but complete set of instructions; these instructions allow an E-code program to access data easily and simply. All arithmetic, logical, and assignment operations occur on the evaluation stack. Data is stored and recalled using the variable registers and, possibly, the address register.. All operations for backing up occur with a minimum of information from the E-code program in question (in general, all the E-code program has to do is utilize the

correct form of the instruction--critical or noncritical--to ensure that backing up can occur correctly).

Instruction Set

This section lists all of the instructions in the instruction set of the E-machine. The argument ADDR refers to any addressing mode described in the next section. The argument TYPE refers to any of the data types integer, real, boolean, char, or address; most instructions require that the type of data being operated upon be specified. The # refers to an integer constant specifying the number of an E-code label or an E-machine variable register. The CFLAG argument must be either c or n and designates whether the instruction is to be treated as critical (c) or noncritical (n). Backing up through a noncritical instruction often still requires that something be pushed onto the evaluation stack to keep the stack of the proper size; in such cases an arbitrary value, called DUMMY is used.

push ADDR, TYPE:

Pushes the value in ADDR onto the evaluation stack.

Forward:
Pushes the value in ADDR onto the evaluation stack.

Backward:
Pops the top value of the evaluation stack and stores it in ADDR.

*pusha ADDR:

Pushes the calculated address of ADDR onto the evaluation stack. This instruction is intended to be used for pushing the addresses of parameters passed by reference onto the evaluation stack.

Forward:
Pushes the calculated address of ADDR onto the evaluation stack.

Backward:
Pops and discards the address on top of the evaluation stack.

pop CFLAG, ADDR, TYPE:

Pops the top value of the evaluation stack and places it in ADDR.

Forward-Critical:
Pushes the value in ADDR onto the save stack and then pops the top value of the evaluation stack and stores it in ADDR.

Forward-Noncritical:
Pops the top value of the evaluation stack and stores it in ADDR.

Backward-Critical:
Pushes the value in ADDR onto the evaluation stack and then pops the top value of the save stack and places it in ADDR.

Backward-Noncritical:
Pushes the value in ADDR onto the evaluation stack.

*popar CFLAG:

Pops the address on top of the evaluation stack and places it in the address register.

Forward-Critical:
The contents of the address register are pushed onto the save stack. The address on top of the evaluation stack is popped off and placed in the address register.

Forward-Noncritical:
The address on top of the evaluation stack is popped off and placed in the address register.

Backward-Critical:
The contents of the address register are pushed onto the evaluation stack. Then the address on top of the save stack is popped off and placed in the address register.

Backward-Noncritical:
The contents of the address register are pushed onto the evaluation stack.

*popir CFLAG:

Pops the integer on top of the evaluation stack and places it in the index register.

Forward-Critical:
The contents of the index register are pushed onto the save stack. Then the integer on top of the evaluation stack is popped off and placed in the index register.

Forward-Noncritical:
The integer on top of the evaluation stack is popped off and placed in the index register.

Backward-Critical:
The contents of the index register are pushed onto the evaluation stack. Then the integer on top of the save stack is popped off and placed in the index register.

Backward-Noncritical:
The contents of the index register are pushed onto the evaluation stack.

*loadar CFLAG, ADDR:

Places the address ADDR in the address register.

Forward-Critical:
The contents of the address register are pushed onto the save stack. Then the address computed for the addressing mode is placed in the address register. Important note: it is the address that is computed by the addressing mode that is used, not the contents of that address.

Forward-Noncritical:
The address computed for the addressing mode is placed in the address register. Same note for Forward-Critical applies here.

Backward-Critical:
The address on top of the save stack is popped off and placed in the address register.

Backward-Noncritical:
Nothing happens.

*loadir CFLAG, #:

Places the # into the index register.

Forward-Critical:
The contents of the index register are pushed onto the save stack. Then # is placed in the address register.

Forward-Noncritical:
is placed in the index register.

Backward-Critical:
The value on top of the save stack is popped off and placed in the index register.

Backward-Noncritical:
Nothing happens.

add CFLAG, TYPE:

Adds the top two values on the evaluation stack and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes them onto the save stack, and then pushes their sum onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes their sum onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards the value. Pops the top two elements of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

sub CFLAG, TYPE:

Subtracts the second value from the top of the evaluation stack from the first and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value minus the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack, and pushes the bottom value minus the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

mult CFLAG, TYPE:

Multiplies the top two value on the evaluation stack and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes their product onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes their product onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

div CFLAG, TYPE:

Divides the second value from the top of the evaluation stack by the first and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and pushes the bottom value divided by the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes the bottom value divided by the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

neg TYPE:

Negates the top value on the evaluation stack.

Forward:
Pops the top of the evaluation stack and pushes the negation of that value onto the evaluation stack.

Backward:
Pops the top of the evaluation stack and pushes the negation of that value onto the evaluation stack.

*and CFLAG, TYPE:

Bitwise and's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise and'ed with the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes the bottom value bitwise and'ed with the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

*or CFLAG, TYPE:

Bitwise or's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise or'ed with the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes the bottom value bitwise or'ed with the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

*xor CFLAG, TYPE:

Bitwise exclusive-or's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise exclusive or'ed with the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes the bottom value bitwise exclusive or'ed with the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

*not CFLAG, TYPE:

Bitwise complements the top value of the evaluation stack.

Forward:
Pops the top of the evaluation stack and pushes the bitwise not of that value onto the evaluation stack.

Backward:
Pops the top of the evaluation stack and pushes the bitwise not of that value onto the evaluation stack.

*shl CFLAG, TYPE, #:

Shifts the value on top of the evaluation stack # bits to the left filling on the right with 0's.

Forward-Critical:
Pops the top value of the evaluation stack, pushes it onto the save stack, then shift it # bits to the left and pushes the result back onto the evaluation stack.

Forward-Noncritical:
Pops the top value of the evaluation stack, shifts it left # bits, then pushes the result back onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack. The pops the top value of the save stack and pushes it onto the evaluation stack.

Backward-Noncritical:
Nothing happens.

*shr CFLAG, TYPE, #:

Shifts the value on top of the evaluation stack # bits to the right filling on the right with 0's.

Forward-Critical:
Pops the top value of the evaluation stack, pushes it onto the save stack, then shift it # bits to the right and pushes the result back onto the evaluation stack.

Forward-Noncritical:
Pops the top value of the evaluation stack, shifts it right # bits, then pushes the result back onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack. The pops the top value of the save stack and pushes it onto the evaluation stack.

Backward-Noncritical:
Nothing happens.

mod CFLAG, TYPE:

Finds the remainder of the division of the second value from the top of the evaluation stack by the first and places the result onto the evaluation stack.

Forward-Critical:
Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value modulo the top value onto the evaluation stack.

Forward-Noncritical:
Pops the top two values of the evaluation stack and pushes the bottom value modulo the top value onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

cast TYPE, TYPE:

Changes the top value of the evaluation stack from the first TYPE to the second.

Forward-Critical:
Pops the top value of the evaluation stack and pushes it onto the save stack, then transforms the value from the first TYPE to the second. The result is pushed onto the evaluation stack.

Forward-Noncritical:
Pops the top value of the evaluation stack, then transforms the value from the first TYPE to the second. The result is pushed onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack. The pops the top value of the save stack and pushes it onto the evaluation stack.

Backward-Noncritical:
Nothing happens.

*write CFLAG, TYPE:

Displays a value for the user.

Forward-Critical:
The top of the evaluation stack is popped and the value pushed onto the save stack. This value is then converted into a string and passed to a user interface function which takes appropriate action to display the value.

Forward-Noncritical:
The top of the evaluation stack is popped and is converted into a string and passed to a user interface function to be displayed.

Backward-Critical:
The value on top of the save stack is popped and pushed onto the evaluation stack. Then a user interface function is called to handle undisplaying of the last value displayed.

Backward-Noncritical:
DUMMY is pushed onto the evaluation stack then a user interface function is called to handle undisplaying of the last value displayed.

*read CFLAG, TYPE:

Reads a value from the user.

Forward:
A user interface function is called to get input from the user. The input is converted from a string to the appropriate type and pushed onto the evaluation stack.

Backward:
The top value is popped off the evaluation stack.

label #:

Marks the location to which a branch may be made.

Forward:
Pushes the previous program counter onto the stack pointed to by label register #.

Backward:
Pops the top value of the stack pointed to by label register # and places it in the program counter.

br #:

Unconditionally branches to label #.

Forward:
Load the program counter with the address of the label # instruction.

Backward:
No operation.

*eql, neql, less, leql, gtr, geql CFLAG, #:

If the second value from the top of the evaluation stack compares favorably with the first, then TRUE is pushed onto the evaluation stack. Otherwise FALSE is pushed onto the evaluation stack.

Forward-Critical:
Pops the top two values off the evaluation stack, pushes the two values onto the save stack, compares the bottom value with the top. If the result of the comparison matches the comparison operation performed, a boolean TRUE is pushed onto the evaluation stack, otherwise, a boolean FALSE is pushed onto the evaluation stack.

Forward-Noncritical:
Pops the top two values off the evaluation stack and compares the bottom value with the top value. If the result matches the comparison operation performed, a boolean TRUE is pushed onto the evaluation stack, otherwise, a boolean FALSE is pushed onto the evaluation stack.

Backward-Critical:
Pops the top value of the evaluation stack and discards it, then pops the top two values off the save stack and pushes them onto the evaluation stack.

Backward-Noncritical:
Pushes DUMMY onto the evaluation stack.

*brt, brf CFLAG, #:

Conditionally branches depending on whether the top of the evaluation stack is TRUE or FALSE.

Forward-Critical:
Pops the top value off the evaluation stack and pushes it onto the save stack. If the value satisfies the conditional on the branch (TRUE for brt, FALSE for brf), the program counter is loaded with the address of the label # instruction.

Forward-Noncritical:
Pops the top value off the evaluation stack. If the value agrees with the conditional branch (TRUE for brt, FALSE for brf), the program counter is loaded with the address of the label # instruction.

Backward-Critical:
Pops the top value of the save stack and pushes it onto the evaluation stack.

Backward-Noncritical:
Arbitrarily pushes DUMMY onto the evaluation stack.

call #:

Branches to label # saving the program address which follows the call instruction so that execution will continue there upon execution of a return instruction.

Forward:
Pushes the current program counter onto the return address stack, then loads the address of the label # instruction into the program counter.

Backward:
No operation.

return:

Returns to the appropriate program address following a call instruction.

Forward:
Pops the top value of the return address stack and loads it into the program counter.

Backward:
No operation.

*alloc CFLAG, #:

Allocates a block of memory of # size.

Forward:
Attempts to allocate # computer words of storage. If successful, the address of the first word of data memory that was allocated is pushed onto the evaluation stack. Otherwise, a NULL address is pushed onto the evaluation stack.

Backward:
Pops the top value off the evaluation stack, which should be a data address, and frees # words of data memory starting at that address.

*unalloc CFLAG, #:

Deallocates a block of memory of # size beginning at the data address atop the evaluation stack.

Forward-Critical:
Pops the top value off the evaluation stack, which should be a data address, copies # words of data memory starting at that address to the save stack, then frees the data memory.

Forward-Noncritical:
Pops the top value off the evaluation stack, which should be a data address, and frees # words of data memory starting at that address.

Backward-Critical:
Pops the top value off the save stack, which should be a data address, pushes it onto the evaluation stack and allocates # words of data memory starting at that location. # words are then moved from the save stack to this data memory.

Backward-Noncritical:
Allocates # words of data memory and pushes the address of the first word of allocated memory onto the evaluation stack.

*inst CFLAG, V#:

Creates an instance of the variable register #.

Forward-Critical:
Allocates enough data memory for the variable represented by the variable register #. The address of the allocated memory is then pushed onto the variable register’s stack.

Forward-Noncritical:
Allocates enough data memory for the variable represented by the variable register #. The size of the variable is stored in the variable register. The address of the allocated memory is then pushed onto the variable register’s stack.

Backward-Critical:
The data memory occupied by the variable register is freed and the top value is popped off the variable register’s stack.

Backward-Noncritical:
Frees the space taken up by the variable in data memory and pops the top value off the variable register's stack.

*uninst CFLAG, V#:

Dispose of an instance of variable register #.

Forward-Critical:
Pushes the variables data onto the save stack, frees the memory occupied by the variable then pops the top data memory address off the variable register's stack and pushes it onto the save stack.

Forward-Noncritical:
Frees the memory occupied by the variable then pops the top address off the variable register's stack.

Backward-Critical:
Pops the address off the save stack and pushes it onto the variable register's stack, reallocates enough data memory for the variable # starting at that address, then pops the variables data off the save stack and places it the address.

Backward-Noncritical:
Reallocates enough data memory for the variable # and pushes the address of the data memory allocated onto the variable register's stack.

link #:

Associates one variable register with the value of another.

Forward:
Pops the top value of the evaluation stack and pushes it onto the variable stack pointed to by variable register #.

Backward:
Pops the top value of the variable stack pointed to by variable register # and pushes it onto the evaluation stack.

unlink #:

Disassociates a variable register from another.

Forward:
Pops the top value of the variable stack pointed to by variable register # and pushes it onto the save stack.

Backward:
Pops the top value of the save stack and pushes it onto the variable stack pointed to by variable register #.

*nop:

This instruction does absolutely nothing except take up space. It is intended to be used to create packets for program statements that don't generate any instructions, but should be highlighted during execution.

Addressing Modes

In this section, the various addressing modes available in the E-machine instruction set are given. Quite a few modes are defined in order to accommodate standard high level language data structures more conveniently. Note that each addressing mode refers to either the data at the computed address or the computed address itself, depending on the instruction. That is, for those instructions that need a data value, such as push, the data value at the address computed from the addressing mode is used. For instructions that need an address, such as pop, the address that was computed for the addressing mode is used.

For each addressing mode listed below, an example of its intended use is given. Each example is given in pseudo assembly language form for clarity; it is important to remember that no assembler (and hence no assembly language) has yet been developed for the E-machine. However, the pseudo assembly language examples should be easily understood. An explanation of the arguments and their meanings were given in the introduction.

constant mode - C#:

This mode is often called the immediate mode in other architectures; # is itself the integer, real, boolean, character, or address constant operand required in the instruction.

Example:

```
A := 1.5;
```

could be translated into:

```
push        R,C1.5      ; push 1.5
pop         c,R,V1      ; assign to A
```

variable mode - V#:

variable register # -> top of variable stack -> data memory

This mode accesses the data memory location given in the top element of the variable stack that is pointed to by variable register #. This mode is intended to address source program variables that are of one of the basic E-machine types.

Example:

```
B := 1;
```

could be translated into:

```
push        I,C1        ; push 1
pop         c,I,V3      ; assign to B
```

*variable indirect - (V#):

variable register # -> top of variable stack -> data memory -> data memory

This mode accesses the data in data memory whose location is stored at another data memory location, which is pointed to by the top of the variable stack pointed to by variable register #. This mode is intended for accessing the contents of a high level language pointer variables. It would be particularly useful for handling parameters in C which are passed as pointers for the intention of passing parameters by reference.

Example:

```
int foo( C )
int *C
{
  *C = 1;
}
```

could be translated into:

```
label       c,5         ; procedure entry
inst        c,V3        ; create new instance of C
pop         c,A,V3      ; assign argument passed to *C
push        I,C1        ; push 1
pop         c,I,(V3)    ; assign to *C
uninst      c,V3        ; destroy instance of C
return                  ; return from call
```

*variable offset mode - V#{offset}:

variable register # -> top of variable stack + IR -> data memory

This mode accesses the data pointed to by the top of the variable register # stack plus a byte offset which was previously loaded into the index register. This mode is useful for accessing fields in a structured data type such as a Pascal record or C struct.

Example:

```
A := D.Field2
```

could be translated into:

```
        push        I,2         ; D is at offset of 2 in structure
        popir       c           ; put offset into index register
        push        R,V4{IR}    ; push D.Field2
        pop         c,R,V1      ; assign to A
```

*address indirect - (A):

address register -> data memory

This mode provides access to data located at the data address in the address register. The address register must be loaded with a data memory address which points to data memory. This mode is useful for multiple indirection.

Example:

```
c = *(*g);
```

could be translated into:

```
loadar      c,V7        ; load addr reg with addr of g
loadar      c,(A)       ; load addr reg with addr of *g
push        I,(A)       ; push *(*g)
pop         c,I,V3      ; assign to c
```

*address offset mode - A{offset}:

address register + IR -> data memory

This mode provides access to structured data through the address register. The index register is added to the address register to provide an address to the data to be accessed. This mode is useful for indirection with structured data, such as pointers to records in Pascal.

Example:

```
I := H^.Data
```

could be translated into:

```
push        A,V8        ; push H^ (address value of H)
popar       c           ; load ar with H^
push        I,C2        ; Data has offset of 2 in record
popir       c           ; load ir with offset
push        I,A{IR}     ; push H^.Data
pop         c,I,V9      ; assign to I
```

*variable indexed mode - V#[index]:

variable register # -> top of variable stack + IR * datasize -> data memory

This address mode uses the top of the variable register # stack as a base address and adds the index register, which must be previously loaded, multiplied by the number of bytes occupied by the data type, which is a basic E-machine data type. The resulting address points to the data item. This mode is useful for accessing an array whose elements are of a basic E-machine data type.

Example:

```
B := L[3];
```

could be translated into:

```
push        n,I,3       ; put index of 3 into
popir       c           ; the index register
push        I,V12[IR]   ; push L[3]
pop         c,I,V2      ; assign to B
```

*address indexed mode - A[index]:

address register + IR * datasize -> data memory

This mode provides the same function as variable indexed mode, except instead of a variable register providing the base address, the address register is loaded with the base address. This mode could be used for accessing elements of an array which is pointed to by a variable.

Example:

```
B := S^[4];
```

could be translated into:

```
push        A,V19       ; put address of array into
popar       c           ; address register
push        I,4         ; put index of 4 into
popir       c           ; the index register
push        I,A[IR]     ; push S^[4]
pop         c,I,V2      ; assign to B
```

Source Program Variable Representation in E-machine Code

Understanding how the E-machine provides for the implementation of high level source language variables is vital to understanding the operation of the E-machine, especially in backing up. (In this context, the term variable refers to any identifier in the source program that requires memory, including, for example, constants, and parameters.) First, a compiler that generates E-code translations of, say, Pascal programs, assigns each variable in the Pascal program a unique E-machine variable register. This is done statically at compile time, so that every variable is associated with a unique variable register for the duration of program execution, regardless of whether that variable is currently active or not. The variable register for a variable does not contain the value of the variable. Rather, it contains a pointer to a unique variable stack for that variable (look at Figure 1 again). Since each variable register

is really only a pointer, it will be the same size regardless of whether the variable is a simple variable or, for example, an array.

The variable stack pointed to by a variable register also does not contain the value of the variable. In this case, each element of the variable stack is itself a pointer to the actual variable value in data memory. The stack is necessary because a particular variable may have multiple associated instances. Consider the case of a variable A that is local to a recursive Pascal procedure. Each new recursive call to that procedure would require that a new data memory location be set aside for new instance of A. A's variable register would point to A's variable stack, and the top of A's variable stack would point to the value of the current instance of A in data memory. The second stack element would point to the previous instance of A in data memory, and so on. Most variables are not in recursive procedures and thus will only have at most one instance during program execution. In such cases, the variable register would point to a variable stack that is just one element deep. The case for a variable A with just a single instance is illustrated in Figure 2. Figure 3 shows the situation of a variable A having three instances as the result of three recursive calls to a procedure.

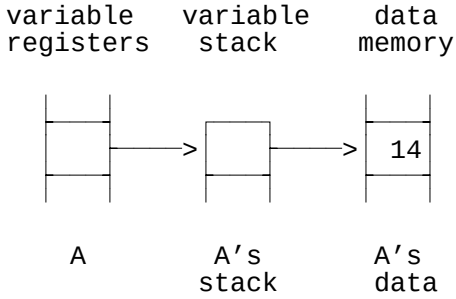


Figure 2
E-machine Global Variable Implementation

Whenever a procedure or function exits, the compiled E-code will ensure that local variable instances are properly removed from data memory by simply causing the top of the variable stacks to be popped for each

affected variable. If a variable is totally deactivated as a result, its variable register will simply point to an empty variable stack.

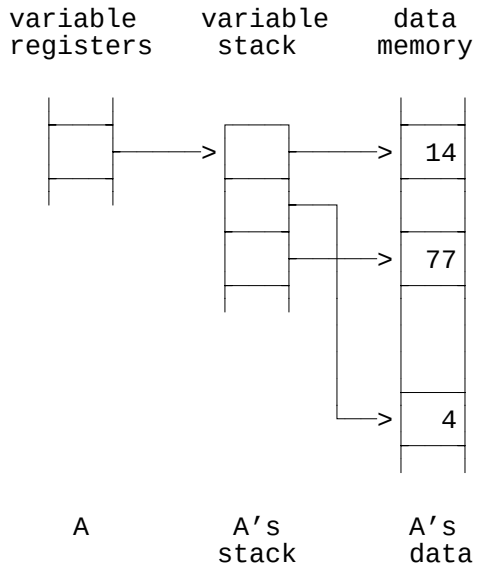


Figure 3
E-machine Recursive Variable Implementation

Notice that arrays and records can be handled in the usual fashion, using offsets (in the index register) from the first location for the variable in data memory to arrive at individual elements.

The Save Stack

To see how backing up is accomplished with this method of representing variables, the role of the save stack must be explained. The first things to consider are the kinds of information associated with a variable. There are two kinds: the location of the variable's memory and the data in the variable in data memory location. Both are subject to destruction or loss during normal program execution. It is easier to see how the second type of information, the data, can be destroyed. Whenever an assignment is made into a variable, the old data in the variable's memory location is destroyed. Therefore, in order to restore the E-machine's state to the state prior to the assignment, it is necessary to save the old data. This is done on the save stack. Upon backing up, the

old variable value can then be restored by retrieving it from the save stack.

Now consider the case of a memory location. Recall that the data memory location of a variable is kept in the stack corresponding to that variable. In the case, say, of a Pascal global variable, the single stack element for that variable continues to point to the proper data memory location for that variable throughout the execution of the program. In the case of a variable (again, this refers to both local variables and parameters) in a procedure or function, however, the data memory location, and hence the pointer to this location on the variable stack, may change with each call. That is, each time a call to the procedure or function is made, a different data memory location may be allocated for the value of the variable and pushed onto the top of that variable's stack; upon return from the procedure or function, that address will be popped off the variable's stack as the variable is deallocated. At this point, information critical to backup would be lost if the address popped off were not saved in some way.

This, again, is where the save stack comes in. Whenever any information is about to be lost in either of the above fashions, the information instead is pushed onto the save stack. Figure 4 shows the initial variable register, variable stack, and data memory location for a variable X. Also included is the save stack.

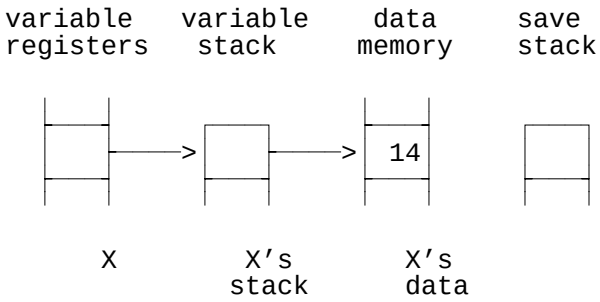


Figure 4
Variable and Save Stack for a Variable X

Notice that the save stack is empty now. (Technically, this situation could not arise since X has a value at this point, and therefore at some point in the past must have had an assignment statement performed upon it that would have required the old value to be pushed onto the save stack. For purposes of this example, we will ignore this fact.) Now, let's perform an assignment operation:

X := 27

The effect this will have upon the E-machine's structures is shown in Figure 5. Notice that the top of the save stack now contains the old value of X and that the new value of X is stored in the old memory location. In this case, the information that would have been destroyed was the data, not the memory location.

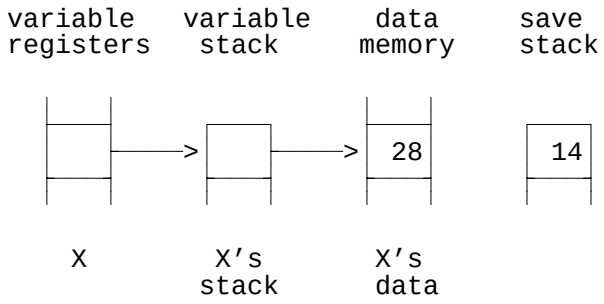


Figure 5
Variable and Save Stack After Assignment to X

In order to back up at this point, all that would be necessary would be to pop the top of the save stack and place the popped value into the memory location pointed to by the top of X's variable stack. This procedure allows any assignment to be reversed.

Preparing for the reverse execution of statements that lose location information is somewhat more involved. To understand this task better, recall from the previous section how the E-machine architecture provides for the implementation of high level language variables (remember, too, that the term variable is used here to stand for any high level language identifier requiring memory, such as actual variables, parameters, and constants). Each variable has a permanently assigned variable register.

Each variable register points to a unique, associated variable stack. Each element of the variable stack is a pointer to an instance of the variable value in data memory; the top of the variable stack points to the current, active instance of the variable.

Thus, since the location of a variable's assigned variable register remains constant throughout program execution, and this variable register always points to the current top element of the associated variable stack, the only location information that can be lost during normal forward execution is the location of a variable's value in data memory as a procedure or function is exited (i.e., the value on top of the variable stack for the variable). Thus, upon exit of a procedure or function, when the values of local variables (including parameters and constants) and their locations in data memory are normally lost, the locations of these variables, must be saved on the save stack. When backing up through a procedure or function call, (i.e. executing the procedure or function in reverse), the original locations of the local variables in data memory can be restored to the top of their respective variable stacks from the save stack.

How can one be certain that the original variable values will be in the restored locations upon backing up? Consider how a value in data memory is changed. The only way this can happen is through an assignment operation. But earlier in this chapter, a mechanism was introduced that allowed an assignment instruction to be reversed. Therefore, even though a memory location may have been assigned numerous different values since the procedure exited, as backing up occurs, that memory location will have been reset to the required value by the time the procedure is encountered in reverse. Consider the example Pascal program fragment in Figure 6. It consists of a header definition for procedure <u>something</u> with one value parameter; there are three calls to that procedure from another routine. The lines labeled 0,1,2, and 3 have corresponding sections in Figure 7. Figure 7 contains the variable structures and the save stack that

correspond to each of the procedure calls in Figure 6. The section labeled 0 in Figure 7 refers to the state of the structures before any procedure call has been executed. Notice that the save stack is empty, and notice too that the variable stack for P is also empty.

```
    Procedure something(P : integer);
              .
              .
              .
0
1       something(7);
2       something(5);
3       something(-16);
```

Figure 6
A Pascal Procedure Fragment something

Now let's examine line 1 in Figure 6. This is the first call to procedure something. Notice that during the call, P's stack in section 1 of Figure 7 now has a value, 1, which is a location in data memory. Notice also that the data memory location which this points to (data memory location 1) contains the value of the parameter which was passed to something. During this call, any references to P are referring to the data memory location that is pointed by the top of P's variable stack.

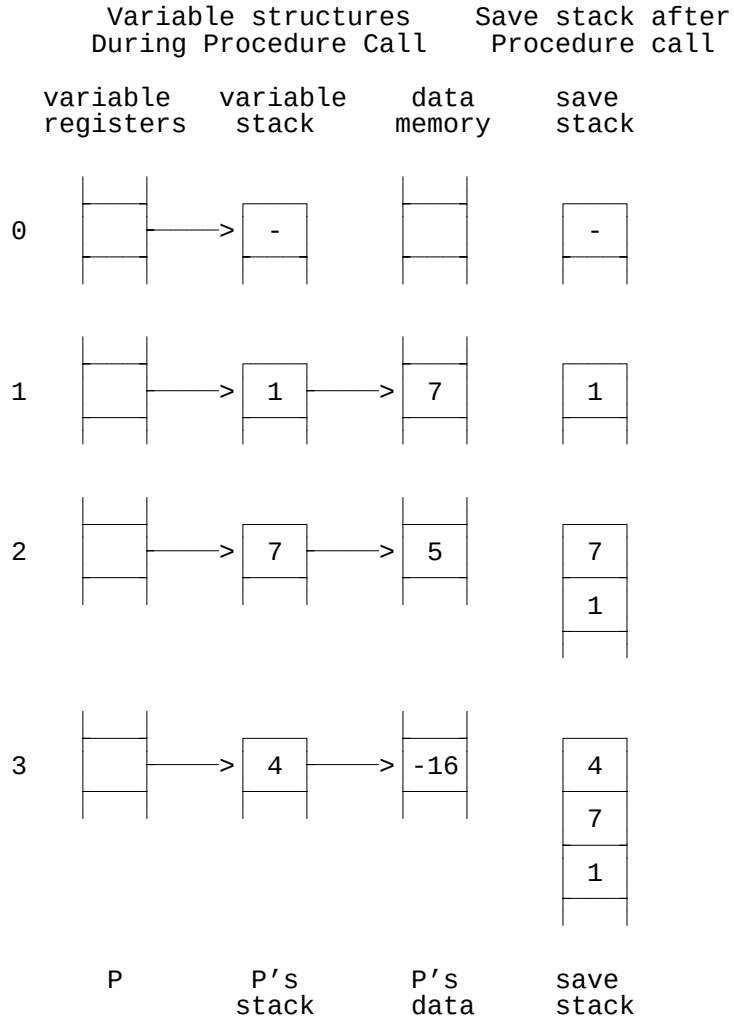


Figure 7
Variable and Save Stack During Successive Calls to Procedure something

The save stack in section 1 of Figure 7 shows the state of the stack after procedure something finishes executing for the first time. Notice that it now has the value 1 on top. This is because when the procedure exited, the data memory location to which P was pointing would have been lost, so it was saved by the E-machine on the top of the save stack. Consider now the second call to something in line 2 of Figure 5 and compare it with section 2 in Figure 7. The top of P's variable stack now contains the value 7, which points to data memory location 7, which contains a 5. Now any references to P will refer to memory location 7 (i.e., to the value 5). When procedure something is exited this time, the

top of the variable stack would again be lost if it were not saved. Thus, the 7 on top of the variable stack for P is pushed onto the save stack by the E-machine, resulting in the save stack configuration of section 2 in Figure 7.

The third call to procedure something follows in exactly the same manner. When the procedure is executing, the variable P refers to the data memory location contained on the top of P's variable stack, and when the procedure ends, that data memory address is pushed onto the save stack.

Reversing through these procedure calls simply consists of popping the addresses off the save stack and pushing them onto P's variable stack when reversing through the procedure's exit, and popping the top of P's variable stack when reversing through the procedure's entrance.

The Label Registers

Execution in a program is not a simple, linear affair. There are branches, calls to subroutines, returns from subroutines, and other non-sequential types of instructions that add complexity to the problem of backing up. We have seen how to reverse many E-machine instructions by utilizing the save stack. What we haven't yet determined is how to reset the current program counter so that it points to the proper previous instruction. If the current instruction was arrived at from some instruction other than the immediately preceding instruction (e.g., via a branch instruction) there must be some method available for recovering the line number of the instruction branched from.

For example, Figure 8 gives a simple E-code program (for clarity, variables are referred to by name rather than their variable registers, and addressing modes, data types, and critical and noncritical designators have been omitted). The program does the following: I's value is pushed onto the evaluation stack followed by J's value. The eql instruction of line 3 then compares the top two stack values, consuming these values, and

pushing the result of the comparison onto the stack. Notice line 4; if the top of stack value denotes "true", a branch must be made to the label 1 instruction, which is in line 7 (that is, the current program counter must be set to 7). Otherwise execution proceeds sequentially through lines 5 and 6 until line 7 is reached.

The label instruction of line 7 is the interesting instruction in this case. As seen, depending on the values of I and J, the instruction executed just previous to line 7 could have been either line 4 or 6. How can it be determined for backing up which one really did precede line 7?

The brute force method of solving this problem is simple but very inefficient. If, at each step, the current program counter is stored on a stack, all that is needed to restore the current program counter upon backing up is to replace its value with the top of stack value. This method

```
1   push  I
2   push  J
3   eql
4   brt   1
5   push  I
6   pop   J
7   label 1
8   halt
```

Figure 8
Simple E-code Program With a Branch

will work, but it is inefficient for the following reason. Most instructions in a program have only one possible previous instruction, the one that directly precedes it in the program. In the example of Figure 8, only line 7 has more than one possible previous instruction. All of the other instructions have only one. A more elegant and efficient method to solving this problem, then, is to identify the instructions with more than one possible previous instruction (referred to hereinafter as "branch points") and only save the previous program counter when one of these instructions is executed in the forward direction. In order to do this, branch point instructions must be identifiable.

How can branch points be identified by the E-machine as it executes an E-code program? The characteristics of a branch point are easy to categorize. A branch point is any instruction that can be executed in some order other than sequentially from the instruction immediately preceding it. Most such instructions can be readily identified: since both branch and call instructions require a label as one of their arguments, any instruction that is a branch point because of a branch or call must be an E-code label instruction. This leaves one class of branch points still unidentified, those arrived at by a return from a procedure or function. The return instruction does not--and indeed cannot--have a label as an argument; instead, control must be returned to the instruction immediately following the call that invoked the procedure or function (the utility of a procedure or function lies in the fact that it can be called from anywhere and, after execution, will return to the instruction immediately after the call).

From the above discussion, it is clear that each E-code instruction that immediately follows a procedure or function call is a branch point. However, examining an arbitrary E-code instruction in isolation does not allow one to determine whether the previous instruction was a call. Thus, some sort of mechanism must be employed to mark such an instruction as a branch point at compile time. Since all branch points except those arrived at by a return are E-code label instructions in any case, the same technique can be employed to branch points arrived at by a return. The compiler can simply be designed to generate an E-code label instruction immediately following each procedure or function call.

This technique ensures that all branch points are E-code label instructions. Thus, for successful backup, when the E-machine executes a label instruction in the forward direction, it must save the previous program counter value in some fashion. Recall that in the E-machine, the previous program counter is always maintained in the register by that name. Every time the current program counter is changed, its old value is

first placed into the previous program counter. (Notice that this structure is not a stack. Only one value is stored at any one time in the previous program counter.)

In order to save the previous program counter for successful backup, then, whenever an E-code label instruction is executed, the E-machine employs its label stacks and label registers (see Figure 1). Each label instruction is to be assigned a label register at compile time, where each label register is a pointer to a unique label stack (the reason for the stack is given later). Thus whenever a label instruction is encountered by the E-machine, the value in the previous program counter is pushed onto the stack referenced by that label's register.

Now, look at the example program given in Figure 9. There are two branch points in this program: line 1 and line 11. This program contains a loop in which lines 1 through 10 are executed until I and J are equal. Obviously, this loop could iterate a large number of times, and each time the label instruction of line 1 is executed, it appears that the previous program counter should be pushed onto the label stack of label 1. However, except for the very first time line 1 is executed, the previous program counter will always contain 10. There should be a way to take advantage of this repetition and save some space.

The E-machine does this in the following way. Each element of the label stack associated with each branch point has two parts, one for holding the value of the previous program counter and one for holding a count, as shown in Figure 10. Rather than just pushing the previous program counter onto the label stack when a label instruction is executed by the E-machine, the E-machine first compares the previous program counter to the number stored in the top element of the label stack. If these two values are equal, the associated counter on the stack is simply incremented, thus recording the number of times this label instruction was reached from the same previous instruction. Thus, rather than storing n identical previous program counter values, where n is the number of times

the loop is iterated, only one copy of the repeated previous program counter value is saved along with n, a tremendous savings.

```
0    ...
1   label 1
2   push  I
3   push  J
4   eql
5   brt   2
6   push  I
7   push  1
8   add
9   pop   I
10  br    1
11  label 2
12  halt
```

Figure 9
Simple E-code Program with a Loop

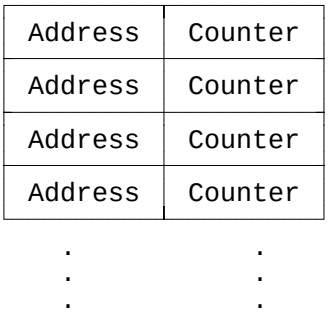


Figure 10
General Label Stack

Look again at Figure 9 and consider what will happen to the label stack for the branch point instruction at line 1 as the E-machine executes the instructions. Assume that I equals 3 and J equals 5. The E-machine will step sequentially through the instructions starting at 0. When the label 1 instruction at line 1 is executed for the first time, the address of the the instruction executed just prior to it (at this point, instruction 0) is pushed onto label 1’s label stack, resulting in the label stack of Figure 11.

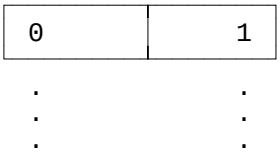


Figure 11
Label Stack After 0 Loop Iterations

As the E-machine continues executing, I and J will be compared, they will be found to be not equal and so the E-machine will continue executing sequentially, incrementing I in the process, until line 10 is reached. At that point, a branch to the label 1 instruction is executed, which loads 1 into the program counter. When the label 1 instruction is executed, the address of the instruction that was executed just prior to this is pushed onto the stack for label 1. Since that instruction was the branch instruction at line 10, a 10 must be pushed as shown in Figure 12.

At this point in the execution, the loop has executed once, I equals 4, J equals 5, and the E-machine has just executed line 1. Proceeding sequentially with the execution of the program results in I and J being compared. Once again, I does not equal J and the E-machine executes sequentially, again incrementing I, until line 10 is reached. At this point, the branch to label 1 is executed. The execution of label 1 causes the address of the instruction executed just prior to the label 1, 10, to be pushed onto label 1's stack. This results in the label stack of Figure 20. Notice that no new address was actually pushed onto the stack. Since the top of the stack had the same value as the value that was to be pushed, the counter of the top of the stack element was simply incremented (from 1 to 2). If the address had been different than the value of the top of the stack, a new value and counter would have been pushed onto the top of the stack.

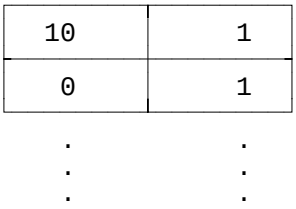


Figure 12
Label Stack After 1 Loop Iteration

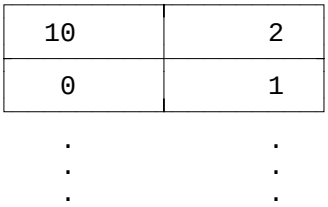


Figure 13
Label Stack After 2 Loop Iterations

How to reverse through a label instruction should now be clear. The address on top of that label's stack is simply placed in the program counter. If the corresponding count is one, the label stack is also popped, otherwise the count is just decremented.

Critical vs. Noncritical Instructions

Early on in the chapter, it was mentioned that a computer running in reverse and a high level language program executing in reverse represented similar but not identical processes. The reason this is so is that one high level language statement will, in general, correspond to many machine language instructions. For example, the Pascal assignment statement

```
Y := X + Y - 17 * Z * Z;
```

will be translated into at least ten machine language instructions, as shown in Figure 14, only one of which has any effect on the values of the variables in this statement (the final pop instruction). Since the intent of the proposed system is to display the execution dynamics of high level language programs, it is unnecessary to be concerned about precisely backing up the E-code instructions that only calculate intermediate

values. This observation led naturally to a classification system for E-machine instructions that reflects this situation. If an E-machine instruction destroys information necessary for backup in the high level language program, it is classified as critical by the compiler; if it does not, it is classified as noncritical.

This identification of E-code instructions as either critical or noncritical allows the E-machine to save for backup purposes only that information necessary to reverse statements in the high level language program. Since the vast majority of compiled E-code instructions will be noncritical, a large savings in storage space and time is realized. However, it should be noted that the flexibility is present to accurately back up E-machine code line by line by simply designating each instruction as critical.

```
push  X
push  Y
push  17
push  Z
mult
push  Z
mult
sub
add
pop   Y
```

Figure 14
E-code Translation of X := X + Y - 17 * Z * Z;

CHAPTER 3

THE DESIGN OF THE E-MACHINE EMULATOR

This chapter describes the design of the E-machine emulator, which essentially follows the design of the E-machine presented in Chapter 2 of this thesis. The emulator is intended eventually to be included as part of another program; the animator. The emulator is partitioned into several modules including the Fetch/Decode/Execute, Address Decode, Program Memory, Instruction Execution, Data Memory, Variable Register/Stack, Label Register/Stack, Evaluation Stack, Call Stack, Save Stack, Packet, Fault, Load, and Flags modules. The animator will also require access to some of these modules in order to produce animation. These include the Fetch/Decode/Execute, Data Memory, Variable Register/Stack, and Packet modules. In addition, there are a few data structures that will need to be accessed directly (i.e., without the use of interfacing functions) by the animator, namely the Symbol/Type Table and the Source Code Array. The emulator, whose source code is included in Appendix A, was written in ANSI Standard C for portability. It was compiled in Turbo C on an IBM PC compatible computer, on which it was also tested.

Figure 15 presents a graphical representation of the emulator design. It is intended to show which modules interact with others. The direction of information flow is also shown with data flow following the arrows. A line with an arrow at either end indicates flow in both directions. The Fault and Flags modules have been left out of the diagram because they are not particularly important to the design of the emulator and their inclusion would make the diagram difficult to read. A description of each module follows.

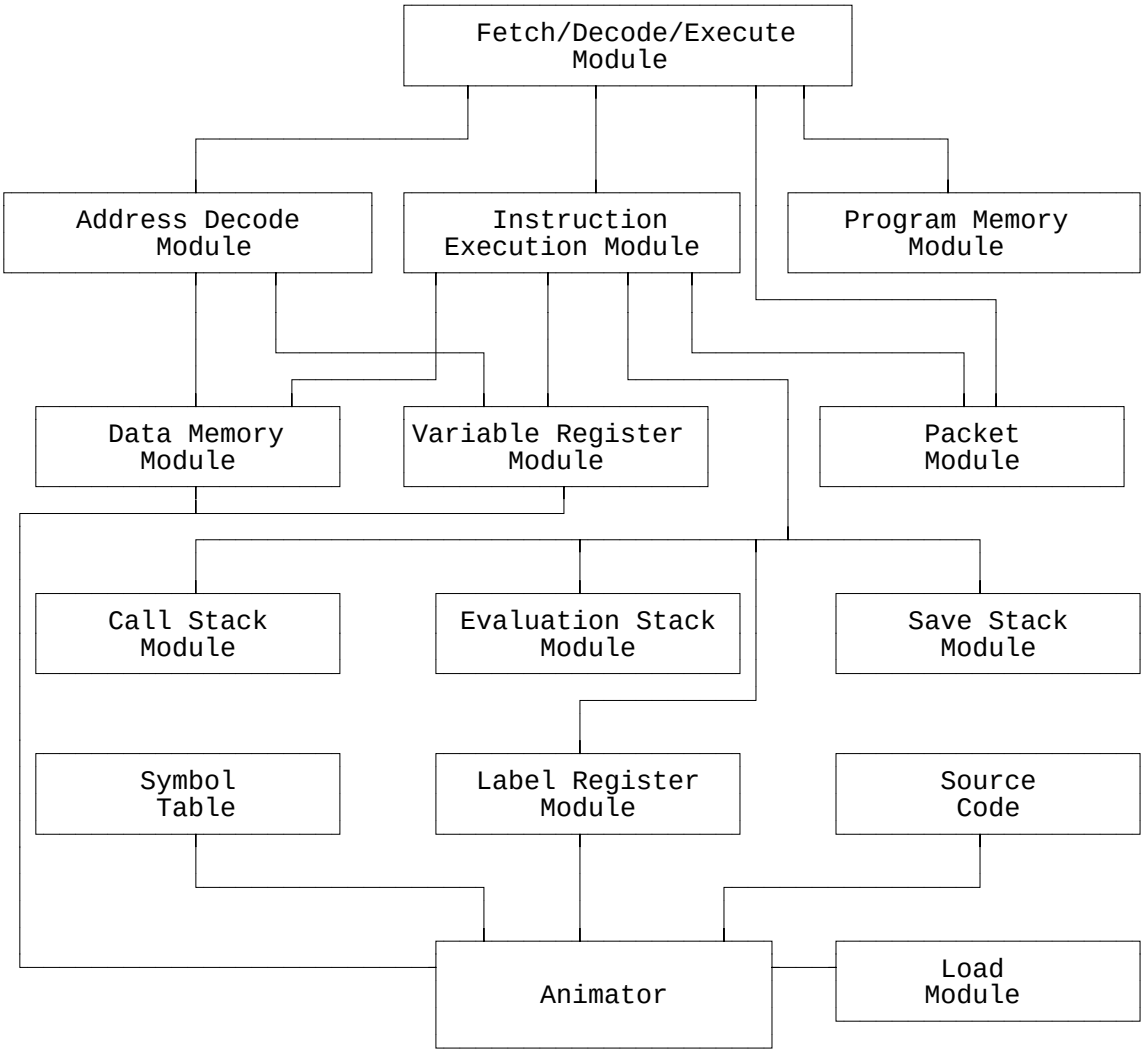


Figure 15
Graphical Representation of emulator

Fetch/Decode/Execute Module

The Fetch/Decode/Execute Module is the driver for the rest of the execution part of the E-machine. This module is comparable to the fetch and execute cycles of standard von Neumann computers. One difference from the von Neumann model is that a packet of instructions corresponding to a high level language statement is executed, then execution is suspended and control returned to the animator, so that information about the current

state of the machine can be managed. Decoding and execution are carried out in separate modules but are controlled from this module.

Data Structures, Defined Types, and Constants

opinfo
: This is a table that includes an entry for each machine opcode. Each table entry consists of a pointer to the function that provides the execution of the opcode, and the type of argument expected by the opcode (DATA, ADDR, or NONE).

currentpacket
: This structure contains all the information about the packet that is currently being executed, or when the animator has control, information about the next packet that is to be executed. For an explanation of the packet data structure, see the packet module description.

programcounter
: This is a variable that has the same function as the program counter in a Von-Neumann machine; it keeps track of which machine instruction should be executed next.

previouspc
: This variable holds the programcounter value that existed just prior to the current programcounter value.

Instruction
: This is the structure definition for an E-machine instruction. This structure holds the opcode, addressing mode, data type, critical flag, and operands for an E-machine instruction.

Functions

executepacket()
: This function carries out execution of all instructions in the current packet. When programcounter is no longer within the bounds of the current packet, execution halts until this function is called again.

setpc(address)
: This function is used to allow some of the instruction execution functions (e.g., the branch instructions) to change programcounter.

incpc()
: This function increments programcounter so that it points to the instruction immediately following the current instruction.

setppc(address)
: This function is used by some of the instruction execution functions (e.g., the branch instructions) to set the previouspc.

reverse()
: This function is intended to be called by the animator to change the direction of execution. If execution is currently forward, programcounter and previouspc are swapped and the FORWARD flag is set to FALSE. If execution is currently

reverse, programcounter and previouspc are swapped and the FORWARD flag is set to TRUE. The changing of programcounter is necessary since it must point to the next instruction to be executed.

getcurrpacket(packet)
This function is meant for use by the animator so that it may get the contents of the currentpacket variable which holds the beginning and ending line and column of the source code that is represented by the current packet.

Calls To

Address Decode Module, Execution Module, Packet Module, Flags Module

Calls From

animator

Address Decode Module

This module is responsible for decoding the addressing mode of an instruction to obtain the appropriate operands. A majority of the instructions can make use of the address mode decoder; however, there are some instructions that are handled by their respective instruction execution function.

Data Structures, Defined Types, and Constants

ArgType
This is an enumeration type with the values DATA, ADDR, and NONE, which represent the kind of information indicated by the addressing mode.

Functions

decaddr(instruction, address, data, defined)
This function decodes the addressing mode in instruction, provided an operand is expected, and returns the data memory address, if one exists, the data at that address, and whether the data is defined or undefined.

Calls To

Data Memory Model, Variable Register/Stack Module

Calls From

Fetch/Decode/Execute Module

Program Memory Module

This module is responsible for handling program memory. This includes loading program memory with object code and returning the contents of a program memory address. The program memory data structure is only accessible within this module, so this module must be called to get the contents of a program memory address.

Data Structures, Defined Types, and Constants

ProgAddress
: This is a type definition for program addresses. Just enough memory is allocated for progmem in this function to hold all of the E-code instructions for the object program.

progmem
: This is an array of type instruction, the indices of which are program addresses.

Functions

loadprogmem(file)
: This function loads E-code instructions into program memory from the object file.

getinstruction(address)
: This function returns the instruction located at the given program memory address.

Calls To

None

Calls From

Fetch/Decode/Execute Module

Instruction Execution Module

This module contains all of the functions that carry out the execution of E-machine instructions. The function pointers in the opinfo structure in the Fetch/Decode/Execute module point to functions in this module.

Data Structures, Defined Types, and Constants

Instruction
This structure defines the format of an E-machine instruction. Included in the structure are the opcode, addressing mode, data type, critical flag, and data.

Functions

There is one function for each of the E-machine instructions with the same name as the opcode mnemonics described in Chapter 2. The functions follow the explanation of the instructions given in Chapter 2 and so will not be described here. The parameters passed to these functions, however, do require some explanation.

There are three parameters that are passed to each function. Not all of these parameters are used by all of the functions, but are included to keep order in the calling of the functions. The three parameters are an instruction, a data item, and a data memory address. The instruction is a data structure of type Instruction described above and contains all the information about the instruction that is currently being executed. Information in the instruction that may be used in a function includes the data type, the critical/noncritical flag, and the operand. The data item is the data found at the address calculated by the address decoding module and the data memory address is the calculated address.

Some of the functions expect data, others addresses, and some nothing at all (i.e. the push instruction requires a data item, which will be pushed onto the evaluation stack, whereas the pop instruction requires an address at which to store data popped from the stack). This information is stored in the opinfo table in the Fetch/Decode/Execute module and is used by the address decoder to signal faults when the requested information represents an impossibility with the given addressing mode. The data and address are always computed for the given addressing mode and passed to each function. The reason the address of the data is passed when the operation only requires data is that the

address of the data may be necessary for saving backup information for critical instructions.

Calls To

Data Memory Module, Variable Register Module, Label Register Module, Flags Module, Evaluation Stack Module, Save Stack Module, Call Stack Module.

Calls From

Fetch/Decode/Execute Module.

Data Memory Module

This module is responsible for managing data memory. This includes the operations of allocating, deallocating, loading, and storing. This module is also responsible for maintaining the defined/undefined status of the data memory locations. All data memory is marked undefined when first allocated, then when a value is placed into a data memory location, that location is marked defined. Data Memory is maintained at the smallest addressable memory size for the computer on which the emulator is running.

Data Structures, Defined Types, and Constants

DataWord
: A type definition given a C data type that represents the smallest addressable piece of memory (e.g., if a machine is byte addressable, then DataWord would most likely be defined by the C type of unsigned char).

WORDSIZE
: A constant whose value is the number of bits in one DataWord.

DoubleWord
: A type definition given a C data type that represents two DataWords (e.g., using the same byte example from above, DoubleWord might be defined by the C type of unsigned int).

DBLWORDSIZE
: A constant whose value is 2 * WORDSIZE and should represent the number of bits in one DoubleWord.

DataAddress
: A type definition for a data memory address.

MAXDATA
: A constant whose value is the size of data memory, which is statically allocated during compilation of the emulator.

datamem
: An array of MAXDATA DataWords whose indices are data memory addresses. This is the data memory structure.

datadefined
: This is a bit array of MAXDATA size. Each bit is used to mark the corresponding data locations as being defined or undefined. A bit value of 1 means the data is defined and a bit value of 0 means the data is undefined.

freemem
: This is an array, each of whose elements is a structure containing two data addresses, a beginning and ending address. This structure is used to allocate and deallocate data memory. Each entry in the array marks a free block of memory. To begin with there is only one entry in the array with a beginning address of 0 and an ending address of MAXDATA -1.

lastentry
: This is an index into the freemem array that marks the last free block entry in the freemem array.

Functions

readdata(address, data, type, defined)
: This function is used to read a data value from data memory. Data of the specified type is read from the given data address and placed in data. The corresponding bit pattern is also retrieved from the datadefined array and placed right justified in defined. A fault is generated if the address is not a valid data memory address.

writedata(address, data, type)
: This function is used to place values into data memory. Data is placed into data memory at the given address and the corresponding bits in the datadefined array are set to indicate that the data is defined. A fault occurs if DataAddress is not a valid data memory address.

allocdata(address, size)
: This function searches the freemem array to find a block of memory at least size DataWords long. If one is found, the address of that block of data is returned, otherwise a null address is returned.

unallocdata(address, size)
: This function places the block of data starting at address, size DataWords long, into the freemem array so that it may be used again later.

savemem(address, size)
: This function pushes a block of data memory size units long beginning at the given data memory address onto the save stack. The defined/undefined status of each location is saved on the stack as well.

unsavemem(address, size)
: This function pops a block of data memory off the save stack and places it at the data memory address supplied. The defined/undefined status of the data is restored from the stack as well.

Calls To

Save Stack Module

Calls From

Instruction Execution Module, Address Module

Variable Register/Stack Module

This module is responsible for maintaining the variable registers and stacks. This includes pushing and popping addresses from the variable stacks, returning variable addresses, and reading the sizes of variables from the object file.

Data Structures, Defined Types, and Constants

VariableReg
: This is a type definition for the variable register numbers which are used to access the individual variable registers.

varregs
: This is an array of type VariableReg. This is the set of variable registers.

lastreg
: This is a variable of type VariableNum which holds the highest numbered variable register so that checks can be made to determine legal variable register numbers.

VarStack
: This is a type definition for the variable register stacks.

Functions

pushvariable(varreg, address)
: This function is used to create a new instance of a variable by pushing address onto the stack corresponding to varreg. A fault occurs if varreg is not a valid variable register number.

popvariable(varreg)
: This function is used to remove an instance of a variable by removing the top address on the stack corresponding to varreg. A fault occurs if varreg is not a valid variable register number.

getvaraddress(varreg, address)
: This function returns in address the top address on the stack corresponding to varreg. A fault occurs if varreg is not a valid variable register number.

getvarsize(varreg, size)
This function returns, in size, the size of the variable represented by varreg. The size will most likely be used to allocate memory for a new instance of the variable. A fault occurs if varreg is not a valid variable register number.

loadvarregs(file)
This function reads in a set of variable registers from an object file, whose form is described in Chapter 5.

Calls To

None

Calls From

Address Decode Module, Instruction Execution Module

Label Register/Stack Module

This module is responsible for maintaining the label registers and stacks. This includes pushing and popping addresses from the label stacks and returning the addresses of label instructions. This module is primarily used by the E-machine branch instructions to branch to the proper address during both forward and reverse execution.

Data Structures, Defined Types, and Constants

LabelReg
This is a type definition for the label number which is an index into the labelregs structure.

labelregs
This is an array of type LabelReg. This is the structure for the set of label registers.

lastreg
This is a variable of type LabelReg, which holds the last valid label register number. This is used to determine if a label register number is valid.

LabelStack
This is a type definition for the label stack structure.

Functions

getlabeladdress(label, address)
This function returns, in address, the program address of the given label register. A fault occurs if label is not a valid label register number.

pushlabel(label, address)
This function pushes address onto the given label register's stack. A fault occurs if label is not a valid label register number.

poplabel(label, address)
This function returns the program address on top of the stack associated with label. A fault occurs if label is not a valid label register number.

loadlabelregs(file)
This function reads in a set of label registers from an object file, whose form is described in Chapter 5.

Calls To

None

Calls From

Instruction Execution Module, Load Module

Evaluation Stack Module

This module maintains the evaluation stack. This includes pushing (and popping) data values onto (and from) the evaluation stack. Unlike the save stack, variables are not marked defined or undefined on the evaluation stack. Any errors due to undefined data are to be handled after reading a value from data memory.

Data Structures, Defined Types, and Constants

evalstack
This is a static array whose elements are of type DataValue (a union declared in the datamem module). This is the evaluation stack structure.

evaltop
This variable is an index into the evalstack structure and is used to mark the top of the evaluation stack.

Functions

pusheval(data)
This function pushes the given data onto the evaluation stack.

popeval(data)
This function pops the top value from the evaluation stack and returns it in data.

Calls To

None

Calls From

Instruction Execution Module

Call Stack Module

This module maintains the emulator's call stack. This includes the pushing and popping of program addresses to and from the call stack. The call stack is used to store the next instruction to be executed upon return from a called procedure.

Data Structures, Defined Types, and Constants

callstack
: This is an array whose elements are of type ProgAddress. This is the call stack structure.

calltop
: This variable is an index into the callstack structure and is used to mark the top of the stack.

Functions

pushcall(address)
: This function pushes address onto the call stack.

popcall(address)
: This function pops the top program address off the call stack and returns it in address.

Calls To

None

Calls From

Instruction Execution Module

Save Stack Module

This module maintains the E-machine save stack. This includes pushing and popping data and data memory. There are two routines for the push and pop, one for data values, the other for a block of data memory. This was done to make pushing data values easier to deal with. Note that the save stack also holds the defined/undefined status of the data.

Data Structures, Defined Types, and Constants

MAXSAVE
: This is a defined constant which holds the size of the save stack.

savestack
: This is an array with MAXSAVE elements of type DataWord. This is the E-machine save stack structure.

savetop
: This is an integer index into the savestack structure that marks the top of the save stack.

savebottom
: This is an integer index into the savestack structure that marks the bottom of the save stack. This is used because the save stack wraps around and thus cannot overflow. This means that backup is always possible, but not necessarily all the way back to the beginning of the program.

savedefined
: This is a bit array associated with the save stack structure and is used to hold the defined/undefined status of the elements stored in the save stack.

packetqueue
: This queue is used to store the position in the save stack of the beginning of each packet that is stored there. Because correct backup is only guaranteed across packets, this structure is necessary to make sure the bottom of the save stack is always at a packet boundary (if this condition weren't monitored, it would be possible in the circular stack for partial packets to occur as the stack wrapped around, possibly leading to problems during extended reverse execution).

packettop
: This is an integer index into the packetqueue that marks the top of the queue where save stack indices are removed.

packetbottom
: This is an integer an index into the packetqueue that marks the bottom of the queue where save stack indices are added.

Functions

newpacket()
This function queues the current top of the save stack in packetqueue marking the beginning of reversal information for a packet on the save stack.

savedata(data, type, defined)
This function pushes data, which has the E-machine data type type, onto the save stack. Defined is a bit string of the defined/undefined status of data.

unsavedata(data, type, defined)
This function pops the top value of E-machine data type type, off the save stack and returns it in data. The defined/undefined status of the data is returned in defined.

Calls To

None

Calls From

Fetch/Decode/Execute Module, Instruction Execution Module

Packet Module

The packet module is responsible for maintaining the packet table. This involves loading the packet table from the object file and returning packets from the table given either a program address or a packet number, which is an index into the packet table.

Data Structures, Defined Types, and Constants

Packet
This is a type definition for a packet. This structure contains seven fields for the beginning and ending E-machine code program address for the packet, the beginning and ending line and column numbers of the source code that is represented by the packet, and an index into the symbol table to mark the variable scope for the packet.

PacketNum
This is a type definition for packet numbers. The packet number is an index into the packettable structure.

packettable
: This is an array of type Packet and is allocated when the object file is read into the emulator.

lastpacket
: This is an integer that contains the largest available packet number for the loaded object program.

Functions

findpacket(address, packet, packetnum)
: This function searches for the packet that contains address between the beginning and ending program addresses for the packet. If found, the packet and packet number are returned in packet and packetnum, respectively, otherwise, a packet that contains a null program pointer for the beginning address is returned.

getpacket(packetnum, packet)
: This function returns the packetnum element of the packet table. If packetnum is not a legitimate packet number then the packet returned will have a null program address for the beginning program address.

loadpackettable(file)
: This function reads a packet table from the object file, whose form is described in Chapter 5.

Calls To

None

Calls From

Instruction Execution Module, Load Module

Fault Module

The fault module reports E-machine errors that may occur during execution of a program, such as divide by zero, or illegal opcode. Each different fault that can occur is given a value from an enumerated type. Execution does not necessarily halt when a fault occurs; the fault handler keeps track of the last fault that occurred and execution continues. A query can be done of the last fault and the emulator makes decisions as to how to execute instructions with the existence of the fault. The fault module would most likely be used by the animator to notify the user of an error in execution.

Data Structure, Defined Types, and Constants

FaultType
This is an enumerated type that contains all of the different faults that can occur throughout the emulator. Refer to the source code in Appendix A for the names of the enumerated constants.

faultmsgs
This is an array of strings, containing explanations for each fault. This is used by the faultmsg function to give access to the descriptions of the faults.

lastfault
This variable holds the last registered fault that was declared.

Functions

fault(faultnum)
This is the function that is called throughout the emulator when an error occurs. Faultnum is the fault number of the that occurred.

getfault()
This function returns the current value of lastfault.

faultmsg(fault, msg)
This function returns a message in msg describing the particular fault that was passed in.

Calls To

None

Calls From

Animator, Fetch/Decode/Execute Module, Address Decoding Module, Program Memory Module, Instruction Execution Module, Data Memory Module, Variable Register/Stack Module, Label Register/Stack Module, Evaluation Stack Module, Call Stack Module, Save Stack Module, Packet Module

Load Module

The load module is used to load an object file into the emulator for execution. This calls a host of load functions in the various other modules in order to accomplish its task. The load module merely reads in a header and determines which of the load functions to call. For a description of the format of the object file see Chapter 5.

Data Structure, Defined Types, and Constants

Sections
This is an enumerated type for the headings of the various sections of the object file.

Functions

loadobjectfile (filename)
This is the only function in this module and it is responsible for reading the object file filename and determining the different sections of the object file and which load functions need to be called. This module also stores the symbol table and source code in their respective data structures as they are read in from the object file.

Calls To

Program Memory Module, Variable Register/Stack Module, Label Register/Stack Module, Packet Module

Calls From

Animator

Flags Module

The flags module maintains a set of flags used in most of the modules in the emulator. The flags are used for a variety of things, such as to specify the direction of program execution. These flags are not meant to be used by the animator, so a description of each of the flags has been left out.

Data Structure, Defined Types, and Constants

flags
This is an array of type BooleanType that holds the values for each of the flags, each of which is represented by their index into the array.

Functions

getflag(flag)
This BooleanType function returns, as the result of the function, the value of the flag passed in flag.

setflag(flag)
This function sets the flag corresponding to flag to the value TRUE.

resetflag(flag)
This function sets the flag corresponding to flag to the value FALSE.

Calls To

None

Calls From

Instruction Execution Module, Fetch/Decode/Execute Module, Data Memory Module, Program Memory Module, Fault Module

Symbol Table

The symbol table is just a data structure and does not have a module to manipulate it. This is because the symbol table contains information that must be available to the animator in order to allow extracting information about program identifiers in an efficient manner. The symbol table not only contains the variable and procedure names that are part of the source program, but also the structure of the variables in the program. The structure of the variables is necessary so that the animator can display variables appropriately.

Because the symbol table conveys a great deal of information about the original source program, each entry is rather large and a complex tree structure is maintained within the table for managing variable scope. Following is a description of each field of a symbol table entry. The fields are: name, size, type, variablereg, offset, lowerbound, upperbound, parent, child, and last.

Name, size, type, and variablereg are easily explained. Name is a pointer to a character array that contains the name of the symbol for the entry. The size field is an integer whose value is the size of the element in the entry. Size represents the size of only a single element if the entry is an array. The type field holds an E-machine data type and has a value only if the entry is a scalar E-machine data type variable or an array whose elements are of a basic E-machine data type. The

variablereg field contains a variable register number if the entry is a source program variable that is represented by a variable register.

Before any further descriptions of the remaining fields an explanation of scoping is necessary. In order to accommodate scopes in the symbol table the idea of scope blocks is incorporated. A scope block is a linear group of symbol table entries that are considered to be part of the same local scope; the last entry in the scope is marked by a blank entry with a value of END in the type field. However, the symbols in the scope of a routine typically include those symbols which were in the scope of the calling procedure or function, excluding identifiers with the same name (and so on, back until the outermost scope is reached). To incorporate this aspect of scoping into the symbol table the parent field was included with each local scope. In addition, the child field was included for each local scope to allow moving deeper into the scope levels. Complete explanations of those fields which are affected by the scope blocks in the symbol table follow.

The offset field is an integer whose value is the offset from some base (from a parent or other ancestor), where the value for the entry is stored. The lowerbound and upperbound fields are the lower and upper bounds, respectively, for array variables. These fields are both zero for an entry that is not an array. The parent field has two purposes. All fields in the first entry in each scope block are blank except for the parent field, which points to the starting symbol table entry of the scope block which immediately contains the current scope block. For example, a record is stored in the symbol table with one entry for each field of the record; for a particular field of a record, the parent entry points to the parent record definition so that the base address and offset of this field in main memory can be calculated.

The child field is used to point to another scope block containing additional information about the current entry. For example, the child field of a symbol table entry for a record variable might point to a child

scope block containing entries for each of the fields in the record. Another use of child is for multidimensioned arrays. Since there is only one set of upper and lower bounds for an array entry, the child field is used to point to another symbol table entry containing the next set of bounds for another dimension of the multidimensional array. Thus, arrays can have an unlimited number of dimensions.

The symbol table is somewhat complex and can best be understood if read in conjunction with the example symbol table presented in the next chapter.

Source Code Array

This is a data structure defined with the emulator, but is only of use by the animator for the display of the original source code. The packet table module makes reference to this array with the startline, startcol, endline, and endcol fields so the source code array was included in the emulator design for clarity. This structure is very simple and requires little explanation. The Source Code Array is an array of character pointers. The array is indexed from 0 to the number of lines of source code and each entry points to one line of source code. Memory space is allocated for this structure during the object file load process.

CHAPTER 4

E-MACHINE EMULATOR OPERATION

This chapter is intended to present an example of the emulator at work. The example provided should give insight into the operations of the emulator, as well as how the various data structures in the E-machine are created and handled (i.e., the save stack, the symbol table, etc.). The example program used manipulates a binary tree. This chapter should be of particular interest to the developer of the animator. This chapter also contains useful information for the compiler writer.

The Pascal source code for the program is given with line numbers as required by the emulator. An E-code translation of the program is provided along with a symbol table, variable register table, label register table, and all other structures which must be contained in the object file generated by a Pascal compiler for the E-machine. The actions of the emulator are presented in conjunction with a simple display environment as a series of figures that illustrate the use of the E-machine for program animation.

```
 0   [Program BinaryTree( Output );]
 1
 2   [Type]
 3     [NodePtr = ^Node;]
 4     [Node = Record
 5       Data  : Integer;
 6       Left  : NodePtr;
 7       Right : NodePtr;
 8     End;]
 9
10   [Var]
11     [Root : NodePtr;]
12
13   [Procedure Insert]([Var Root : NodePtr;] [Data : Integer]);
14
15   [Var]
16     [NewNode,] [CurNode] : NodePtr;
```

Figure 16
Example Packetized Pascal Program

```
17
18   [Begin]
19     [New(NewNode);]
20     [NewNode^.Data := Data;]
21     [NewNode^.Left := Nil;]
22     [NewNode^.Right := Nil;]
23     [If (Root = Nil) Then]
24       [Root := NewNode]
25     [Else]
26       [Begin]
27         [CurNode := Root;]
28         [While (CurNode <> Nil) Do]
29           [If (Data < CurNode^.Data) Then]
30             [If (CurNode^.Left = Nil) Then]
31               [Begin]
32                 [CurNode^.Left := NewNode;]
33                 [CurNode := Nil]
34               [End]
35             [Else]
36               [CurNode := CurNode^.Left]
37           [Else]
38             [If (CurNode^.Right = Nil) Then]
39               [Begin]
40                 [CurNode^.Right := NewNode;]
41                 [CurNode := Nil;]
42               [End]
43             [Else]
44               [CurNode := CurNode^.Right;]
45       [End;]
46   [End;]
47
48
49   [Procedure Traverse]([Root : NodePtr]);
50
51   [Begin]
52     [If (Root <> Nil) Then]
53       [Begin]
54         [Traverse(Root^.Left);]
55         [Write(Root^.Data);]
56         [Traverse(Root^.Right);]
57       [End;]
58   [End;]
59
60   [Begin]
61     [Root := Nil;]
62     [Insert(Root, 5);]
63     [Insert(Root, 4);]
64     [Insert(Root, 6);]
65     [Traverse(Root);]
66   [End.]
```

Figure 16 (continued)

Figure 16 represents a binary tree Pascal program that will be used as an example throughout this chapter. The program is shown packetized (with portions of code grouped together into packets) and with line numbers. The packets are the pieces contained within the brackets ([]).

The line numbers begin with 0, as this is what the emulator expects. Figure 17 shows the same Pascal program interspersed with an E-code translation of each packet and packet numbering. The numbers that appear before each line of E-code are the program addresses used by the emulator. The translation given is meant to represent what might be generated by a compiler for this particular Pascal program, although there is no intention to indicate that this is what should be generated by the compiler. Notice the abundance of nop (no operation) instructions. They are necessary to create an E-code packet for Pascal packets that actually have no translation into E-code, but which the animator might wish to highlight.

```
 0   [Program BinaryTree( Output );]
     Packet #00:
       0   nop
 1
 2   [Type]
     Packet #01:
       1   nop
 3     [NodePtr = ^Node;]
     Packet #02:
       2   nop
 4     [Node = Record
 5       Data  : Integer;
 6       Left  : NodePtr;
 7       Right : NodePtr;
 8     End;]
     Packet #03:
       3   nop
 9
10   [Var]
     Packet #04:
       4   nop
11     [Root : NodePtr;]
     Packet #05:
       5   inst        n,V0        ; create an instance of Root
       6   br          c,0         ; branch to beginning of main program
12
13   [Procedure Insert]([Var Root : NodePtr;] [Data : Integer]);
     Packet #06:
       7   label       c,1         ; Insert entry point
     Packet #07:
       8   link        V1          ; link Root to actual param
     Packet #08:
       9   inst        c,V2        ; create instance of Data
      10   pop         c,I,V2      ; put actual param into Data
14
```

Figure 17
An E-code Translation for an Example Program

```
15   [Var]
     Packet #09:
      10   nop
16     [NewNode,] [CurNode] : NodePtr;
     Packet #10:
      11   inst        c,V3        ; create instance of NewNode
     Packet #11:
      12   inst        c,V4        ; create instance of CurNode
17
18   [Begin]
     Packet #12:
      13   nop
19     [New(NewNode);]
     Packet #13:
      14   alloc       c,C8        ; allocate block for a new node
      15   pop         c,A,V3      ; put address of block in NewNode
20     [NewNode^.Data := Data;]
     Packet #14:
      16   push        I,V2        ; push Data onto stack
      17   push        A,V3        ; push address in NewNode onto stack
      18   popar       c           ; put address into address register
      19   loadir      c,0         ; load index reg with offset of .Data
      20   pop         c,A,AR{IR}  ; pop Data into NewNode^.Data
21     [NewNode^.Left := Nil;]
     Packet #15:
      21   push        A,NIL       ; push null address onto stack
      22   push        A,V3        ; push address in NewNode onto stack
      23   popar       c           ; put address into address register
      24   loadir      c,4         ; load index reg with offset of Left
      25   pop         c,A,AR{IR}  ; pop null address into NewNode^.Left
22     [NewNode^.Right := Nil;]
     Packet #16:
      26   push        A,NIL       ; push null address onto stack
      27   push        A,V3        ; push address in NewNode onto stack
      28   popar       c           ; put address into address register
      29   loadir      c,6         ; load index reg with offset of Right
      30   pop         c,A,AR{IR}  ; pop null address into NewNode^.Right
23     [If (Root := Nil) Then]
     Packet #17:
      31   push        A,V1        ; push Root onto stack
      32   push        A,NIL       ; push null address onto stack
      33   eql         n,A         ; Root = Nil?
      34   brf         c,2         ; if not, goto else
24        [Root := NewNode]
     Packet #18:
      35   push        A,V3        ; push NewNode onto stack
      36   pop         c,A,V1      ; assign to Root
      37   br          c,3         ; skip else
25     [Else]
     Packet #19:
      38   label       c,2         ; else clause
26        [Begin]
     Packet #20:
      39   nop
27           [CurNode := Root;]
     Packet #21:
      40   push        A,V1        ; push Root onto stack
      41   pop         c,A,V4      ; assign to CurNode
```

Figure 17 (continued)

```
28          [While (CurNode <> Nil) Do]
     Packet #22:
      42   label       c,4          ; top of while loop
      43   push        A,V4         ; push CurNode onto stack
      44   push        A,NIL        ; push null address onto stack
      45   neql        n,A          ; CurNode <> Nil?
      46   brf         c,5          ; if not, leave loop
29           [If (Data < CurNode^.Data) Then]
     Packet #23:
      47   push        I,V2         ; push Data onto stack
      48   push        A,V4         ; push CurNode onto stack
      49   popar       c            ; place address in address reg
      50   loadir      c,0          ; load index reg with offset of Data
      51   push        I,AR{IR}     ; push CurNode^.Data onto stack
      52   less        n,A          ; Data < CurNode^.Data?
      53   brf         c,6          ; if not, goto else
30             [If (CurNode^.Left = Nil) Then]
     Packet #24:
      54   push        A,V4         ; push CurNode onto stack
      55   popar       c            ; place address into address reg
      56   loadir      c,4          ; load index reg with offset of Left
      57   push        A,AR{IR}     ; push CurNode^.Left
      58   push        A,NIL        ; push null address
      59   eql         c,A          ; CurNode^.Left = Nil?
      60   brf         c,7          ; if not, goto else
31               [Begin]
     Packet #25:
      61   nop
32                 [CurNode^.Left := NewNode;]
     Packet #26:
      62   push        A,V3         ; push NewNode onto stack
      63   push        A,V4         ; push CurNode onto stack
      64   popar       c            ; put address into address reg
      65   loadir      c,4          ; load index reg with offset of Left
      66   pop         c,A,AR{IR}   ; assign NewNode to CurNode^.Left
33                 [CurNode := Nil]
     Packet #27:
      67   push        A,NIL        ; push null address onto stack
      68   pop         c,A,V4       ; assign to CurNode
34               [End]
     Packet #28:
      69   br          c,8          ; jump past else
35             [Else]
      70   label       c,7
36               [CurNode := CurNode^.Left]
     Packet #29:
      71   push        A,V4         ; push CurNode onto stack
      72   popar       c            ; put address into address reg
      73   loadir      c,4          ; load index reg with offset of Left
      74   push        c,A,AR{IR}   ; push CurNode^.Left
      75   pop         A,V4         ; assign to CurNode
      76   br          c,9          ; skip else
37           [Else]
     Packet #30:
      77   label       c,6
```

Figure 17 (continued)

```
38            [If (CurNode^.Right = Nil) Then]
    Packet #31
     78   push        A,V4        ; push CurNode onto stack
     79   popar       c           ; place address into address reg
     80   loadir      c,6         ; load index reg with offset of Right
     81   push        A,AR{IR}    ; push CurNode^.Right
     82   push        A,NIL       ; push null address
     83   eql         c,A         ; CurNode^.Right = Nil?
     84   brf         c,10        ; if not, goto else
39              [Begin]
    Packet #32:
     85   nop
40                [CurNode^.Right = NewNode;]
    Packet #33:
     86   push        A,V3        ; push NewNode onto stack
     87   push        A,V4        ; push CurNode onto stack
     88   popar       c           ; put address into address reg
     89   loadir      c,4         ; load index reg with offset of Right
     90   pop         c,A,AR{IR}  ; assign NewNode to CurNode^.Right
41                [CurNode := Nil;]
    Packet #34:
     91   push        A,NIL       ; push null address onto stack
     92   pop         c,A,V4      ; assign to CurNode
42              [End]
    Packet #35:
     93   br          c,11        ; skip else
43            [Else]
    Packet #36:
     94   label       c,10
44              [CurNode := CurNode^.Right;]
    Packet #37:
     95   push        A,V4        ; push CurNode onto stack
     96   popar       c           ; put address into address reg
     97   loadir      c,4         ; load index reg with offset of Left
     98   push        c,A,AR{IR}  ; push CurNode^.Left
     99   pop         A,V3        ; assign to CurNode
45         [End;]
    Packet #38:
    100   label       c,3
    101   label       c,8
    102   label       c,9
    103   label       c,11
    104   label       c,5
46  [End;]
    Packet #39:
    105   label       c,2
    106   uninst      c,V4        ; delete instance of CurNode
    107   uninst      c,V3        ; delete instance of NewNode
    108   uninst      c,V2        ; delete instance of Data
    109   unlink      c,V1        ; remove link to Root
    110   return
47
48
49  [Procedure Traverse]([Root : NodePtr]);
    Packet #40:
    111   label       c,12        ; Traverse entry point
    Packet #41:
    112   inst        c,V5        ; create instance of Root
    113   pop         c,A,V5      ; put actual param in Root
50
```

Figure 17 (continued)

```
51   [Begin]
     Packet #42:
     114   nop
52     [If (Root <> Nil) Then]
     Packet #43:
     115   push        A,V5        ; push Root onto stack
     116   push        A,NIL       ; push null address onto stack
     117   neql        c,A         ; Root <> Nil?
     118   brf         c,13        ; if not, skip to end of if
53       [Begin]
     Packet #44:
     119   nop
54         [Traverse(Root^.Left);]
     Packet #45:
     120   push        A,V5        ; push Root onto stack
     121   popar       c           ; put address into address reg
     122   loadir      c,4         ; put index of Left into index reg
     123   push        A,AR{IR}    ; push Root^.Left onto stack
     124   call        c,12        ; call Traverse
     125   label       c,14        ; records where returned from
55         [Write(Root^.Data);]
     Packet #46:
     126   push        A,V5        ; push Root onto stack
     127   popar       c           ; put address into address reg
     128   loadir      c,0         ; put offset of Data into index reg
     129   push        I,AR{IR}    ; push Root^.Data onto stack
     130   write       c,I         ; write it out
56         [Traverse(Root^.Right);]
     Packet #47:
     131   push        A,V5        ; push Root onto stack
     132   popar       c           ; put address into address reg
     133   loadir      c,4         ; put index of Right into index reg
     134   push        A,AR{IR}    ; push Root^.Right onto stack
     135   call        c,12        ; call Traverse
     136   label       c,15        ; records where returned from
57       [End;]
     Packet #48:
     137   label       c,13
58   [End;]
     Packet #49:
     138   uninst      v,V5        ; delete instance of Root
     139   return
59
60   [Begin]
     Packet #50:
     140   label       c,0         ; entry point of main program
61     [Root := Nil;]
     Packet #51:
     141   push        A,NIL       ; push null address onto stack
     142   pop         c,A,V0      ; assign to Root
62     [Insert(Root, 5);]
     Packet #52:
     143   push        I,C5        ; push 5 onto stack
     144   pusha       V0          ; push address of Root onto stack
     145   call        c,1         ; call Insert
     146   label       c,16        ; records where returned from
```

Figure 17 (continued)

```
63     [Insert(Root, 4);]
       Packet #53:
       147   push        I,C4        ; push 5 onto stack
       148   pusha       V0          ; push address of Root onto stack
       149   call        c,1         ; call Insert
       150   label       c,17        ; records where returned from
64     [Insert(Root, 6);]
       Packet #54:
       152   push        I,C6        ; push 5 onto stack
       153   pusha       V0          ; push address of Root onto stack
       154   call        c,1         ; call Insert
       155   label       c,18        ; records where returned from
65     [Traverse(Root);]
       Packet #55:
       156   push        A,V0        ; push Root onto stack
       157   call        c,12        ; call Traverse
       158   label       c,19        ; records where returned from
66     [End.]
       Packet #56:
       159   uninst      c,V0
```

Figure 17 (continued)

Entry Number	Symbol Name	Type	Offset	Variable reg	Parent	Child
0		Header			--	--
1	BinaryTree	Procedure			--	3
2		End			--	--
3		Header			0	--
4	Root	Address		V0	--	17
5	Insert	Procedure			--	8
6	Traverse	Procedure			--	14
7		End			0	--
8		Header			3	--
9	Root	Address		V1	--	17
10	Data	Integer		V2	--	--
11	NewNode	Address		V3	--	17
12	CurNode	Address		V4	--	17
13		End			3	--
14		Header			6	--
15	Root	Address		V5	--	17
16		End			6	--
17		Record			--	--
18	Data	Integer	0		--	--
19	Left	Address	4		--	17
20	Right	Address	6		--	17
21		End			--	--

Figure 18
Symbol Table for Example Program

Pack Num	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope
00	0	0	0	0	0	28	0
01	1	1	2	0	2	3	0
02	2	2	3	2	3	17	0
03	3	3	4	2	8	5	0
04	4	4	10	0	10	2	0
05	5	6	11	2	11	16	1
06	7	7	13	0	13	15	1
07	8	8	13	17	13	35	3
08	9	10	13	37	13	50	3
.							
.							
.							
54	152	155	64	2	64	17	0
55	156	158	65	2	65	16	0
56	159	159	66	9	66	3	0

Figure 19
Packet Table for Example Program

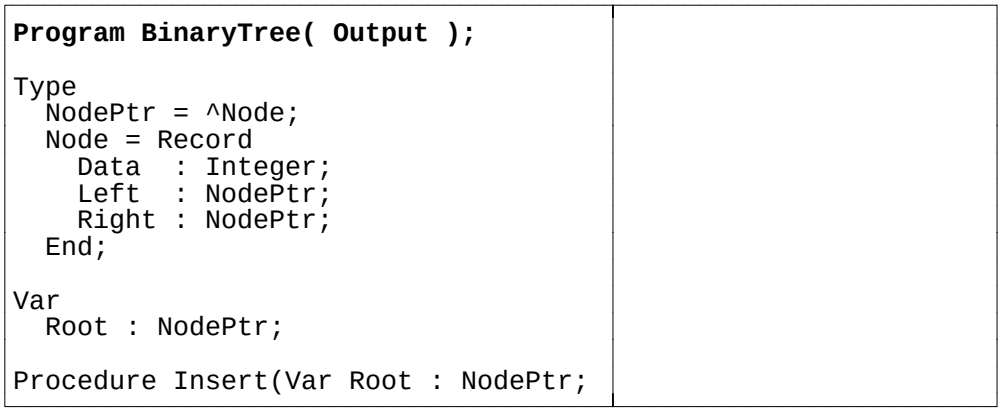
Variable Register	Variable Size
V0	2
V1	2
V2	4
V3	2
V4	2
V5	2

Figure 20
Variable Register Table for Example Program

Label Register	Label Address
0	141
1	6
2	39
.	
.	
.	
17	151
18	155
19	158

Figure 21
Label Register Table for Example Program

The next set of figures is intended to provide a rudimentary illustration of how a program animator might work in conjunction with the emulator. Each of these figures, beginning with Figure 22 depicts a computer video screen with a snapshot of the animation in progress.



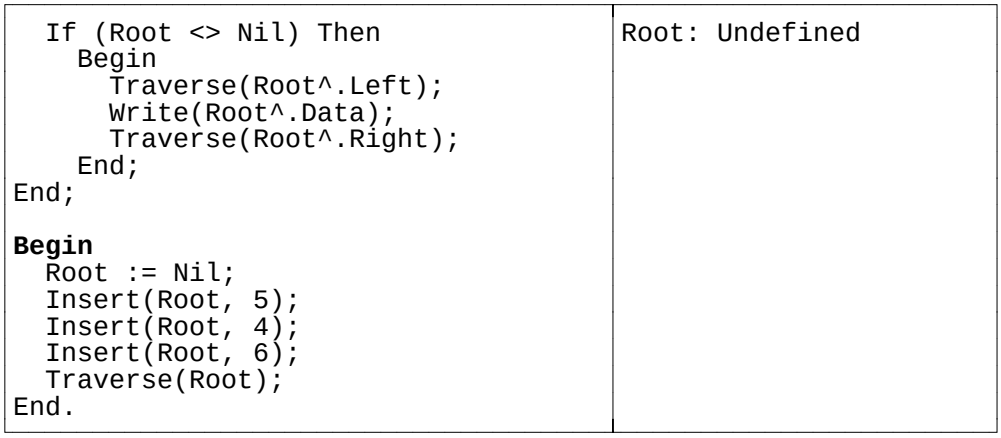
```
Program BinaryTree( Output );

Type
  NodePtr = ^Node;
  Node = Record
    Data  : Integer;
    Left  : NodePtr;
    Right : NodePtr;
  End;

Var
  Root : NodePtr;

Procedure Insert(Var Root : NodePtr;
```

Figure 22
Display Before Execution

Figure 22 shows what the display might look like before any instructions had been executed. All of the structures in the emulator--the evaluation stack, program memory, data memory, and so on, would have been initialized and the variable currentpacket would contain packet number 0, the next packet to be executed. From the packet table shown in Figure 19, the beginning and ending lines and columns in the source code corresponding to E-code packet 0, are line 0, column 0, line 0, column 28, which are shown highlighted in the display in boldface. After the emulator was called to execute a packet, the display would remain the same, except that instead of the first line being highlighted, the word Type on the third line would be highlighted. This is because packet 0 contains only a nop instruction, which would result in no changes in the state of the E-machine except to move on to the next packet. In fact, the highlighted text in the display would not change until after execution of packets 0 through 4 since they each consist only of a nop.

The twelfth line of the display would be highlighted after packet 4 had been executed. In this case the highlighted text would be the variable declaration of Root, and the instructions in packet 5 would create an instance for variable register 0 and then branch to the beginning of the main program. Variable register 0 represents the pointer Root in the original source program. According to the variable register

table in Figure 20, the size of the variable represented by variable register 0 is 2. (This is the number of data memory words required to store a data memory address for this particular version of the emulator.) Execution of this packet would cause two data words to be allocated and the top of variable register 0's stack to contain a 0, which is the data memory address at which the 2 data words would have been allocated. The two data words at data memory address 0 would have been marked as undefined since no value would yet have been placed in that memory location. Figure 23 shows what the display might look like after execution of packet 5 was completed.



```
  If (Root <> Nil) Then
    Begin
      Traverse(Root^.Left);
      Write(Root^.Data);
      Traverse(Root^.Right);
    End;
End;

Begin
  Root := Nil;
  Insert(Root, 5);
  Insert(Root, 4);
  Insert(Root, 6);
  Traverse(Root);
End.
```

```
Root: Undefined
```

Figure 23
Display After Executing Packet 5

Note that the program lines that were displayed in the previous figure are no longer in the display in Figure 23. This is because the last packet that was executed would have caused a branch to the beginning of the main program, which is no longer in the immediate vicinity of the source lines corresponding to the previous E-code packet. The animator would have to keep track of the source lines being displayed and when the source lines corresponding to the current packet are not part of the current display, change the display appropriately.

Figure 23 shows a possible display as packet 50 is about to be executed. Once again, packet 50 is another one that only contains a nop

instruction, so the display would likely not be updated after its execution. Packet 51 would then be the next packet to be executed, in which the value Nil would be assigned to the variable Root. The instructions in packet 51 would push a constant called NIL onto the evaluation stack and then pop this value off into variable register 0. At the time of the pop, there should be an address on top of variable register 0's stack (which, in this case, would be 0). So the value Nil would be placed in data memory at location 0, and since the value of Nil is an address occupying two data words, both location 0 and location 1 in data memory would contain a value representing the constant NIL. Figure 24 shows what the display might look like after execution of packet 51.

```
  If (Root <> Nil) Then
    Begin
      Traverse(Root^.Left);
      Write(Root^.Data);
      Traverse(Root^.Right);
    End;
End;

Begin
  Root := Nil;
  Insert(Root, 5);
  Insert(Root, 4);
  Insert(Root, 6);
  Traverse(Root);
End.
```

```
Root: Nil
```

Figure 24
Display After Executing Packet 51

Note in Figure 24 that Root is no longer displayed as undefined, but instead has a value of Nil. The variable Root should contain an E-machine address, but Nil is not an address, so how can Root have a value of Nil? Nil is actually a special value that does not represent a data memory location and is used to denote that an address variable (or pointer in Pascal) does not point to anything in particular. This is important, because, if Root did point to something, the data to which it pointed might be of interest, and indeed would be desirable to display. In fact, it is likely that the animator would display the value(s) to which Root

pointed if Root did point to something. An example of what the animator might do for such a circumstance is forthcoming.

E-code Packet 51 would now be the next packet to be executed. This packet is a procedure call. Execution of packet 51 would cause the emulator to push the address of the instruction following the call, specifically program address 140, onto the call stack. This is so that when procedure Insert was finished, it could perform a return properly. Next, the emulator would need to branch to label 1. This would be done by looking up the address of label 1 in the label register table, which is given in Figure 21. From this table the emulator would determine that label 1 was at program address 7, so the program counter would be loaded with this address.

Now look at packet 6. Executing packet 6 would likely not change the display at all, but it has a particularly important effect on the emulator. Packet 6 contains a label instruction, whose sole purpose is to mark the point to which a branch may have occurred. If the animator were to back up past the entrance to procedure insert, the emulator would need to know where the call came from initially. To accommodate this, the emulator would save the contents of the previous program counter, which would contain the address of the call instruction (program address 139), on top of label 1's stack. Execution would now move on to packet 7.

Packet 7 and packet 8 are responsible for setting up the parameters for the procedure. The parameter Root is passed by reference, so packet 7 contains a link instruction, which would pop the address from the top of the evaluation stack and push it onto variable register 1's stack. So now variable register 1 would have an address of 0, the same as the variable Root from the main program (the importance of this is shown shortly). The parameter Data is passed by value, so packet 8 would create an instance for variable register 2 and pop the integer (actual parameter value 5) off the top of the evaluation stack and place it into the data memory location that was allocated for variable register 2.

Packets 10 and 11 would create instances for the variables NewNode and CurNode in variable registers 3 and 4 respectively. These variables along with the procedure's parameters would now all be available for use in procedure Insert. Figure 25 shows what the animator might display to reflect this.

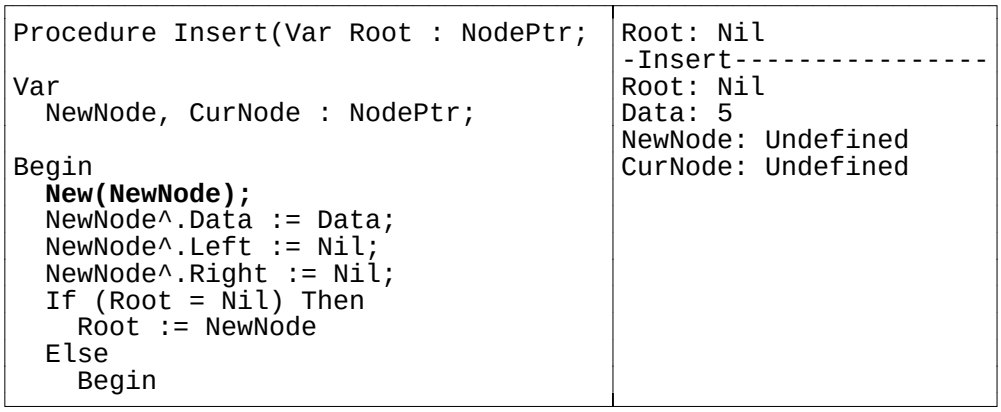


Figure 25
Display After Executing Packet 11

At this point, the user might be curious about why Data has a value of 5. To find out, the user could decide that he would like to back up until he finds the reason Data has value 5. The animator would call the reverse function of the emulator to place the emulator in reverse execution mode. The reverse function swaps programcounter and previouspc, sets the emulator's FORWARD flag to FALSE, and adjusts the current packet. The program counter would contain program address 13, which would make the current packet 11. Now when the execute packet function was called, packet 11 would be executed in reverse. Specifically, the address on top of variable 4's stack would be removed, and the data memory allocated for variable 4 would be released. The same would happen for variable register 3 when packet 10 was executed. For packet 8, the contents of the data memory occupied by variable 2 would have been pushed onto the save stack during forward execution and would be replaced when reversed, then variable register 2 would handled as just described for variable registers

3 and 4. Finally, packet 7 would uninstantiate variable register 1 and the emulator would be ready to reverse the label instruction in packet 6.

Remember that when this packet was executed in the forward direction, the previous program counter was pushed onto label register 1's stack. That address was the address of the instruction that called procedure Insert. By popping that address off label 1's stack and placing it in the program counter, the label instruction would be reversed. Once packet 6 was reversed, the animator would, most likely, again show a display similar to that of Figure 24, although the emulator would not be in the same state as described for Figure 24. This is because the program counter always points to the next instruction to be executed either forward or in reverse. So if the direction were changed to forward at this point, the program counter would be swapped with the previous program counter, and would then contain program address 7 (the address of the label instruction that was just reverse executed), and thus the current packet would be 6. So in order for the emulator to return to the state it was in with the display shown in Figure 24 and executing in the forward direction, the call to procedure Insert would have to be reversed as well.

To continue discussing the emulator, this example will pick up where it left off, with the emulator executing in the forward direction, the program counter at address 13, and the current packet being 12. The corresponding display is Figure 25.

At this point the next packet of interest is packet 13, which is meant to allocate space to store a value of type Node as a translation of the Pascal New function. For the E-machine, this is a simple task. First, eight data words would be allocated using the E-machine alloc instruction, which would push the address of the allocated memory onto the evaluation stack. The following pop instruction would pop the address off the stack and place it in the data memory location pointed to by the top of variable register 3's stack giving NewNode a value. Figure 26 shows what the animator might display once the pointer variable NewNode

contained a data memory address. Since NewNode is a pointer (or address), the contents of the data memory pointed to would be of interest; that is what is displayed beneath the NewNode variable in Figure 26.

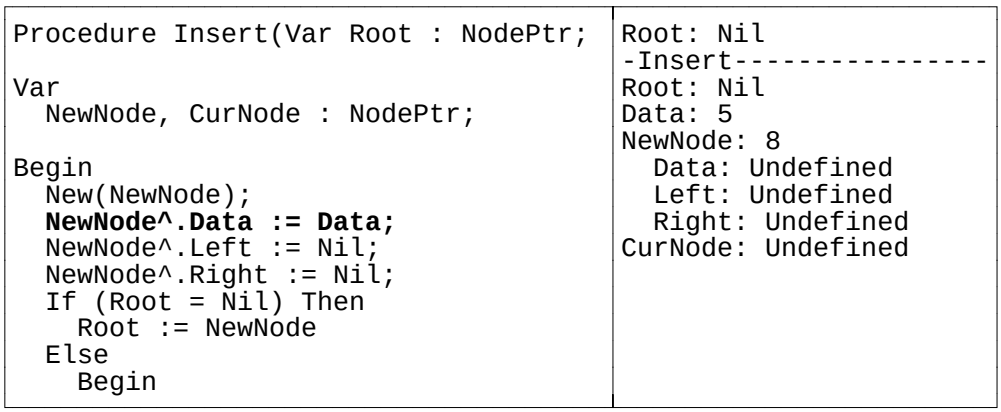


Figure 26
Display After Executing New(NewNode);

The next three packets are straightforward, designed to assign values to the fields of NewNode. Packet 17 is a good example of a conditional evaluation in the emulator. The value of Root would be pushed onto the evaluation stack, the constant NIL would also be pushed onto the evaluation stack, and then the eql instruction would be executed. The two are equal, so TRUE would be pushed onto the evaluation stack. The conditional branch would pop TRUE off the evaluation stack and continue with the next instruction in program memory since the branch was to occur only if the value on top of the stack were FALSE. Thus the then portion of the if would be executed, which would assign NewNode to Root. The branch instruction in packet 23 would send the emulator to packet 38.

Figure 27 shows what the animator display might look like after reaching packet 38. Notice the variable display and that not only Root in the procedure Insert has changed, but so has the instance of Root in the main program. This is because of the link instruction that was executed at the beginning of procedure Insert, which copied the top of the first Root's stack and pushed it onto the second Root's stack, thus making the

Root in the procedure Insert occupy the same data memory (as should be the case for call by reference parameters, such as Root).

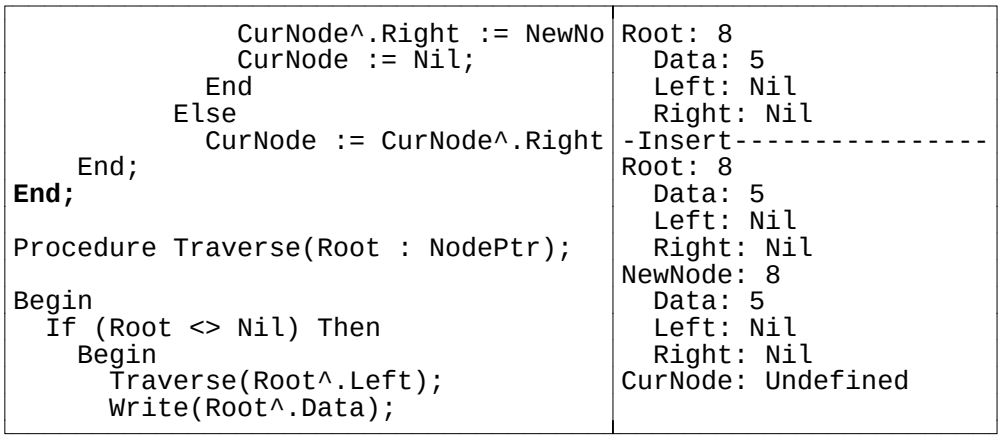


Figure 27
Display at end of Procedure Insert

The emulator would continue executing single packets at a time, either in the forward or reverse direction, as called by the program animator. In between calls to the emulator, the animator would update the screen displays in a manner similar to that shown in the example screen snapshots. The example presented here should be sufficient to understand how an animator could work in conjunction with the emulator to visualize programs.

CHAPTER 5

CREATING OBJECT PROGRAM FILES

Building Instructions

In order to write a compiler, the compiler writer needs to know how to create E-code instructions as translations of the high level language programs for which the compiler is written. The exact format of the instruction need not be known, although, it can be found in the "decex.h" header file. This header file must be included in the compiler's source code, as it contains the definition of the structure Instruction, which is the format of the E-code instruction expected by the Emulator. The compiler writer, then, need only follow the directions below to build the E-code instructions in the translation phase of the compiler.

An E-code instruction is a structure with several fields that must be filled in order to create the E-code instruction. A variable in the compiler must be declared of type Instruction to hold the instructions that are created. The structure is broken up into the following fields: opcode, addrmode, cflag, type, and data. Each of these fields must be filled with the appropriate information to create the E-code instruction.

The opcode field holds a code that specifies an E-code operation (e.g., push). There are no specific values for the opcodes; instead there is a defined constant to represent each of the possible operations. This is done so that if the E-machine emulator changes, the compiler only needs to be recompiled using the new constant definitions. The constants are names identical to the mnemonics given in Chapter 2 for the E-code instructions, with all letters in the name capitalized (e.g., the push instruction opcode constant is PUSH).

The addrmode field holds a code defining the addressing mode for the instruction. Once again named constants are used to specify the

addressing modes. There are six constants that make up the addressing mode: IMMEDIATE, ADDRESS, VARIABLE, INDEX, OFFSET, and INDIRECT. These constants are mixed and matched to create different addressing modes. For example, to specify variable indirect mode the constants VARIABLE and INDIRECT are or'ed together to create the addressing mode code. Note that not all combinations are legal and that mixing them incorrectly will yield unexpected results when the object program is run. IMMEDIATE is always used alone. Either ADDRESS or VARIABLE, but not both, may be used with zero or one of INDEX, OFFSET, or INDIRECT. Because of the manner in which addressing modes were implemented, no other combinations of constants should be used. A new method for handling addressing modes should be considered for future versions of the Emulator to eliminate this flaw.

The cflag (critical flag) field is used to mark the instruction as either critical or non-critical. There are two defined constants, CRITICAL and NONCRITICAL, which should be used as values for this field. Any value other than these constants will yield unpredictable results when the object program executes.

The type field holds a code for the data type involved for the instruction. This type is one of the basic E-machine types: real, integer, character, boolean, and address. The defined constants REAL, INTEGER, CHARACTER, BOOLEAN, and ADDRESS should be used as values for this field. Only one of these values is used, with one exception, the cast instruction. The cast instruction requires two types to be declared. This is done by joining two of the constants in the following manner. The constant for the type that is being cast from should be shifted left four bits and or'ed with the type being cast to (see the example below). Once again, values other than these constants will yield unpredictable results when executing the object program.

The data field is primarily used to hold the constant of the constant addressing mode when used; however, it is also used to hold the label numbers for branch and label instructions, and variable register

numbers for the link, unlink, inst, and uninst instructions. The data field is a union of the basic E-machine data types so that any data type can be placed in the field. Each field in the union is named after the basic E-machine type: real, integer, character, boolean, and address. To place a value in the data field, the appropriate union field must be selected (e.g., data.integer for integer values and data.real for floating point values). The integer field is used to store the label numbers and variable register numbers.

Note that not all of these fields are used for all instructions. The opcode field always has a value. The addrmode field is used only with those operations that require an address or data, such as push and pop. The add operation does not use the addrmode field since its operands and result use the evaluation stack. Some instructions behave the same when they are critical as when they are non-critical (those instructions in Chapter 2 that have only an explanation for Forward and Backward execution). In this case, cflag is ignored and need not be set to any particular value. Only those instructions that deal with data need a value in the type field, such as add, sub, and so on. Instructions like alloc and inst, which do nothing with a particular E-machine data type, do not need a value in the type field. The data field is used for constant addressing mode and those instructions needing a label or variable register number.

To further explain how to create E-code instructions, a few examples are given here. There will be a description of what the example demonstrates followed by an E-code instruction and the C code necessary to build such an instruction. The instruction will be built in a variable named instr of type Instruction.

Example 1:

This example shows an instruction having no value in the data field. The constant addressing mode is not used and the instruction does not expect a label or variable register, so the data field is unnecessary.

The instruction:

```
pop         n,I,V2
```

could be created in C as follows:

```
...
Instruction instr;

instr.opcode = POP;
instr.cflag = NONCRITICAL;
instr.type = INTEGER;
instr.addrmode = VARIABLE;
```

Example 2:

This example shows an instruction requiring no addressing mode or value in the data field. No value is needed in the data field for the same reasons as in the previous example, and no addressing mode is necessary since the operation uses the evaluation stack for its operands and storing of the result.

The instruction:

```
add         c,R
```

could be created in C as follows:

```
...
Instruction instr;

instr.opcode = ADD;
instr.cflag = CRITICAL;
instr.type = REAL;
```

Example 3:

This example shows an instruction needing no addressing mode or data type. The addressing mode is unnecessary for the same reasons as in the previous example, and the data type is not necessary since the instruction is dealing with a variable register and not an E-machine data value.

The instruction:

```
inst        c,V3
```

could be created in C as follows:

```
...
Instruction instr;

instr.opcode = INST;
instr.cflag = CRITICAL;
instr.data.integer = 3;
```

Example 4:

This example shows an instruction that uses the data field to hold a label number, which means that no addressing mode or type need be generated, because the label number is expected to be an integer in the data field. This instruction, also, does not require a cflag because there is no information to be saved that is critical for backup, except the previous program counter, which is maintained automatically.

The instruction:

```
br          8
```

could be created in C as follows:

```
...
Instruction instr;

instr.opcode = BR;
instr.data.integer = 8;
```

Example 5:

This example shows an instruction using the constant addressing mode; hence it requires storing the constant in the data field. There is no critical flag because the push operation does not store any critical information.

The instruction:

```
push        R,C1.5
```

could be created in C as follows:

```
...
Instruction instr;

instr.opcode = PUSH;
instr.type = REAL;
instr.data.real = 1.5;
```

Example 6:

This is an example of how to generate the type field for the cast instruction. This instruction casts an integer to a real. No other fields are filled since all that is needed are the two types.

The instruction:

```
cast        I,R
```

could be created in C as follows:

```
...
Instruction instr;
```

```
instr.opcode = CAST;
instr.type = (INTEGER << 4) | REAL;
```

The instructions of a compiled E=code program should be kept in an array numbered starting at 0. The array position of an instruction is also its program address. The program address is important for the label registers, as described below.

Creating Variable Registers

The type definition for the variable register structure is in the "variable.h" header file and is called VariableReg. The variable registers are labelled starting at 0 and must be kept track of by the compiler. The only field of concern to the compiler writer is the size field. Each variable in the source program takes up a certain amount of data memory space and the compiler must keep track of this for each variable register that is used. The size is measured in DataWords. Recall that DataWord is a data type representing the smallest accessible piece of memory, taking into account alignment requirements, and is defined in the "datamem.h" header file. The size of DataWord is a defined constant also in the "datamem.h" header file and is the number of bits in the DataWord. These constants may be different from one computer to another, depending on the memory word size and alignment requirements for that computer. For the Emulator presented in Appendix A, which was created for an IBM PC, DataWord is defined to be of C type unsigned char, which uses one byte or 8-bits of memory.

Creating The Label Registers

The structure for the label registers is defined in the "label.h" header file and is called LabelReg. The label registers are numbered starting at 0 and must be kept track of by the compiler. The only field in the label register structure that is important to the compiler writer is the address field. This field is intended to hold the program address

of the corresponding label instruction. For example, the address field of label register 0 will contain the program address of the instruction label 0.

Creating The Symbol Table

The symbol table can be built by creating an array of symbol table entries and creating the scope blocks as described in Chapter 3. An example is also given in Chapter 4. All of the fields, except the type field, are simple C types and can be filled with standard C statements. The type field can be filled by assigning one of the following enumerated constants to the field: ADDRESS, INTEGER, REAL, BOOLEAN, CHARACTER, PROCEDURE, HEADER, END, or RECORD. These constants are defined in the symbol table header file "symbol.h".

Creating The Packet Table

The packet table holds packet and scope information for the compiled program. Its structure is defined in the "packet.h" header file and is called Packet. Each packet is numbered sequentially starting with 0, the number representing the position of the packet in the packet table. All fields in this structure are important to the compiler writer and must be given values during compilation. The Packet structure consists of the fields startaddr, endaddr, startline, startcol, endline, endcol, and scope. The startaddr and endaddr fields hold the beginning and ending E-code object program address for a packet. The startline, startcol, endline, and endcol fields hold the beginning and ending lines and column positions of the original program source code that corresponds to the object program packet of instructions starting at program address startaddr and ending at program address endaddr. The scope field is an index into the symbol table pointing to the scope block for this packet. The packet table must be organized so that each successive packet contains higher numbered program addresses. This does not necessarily mean that

the startline, startcol, and so on, will be successively larger, although it is likely.

Format of the Object Code File

The object code file the compiler builds must contain all of the structures listed above plus the original source code. The object code file is broken up into sections, the order of which is not important. Each section begins with a section header, followed by a count of the number of items in the section. The section header is just a C integer type with a value of one of the defined constants: CODESECTION, VARIABLESECTION, LABELSECTION, SYMBOLSECTION, PACKETSECTION, and SOURCESECTION. The count is also a C integer type and its value must be the number of items for that section that have been written to the file. So, to write out the label section header and count, the following C code might be used:

```
section = LABELSECTION;
write(file, section, sizeof(int));
write(file, &count, sizeof(int));
```

CODESECTION is the section header for the E-code instruction section. The instructions must be in program address order and must include an instruction for every program address from 0 to the number of instructions, minus one. The count is the total number of instructions written to the object file, not the last program address.

VARIABLESECTION is the section header for the variable registers. The registers must be written in order starting at 0 and include all variable registers up to the number of variable registers, minus one. Each entry under VARIABLESECTION is the size of the variable (to be stored in the size field of the variable register structure). No other information is to be included. The count is the total number of variable registers used for the program, not the last variable register used.

LABELSECTION is the section header for the label registers. The registers must be written in order starting at 0 and include all label

registers up to the number of label registers, minus one. Only the address field of the label register structure should be written to the object file for each label register. The count is the total number of label registers used for the program, not the last variable register used.

SYMBOLSECTION is the section header for the symbol table. The symbol table must be written to the file exactly as it was created during translation. The full structure should be written to the object file for each entry in the symbol table. The count is the total number of symbol table entries used for the program.

PACKETSECTION is the section header for the packet table. The packets must be written in order starting with packet number 0 and include all packets up to the number of packets, minus one. The entire packet structure should be written to the object file for each packet in the packet table. The count is the total number of packets, not the last packet number.

SOURCESECTION is the section header for the original source code. Each line of source should be written in order, the first line numbered 0, (this is important for use with the packet table) up to the number of source lines, minus one. Each line of source should be written to the object file followed by a null character marking the end of the source line. The ending carriage return and line feed should not be written to the object file. The count is the total number of source lines, not the last source line number.

CHAPTER 6

CONCLUSIONS AND NEW DIRECTIONS

New Directions for the Emulator

The version of the E-machine emulator presented here was designed with procedural languages like Pascal and C in mind, since that they are currently the most common languages used for instruction. Thus FORTRAN, BASIC, Ada, Pascal, and Modula 2 should all be successful in this environment as well. Other language types, such as LISP and SMALLTALK are also certainly compilable to E-code, but animation of these would require different techniques. The E-machine has been designed to be flexible, modifiable, and extensible in order to handle many types of high level languages.

The design of the E-machine and Emulator are not intended to be fixed. For the reasons stated above, it is expected that the E-machine and Emulator will change over time, either to make things easier, or to support a new concept. The purpose of the E-machine and Emulator are to provide a foundation for a tool useful in the teaching and learning of programming languages and concepts fundamental to computer science, not for a production environment. Thus, the E-machine is expected to change as new features are included to support new concepts.

New Directions for the Program Animation Project

Since, as of yet, there is no program animator, there are many directions that can be taken from here. The next step in the project is to design an animator. At the same time, however, at least one compiler needs to be developed for use with the animator. The first version of the animator is likely to include rudimentary features for displaying variable values, current line of execution, parameter correspondences, and

statement counts for student programmers. Unlike debuggers (see, for example, [ALICE],[DR PASCAL], and [TURBO], the animator should be able to display these things in a completely automated fashion, so that the utter novice can watch the program in action. Reverse execution will be particularly useful to such a student, allowing backing up to observe complex actions numerous times. Fundamental computer science concepts, such as time complexity and space complexity will be reinforced through the animated display. Simple experiments based on changing program input will allow students to analyze the run time behavior of programs more effectively. The animation environment will also provide instructors with an affective tool for in-class demonstration of various concepts underlying programming and computer science.

Eventually the animator may also provide visualization of algorithms through graphical displays similar to those described in [Brown 88] and [London 85]. Graphs and trees may be visualized ([Jabolonowski 89], [Pazel 89], and [Reingold 81]), so that the mechanics of various algorithms, such as B-tree insertion, can be more easily understood. Recursive procedures might be unravelled in some visual fashion to get the concept of recursion across better. The possibilities are endless. At the very least, however, the animator must somehow display the mechanics of programs so that instructors can better get the ideas of program dynamics across to the students, and the students can view animated programs on their own for learning.

Conclusions

The E-machine and its emulator provide the necessary foundation for further development of a comprehensive program animation system for teaching and learning concepts fundamental to programming and computer science. It follows as the logical next step in an ongoing project in this arena (see [Patton 89], [Meng 83], [Ng 82-1, 82-2], and [Ross 81, 82, 88]). The special features of reverse execution, close associations with

the symbol table and source code of a high level language program, and packet-at-a-time execution will greatly facilitate the development of an animator. The resulting animation system will be useable for instruction at both secondary and post-secondary institutions and could be developed for use as a dynamic companion to an introductory computer science textbook.

The E-machine and the planned animation environment are both intended as evolutionary systems that will undergo constant improvement as more is understood about program animation and its effect on teaching and learning. Thus, this incarnation of the E-machine, although complete, should be considered transitory. Hopefully, the E-machine design presented here, as well as the structure of the emulator, will make modifications and extensions quite easy.

REFERENCES CITED

ALICE:The Personal Pascal. Looking Glass Software Limited, 123 King St. N. Waterloo, ON, N2J2X8. 1986.

Birch, M. L., Patton, S. D., and Ross, R. J. 1990. The E-machine: A Virtual Computer in Support of Program Animation. Unpublished paper.

Brown, M. H. Exploring Algorithms Using Balsa-II. COMPUTER. vol. 21, no. 5, May 88, pp 14-36.

Dr. Pascal User Manual. Visible Software. P. O. Box 7788, Princeton, NJ. 1989.

Hille, R. F. and Higginbottom, T. F. A System for Visible Execution of Pascal Programs. The Australian Computer Journal. vol. 15, no. 2, May 83, pp 76-77.

Jabolonowski, D. and Guarna, V. GMB: A Tool for Manipulating and Animating Graph Data Structures. Software--Practice and Experience, vol. 19, no. 3, March 1989, pp 283-301.

London, R., and Duisberg, R. Animating Programs Using Smalltalk. COMPUTER, vol. 18, no. 8, August 1985, pp 61-71.

Meng-Kawalek, L. A Pascal Pedagogical System for the Convertsational Monitor System. Unpublished MS project. Computer Science Department, Washington State University. June 1983.

Ng, C. Ling Users Guide. Unpublished MS project. Computer Science Department, Washington State University. June 1982.

_____. Ling Programmers Guide. Unpublished MS project. Computer Science Department, Washington State University. June 1982.

Patton, S. D. The E-machine: Supporting the Teaching of Program Execution Dynamics. MS thesis. Computer Science Department, Montana State University. June 1989.

Pazel, D. DS-Viewer--An Interactive Graphical Data Structure Presentation Facility. IBM Systems Journal, vol. 28, no. 2, 1989, pp 307-323.

Reingold, E. and Tilford, J. Tidier Drawings of Trees. IEEE Transactions on Software Engineering, vol se-7, no. 2, March 1981, pp 223-228.

Ross, R. J. LOPLE: A Dynamic Library of Programming Language Examples. ACM SIGCUE Bulletin, 1981.

_____. Teaching Programming to the Deaf. ACM SIGCAPH Newsletter, no. 30, Autumn 1982, pp 18-24.

_____. DYNAMOD USER'S GUIDE Version 2.0 Release 1.1. Technical Report 88-1, Computer Science Department, Montana State University. June 1988.

Turbo Debugger 2.0. Borland International. 1800 Green Hills Road, Scotts Valley, California. 1990.

APPENDIX

Figure 28
Emulator Source Code

decex.h

```
/* decex.h: decode/execute header */

/* An enumeration of all machine operations available in the E-machine. */
typedef enum {
  PUSH, PUSHA, POP, POPIR, POPAR, LOADIR, LOADAR,
  ADD, SUB, MULT, DIV, NEG, AND, OR, XOR, NOT, SHL, SHR, MOD,
  CAST, LABEL, BR, BRT, BRF, EQL, NEQL, LESS, LEQL, GTR, GEQL, CALL,
  RETURN, ALLOC, UNALLOC, INST, UNINST, LINK, UNLINK, NOP, LASTOP
} Opcode; /* LASTOP is used to mark the size of the enumeration. */

typedef enum { CRITICAL, NONCRITICAL } ModeType;

typedef struct {
  Opcode opcode;
  ModeType mode;
  unsigned char type;
  DataValue data;
  int addrmode;
} Instruction;

typedef int ProgAddress;

/* Prototype for execution function, which executes the given machine
   instruction with the given decoded arguments. */

void executepacket( );

void setpc( ProgAddress );

void setppc( ProgAddress );

void getpc( ProgAddress * );

void getppc( ProgAddress * );
```

decex.c

```
/* decex.c: Fetch/Decode/Execute Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"
#include "flags.h"
#include "decaddr.h"
#include "execute.h"
#include "variable.h"
#include "symbol.h"
#include "packet.h"
```

Figure 28 (continued)

```
/* Decode/Execute information table. */
struct {
  void (*func)();
  ArgType expected;
} opinfo[LASTOP] = {
     push, DATA,
     pusha, ADDR,
     pop, ADDR,
     popir, NONE,
     popar, NONE,
     loadir, DATA,
     loadar, DATA,
     add, NONE,
     sub, NONE,
     mult, NONE,
     div, NONE,
     neg, NONE,
     and, NONE,
     or, NONE,
     xor, NONE,
     not, NONE,
     shl, DATA,
     shr, DATA,
     mod, NONE,
     cast, NONE,
     label, NONE,
     br, NONE,
     brt, NONE,
     brf, NONE,
     eql, NONE,
     neql, NONE,
     less, NONE,
     leql, NONE,
     gtr, NONE,
     geql, NONE,
     call, NONE,
     returnf, NONE,
     alloc, DATA,
     unalloc, DATA,
     inst, NONE,
     uninst, NONE,
     link, NONE,
     unlink, NONE,
     nop, NONE };

ProgAddress programcounter = 0, previouspc = -1;

Packet currentpacket;

PacketNum currpacknum = 0;

void executepacket( ) {

  DataAddress address;
  DataValue data;
  Instruction instr;

  newpacket();
  findpacket(programcounter, &currentpacket, &currpacknum);
  while (programcounter >= currentpacket.startaddr &&
```

Figure 28 (continued)

```
       programcounter <= currentpacket.endaddr) {
    resetflag(BRANCH);
    getinstruction(programcounter, &instr);
    decodeaddr(instr, opinfo[instr.opcode].expected, &address, &data);
    (*opinfo[instr.opcode].func)(instr, address, data);
    if (getflag(BRANCH) == FALSE) {
      previouspc = programcounter;
      incpc();
    }
  }
  findpacket(programcounter, &currentpacket, &currpacknum);
}


/* Make program counter point to next instruction. */
void incpc() {

  if (getflag(FORWARD) == TRUE) {
    programcounter++;
  }
  else {
    programcounter--;
  }
}

/* Set program counter to particular address. Would be used
   to handle a branch. */


void setpc( ProgAddress address ) {

  programcounter = address;
}

void setppc( ProgAddress address ) {

  previouspc = address;
}

void getpc( ProgAddress *address ) {

  *address = programcounter;
}

void getppc( ProgAddress *address ) {

  *address = previouspc;
}

void reverse() {

  ProgAddress temp;

  if (getflag(FORWARD) == TRUE) {
    resetflag(FORWARD);
  }
  else {
    setflag(FORWARD);
  }
  temp = programcounter;
```

Figure 28 (continued)

```
  programcounter = previouspc;
  previouspc = temp;
}

void getcurrpacket(Packet *packet) {

  *packet = currentpacket;
}

void init() {

  findpacket(programcounter, &currentpacket, &currpacknum);
}
```

decaddr.h

```
/* decaddr.h: Decode Address header */

/* Bit positions of types of addressing modes. */
enum { IMMEDIATE = 1, VARIABLE = 2, ADDRREG = 4, INDEXED = 8,
       OFFSET = 16, INDIRECT = 32, INDEXFIRST = 64 };

/* An instruction can expect data, a data address, or nothing. */
typedef enum {
  DATA, ADDR, NONE
} ArgType;

void setar( DataAddress );

void getar( DataAddress * );

void setir( IntegerType );

void getir( IntegerType * );

void decodeaddr( Instruction, ArgType, DataAddress *, DataValue * );
```

decaddr.c

```
/* decaddr.c: Address Mode Decoder Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"
#include "decaddr.h"
#include "variable.h"

/* The index and address registers. Declared as DataValue to
   be compatible with the Save Stack. */
DataValue indexreg;
DataValue addressreg;

void  decodeaddr(  Instruction  instr,  ArgType  expected,  DataAddress
*address,
             DataValue *data ) {

  DataValue temp;
  DataWord defined;

  if (expected != NONE) {
```

Figure 28 (continued)

```
    if (instr.addrmode & IMMEDIATE) {
      *address = NULLDATAADDR;
      *data = instr.data;
    }
    else {
      if (instr.addrmode & ADDRREG) {
      *address = addressreg.address;
      }
      else {
      getvaraddress(instr.data.integer, address);
      }
      if (instr.addrmode & INDEXED) {
      *address += indexreg.integer * datasize(instr.type);
      }
      else if (instr.addrmode & OFFSET) {
      *address += indexreg.integer;
      }
      else if (instr.addrmode & INDIRECT) {
      readdata(ADDRESS, *address, data, &defined);
      if (defined == 0) {
        fault(UNDEFDATA);
      }
      }
      if (getfault() == NOFAULT) {
      if (*address != NULLDATAADDR) {
        readdata(instr.type, *address, data, &defined);
      }
      }
      if ((expected == DATA) && (defined == 0)) {
      fault(UNDEFDATA);
      }
      else if ((expected == ADDR) && (*address == NULLDATAADDR)) {
      fault(ILLEGALMODE);
      }
    }
  }
}


void setar( DataAddress address ) {

  addressreg.address = address;
}

void getar( DataAddress *address ) {

  *address = addressreg.address;
}

void setir( IntegerType index ) {

  indexreg.integer = index;
}

void getir( IntegerType *index ) {

  *index = indexreg.integer;
}
```

Figure 28 (continued)

progmem.h

```
/* progmem.h: Program Memory header */

#define NULLPROGADDR (ProgAddress)-1

void getinstruction( ProgAddress, Instruction * );

void incpc();

void setpc( ProgAddress );
```

progmem.c

```
/* progmem.c: Program Memory Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"
#include "decaddr.h"
#include <stdlib.h>

/* Program Memory structure. */
Instruction *programmem;
static ProgAddress lastlocation = 0;

/* Program counter. */
static ProgAddress progcounter;

/* Return next instruction to be executed. */
void getinstruction( ProgAddress address, Instruction *instr ) {

  if ((address < 0) || (address > lastlocation)) {
    /* ERROR: bad program address. */
    fault(BADPROGADDR);
  }
  else {
    *instr = programmem[address];
  }
}


/* Load program instructions from a file. */
void loadprogram( int fileid ) {

  IntegerType size;
  ProgAddress i;

  if (sizeof(size) != read(fileid, &size, sizeof(size))) {
    /* ERROR: bad file format. */
    fault(BADFILE);
  }
  else {
    programmem = calloc(size, sizeof(Instruction));
    if (programmem == NULL) {
      /* Not enough system memory left. */
      fault(OUTMEM);
    }
    else {
      i = 0;
```

Figure 28 (continued)

```
      while ((getfault() == NOFAULT) && (i < size)) {
        if (read(fileid, &(programmem[i]), sizeof(Instruction)) !=
            sizeof(Instruction)) fault(BADFILE);
      }
      lastlocation = size - 1;
      setpc((ProgAddress) 0);
    }
  }
}
```

execute.h

```
/* execute.h: Instruction Execution Module header */

void push(Instruction, DataAddress, DataValue);

void pusha(Instruction, DataAddress, DataValue);

void pop(Instruction, DataAddress, DataValue);

void popir(Instruction, DataAddress, DataValue);

void popar(Instruction, DataAddress, DataValue);

void loadir(Instruction, DataAddress, DataValue);

void loadar(Instruction, DataAddress, DataValue);

void add(Instruction, DataAddress, DataValue);

void mult(Instruction, DataAddress, DataValue);

void sub(Instruction, DataAddress, DataValue);

void div(Instruction, DataAddress, DataValue);

void neg(Instruction, DataAddress, DataValue);

void and(Instruction, DataAddress, DataValue);

void or(Instruction, DataAddress, DataValue);

void xor(Instruction, DataAddress, DataValue);

void not(Instruction, DataAddress, DataValue);

void shl(Instruction, DataAddress, DataValue);

void shr(Instruction, DataAddress, DataValue);

void br(Instruction, DataAddress, DataValue);

void brt(Instruction, DataAddress, DataValue);

void brf(Instruction, DataAddress, DataValue);

void eql(Instruction, DataAddress, DataValue);

void neql(Instruction, DataAddress, DataValue);
```

Figure 28 (continued)

```
void less(Instruction, DataAddress, DataValue);

void leql(Instruction, DataAddress, DataValue);

void gtr(Instruction, DataAddress, DataValue);

void geql(Instruction, DataAddress, DataValue);

void mod(Instruction, DataAddress, DataValue);

void label(Instruction, DataAddress, DataValue);

void call(Instruction, DataAddress, DataValue);

void cast(Instruction, DataAddress, DataValue);

void returnf(Instruction, DataAddress, DataValue);

void alloc(Instruction, DataAddress, DataValue);

void unalloc(Instruction, DataAddress, DataValue);

void inst(Instruction, DataAddress, DataValue);

void uninst(Instruction, DataAddress, DataValue);

void link(Instruction, DataAddress, DataValue);

void unlink(Instruction, DataAddress, DataValue);

void nop(Instruction, DataAddress, DataValue);
```

execute.c

```
/* execute.c: Instruction Execution Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"
#include "flags.h"
#include "save.h"
#include "eval.h"
#include "label.h"
#include "variable.h"
#include "execute.h"

void push(Instruction instr, DataAddress address, DataValue data) {

  if (getflag(FORWARD) == TRUE) {
    pusheval(data);
  }
  else {
    popeval(&data);
  }
}

void pusha(Instruction instr, DataAddress address, DataValue data) {
```

Figure 28 (continued)

```
  DataValue tempdata;

  if (getflag(FORWARD) == TRUE) {
    tempdata.address = address;
    pusheval(tempdata);
  }
  else {
    popeval(&data);
  }
}

void nop(Instruction instr, DataAddress address, DataValue data) {

/* This procedure does exactly what the instruction is suppose to do */
/*    NOTHING   */

}


void pop(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    if (instr.mode == CRITICAL) {
      readdata(instr.type, address, &tempdata, &defined);
      savedata(tempdata, instr.type, defined);
    }
    popeval(&tempdata);
    writedata(instr.type, address, tempdata);
  }
  else {
    pusheval(data);
    if (instr.mode == CRITICAL) {
      unsavemem(address, datasize(instr.type));
    }
  }
}

void popar(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    if (instr.mode == CRITICAL) {
      getar( &tempdata.address );
      savedata(tempdata, ADDRESS, HIGHWORD);
    }
    popeval(&tempdata);
    setar(tempdata.address);
  }
  else {
    getar(&tempdata.address);
    pusheval(tempdata);
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, ADDRESS, &defined);
    }
  }
```

Figure 28 (continued)

```
}

void popir(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    if (instr.mode == CRITICAL) {
      getir(&tempdata.integer);
      savedata(tempdata, INTEGER, HIGHWORD);
    }
    popeval(&tempdata);
    setir(tempdata.integer);
  }
  else {
    getir(&tempdata.integer);
    pusheval(tempdata);
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, INTEGER, &defined);
      setir(tempdata.integer);
    }
  }
}

void loadar(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    if (instr.mode == CRITICAL) {
      getar( &tempdata.address );
      savedata(tempdata, ADDRESS, HIGHWORD);
    }
    setar(data.address);
  }
  else {
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, ADDRESS, &defined);
    }
    setar(tempdata.address);
  }
}

void loadir(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    if (instr.mode == CRITICAL) {
      getir(&tempdata.integer);
      savedata(tempdata, INTEGER, HIGHWORD);
    }
    setir(data.integer);
  }
  else {
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, INTEGER, &defined);
```

Figure 28 (continued)

```
    }
    setir(tempdata.integer);
  }
}

void sub(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer - op1.integer;
        pusheval(result);
        break;

      case REAL:
        result.real = op2.real - op1.real;
        pusheval(result);
        break;

      case CHARACTER:
        result.character = op2.character - op1.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = op2.boolean ^ op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
        result.address = op2.address - op1.address;
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void add(Instruction instr, DataAddress address, DataValue data) {
```

Figure 28 (continued)

```
  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer + op1.integer;
        pusheval(result);
        break;

      case REAL:
        result.real = op2.real + op1.real;
        pusheval(result);
        break;

      case CHARACTER:
        result.character = op2.character + op1.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = op2.boolean | op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
        result.address = op2.address + op1.address;
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void div(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
```

Figure 28 (continued)

```
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op1.integer != 0) {
          result.integer = op2.integer / op1.integer;
          pusheval(result);
          }
          else {
            fault(DIVBYZERO);
          }
        break;

      case REAL:
          if (op1.real != 0.0) {
          result.real = op2.real / op1.real;
          pusheval(result);
          }
          else {
            fault(DIVBYZERO);
          }
        break;

      case CHARACTER:
          if (op1.character != 0) {
            result.character = op2.character / op1.character;
            pusheval(result);
          }
          else {
            fault(DIVBYZERO);
          }
        break;

      case BOOLEAN:
          fault(ILLEGALTYPE);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void mod(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
```

Figure 28 (continued)

```
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op1.integer != 0) {
          result.integer = op2.integer % op1.integer;
          pusheval(result);
          }
          else {
            fault(DIVBYZERO);
          }
        break;

      case REAL:
        fault(ILLEGALTYPE);
        break;

      case CHARACTER:
          if (op1.character != 0) {
          result.character = op2.character % op1.character;
            pusheval(result);
          }
          else {
            fault(DIVBYZERO);
          }
        break;

      case BOOLEAN:
          fault(ILLEGALTYPE);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void mult(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
```

Figure 28 (continued)

```
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer * op1.integer;
        pusheval(result);
        break;

      case REAL:
        result.real = op2.real * op1.real;
        pusheval(result);
        break;

      case CHARACTER:
        result.character = op2.character * op1.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = op2.boolean & op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void and(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
```

Figure 28 (continued)

```
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer & op1.integer;
        pusheval(result);
        break;

      case REAL:
          fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op2.character & op1.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = op2.boolean & op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void or(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer | op1.integer;
        pusheval(result);
        break;
```

Figure 28 (continued)

```
      case REAL:
          fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op2.character | op1.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = op2.boolean | op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void xor(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op2.integer ^ op1.integer;
        pusheval(result);
        break;

      case REAL:
          fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op2.character ^ op1.character;
        pusheval(result);
        break;
```

Figure 28 (continued)

```
      case BOOLEAN:
        result.boolean = op2.boolean ^ op1.boolean;
        pusheval(result);
        break;

      case ADDRESS:
          fault(ILLEGALTYPE);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void shl(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op1.integer << data.integer;
        pusheval(result);
        break;

      case REAL:
          fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op1.character << data.integer;
        pusheval(result);
        break;

      case BOOLEAN:
        fault(ILLEGALTYPE);
        break;

      case ADDRESS:
        result.address = op1.address << data.integer;
        pusheval(result);
        break;
      }
    }
  }
```

Figure 28 (continued)

```
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op1);
  }
}

void shr(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
        result.integer = op1.integer >> data.integer;
        pusheval(result);
        break;

      case REAL:
          fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op1.character >> data.integer;
        pusheval(result);
        break;

      case BOOLEAN:
        fault(ILLEGALTYPE);
        break;

      case ADDRESS:
        result.address = op1.address >> data.integer;
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op1);
  }
}

void not(Instruction instr, DataAddress address, DataValue data) {

  DataValue op, result;
```

Figure 28 (continued)

```
  popeval(&op);
  if (getfault() == NOFAULT) {

    switch(instr.type) {

      case INTEGER:
        result.integer = op.integer ^ 0xFFFFFFFF;
        pusheval(result);
        break;

      case REAL:
        fault(ILLEGALTYPE);
        break;

      case CHARACTER:
        result.character = op.character ^ 0xFF;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = (op.boolean == TRUE) ? FALSE : TRUE;
        pusheval(result);
        break;

      case ADDRESS: /* addresses cannot be negated */
        fault(ILLEGALTYPE);
        break;
    }
  }
}

void eql(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op2.integer == op1.integer) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case REAL:
          if (op2.real == op1.real) {
            result.boolean = TRUE;
          }
```

Figure 28 (continued)

```
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case CHARACTER:
          if (op2.character == op1.character) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case BOOLEAN:
          if (op2.boolean == op1.boolean) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case ADDRESS:
          if (op2.address == op1.address) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void neql(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
```

Figure 28 (continued)

```
    savedata(op2, instr.type, HIGHWORD);
    }
    switch(instr.type) {

    case INTEGER:
        if (op2.integer != op1.integer) {
          result.boolean = TRUE;
        }
        else {
          result.boolean = FALSE;
        }
      pusheval(result);
      break;

    case REAL:
        if (op2.real != op1.real) {
          result.boolean = TRUE;
        }
        else {
          result.boolean = FALSE;
        }
      pusheval(result);
      break;

    case CHARACTER:
        if (op2.character != op1.character) {
          result.boolean = TRUE;
        }
        else {
          result.boolean = FALSE;
        }
      pusheval(result);
      break;

    case BOOLEAN:
        if (op2.boolean != op1.boolean) {
          result.boolean = TRUE;
        }
        else {
          result.boolean = FALSE;
        }
      pusheval(result);
      break;

    case ADDRESS:
        if (op2.address != op1.address) {
          result.boolean = TRUE;
        }
        else {
          result.boolean = FALSE;
        }
      pusheval(result);
      break;
    }
  }
}
else {
  popeval(&result);
  if (instr.mode == CRITICAL) {
    unsavedata(&op2, instr.type, &defined);
```

Figure 28 (continued)

```
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void less(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op2.integer < op1.integer) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case REAL:
          if (op2.real < op1.real) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case CHARACTER:
          if (op2.character < op1.character) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case BOOLEAN:
          if (op2.boolean < op1.boolean) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
```

Figure 28 (continued)

```
        break;

      case ADDRESS:
          if (op2.address < op1.address) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void leql(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op2.integer <= op1.integer) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case REAL:
          if (op2.real <= op1.real) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
```

Figure 28 (continued)

```
      case CHARACTER:
          if (op2.character <= op1.character) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case BOOLEAN:
          if (op2.boolean <= op1.boolean) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case ADDRESS:
          if (op2.address <= op1.address) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void gtr(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op2.integer > op1.integer) {
```

Figure 28 (continued)

```
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case REAL:
          if (op2.real > op1.real) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case CHARACTER:
          if (op2.character > op1.character) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case BOOLEAN:
          if (op2.boolean > op1.boolean) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case ADDRESS:
          if (op2.address > op1.address) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}
```

Figure 28 (continued)

```
void geql(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, op2, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&op1);
    popeval(&op2);
    if (getfault() == NOFAULT) {
      if (instr.mode == CRITICAL) {
      savedata(op1, instr.type, HIGHWORD);
      savedata(op2, instr.type, HIGHWORD);
      }
      switch(instr.type) {

      case INTEGER:
          if (op2.integer >= op1.integer) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case REAL:
          if (op2.real >= op1.real) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case CHARACTER:
          if (op2.character >= op1.character) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case BOOLEAN:
          if (op2.boolean >= op1.boolean) {
            result.boolean = TRUE;
          }
          else {
            result.boolean = FALSE;
          }
        pusheval(result);
        break;

      case ADDRESS:
          if (op2.address >= op1.address) {
            result.boolean = TRUE;
          }
          else {
```

Figure 28 (continued)

```
            result.boolean = FALSE;
          }
        pusheval(result);
        break;
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op2, instr.type, &defined);
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op2);
    pusheval(op1);
  }
}

void neg(Instruction instr, DataAddress address, DataValue data) {

  DataValue op, result;

  popeval(&op);
  if (getfault() == NOFAULT) {

    switch(instr.type) {

      case INTEGER:
        result.integer = -op.integer;
        pusheval(result);
        break;

      case REAL:
        result.real = -op.real;
        pusheval(result);
        break;

      case CHARACTER:
        result.character = -op.character;
        pusheval(result);
        break;

      case BOOLEAN:
        result.boolean = (op.boolean == TRUE) ? FALSE : TRUE;
        pusheval(result);
        break;

      case ADDRESS: /* addresses cannot be negated */
        fault(ILLEGALTYPE);
        break;
    }
  }
}

void cast(Instruction instr, DataAddress address, DataValue data) {

  DataValue op1, result;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
```

Figure 28 (continued)

```
popeval(&op1);
if (getfault() == NOFAULT) {
  if (instr.mode == CRITICAL) {
  savedata(op1, instr.type, HIGHWORD);
  }
  switch((instr.type & 0xF0) >> 4) {

  case INTEGER:
      switch (instr.type & 0x0F) {

        case INTEGER:
          break;

        case REAL:
          result.real = (RealType) op1.integer;
          pusheval(result);
          break;

        case CHARACTER:
          result.character = (CharacterType) op1.integer;
          pusheval(result);
          break;

        default:
          fault(ILLEGALTYPE);
          break;
      }

    case REAL:
      switch (instr.type & 0x0F) {

        case REAL:
          break;

        case INTEGER:
          result.integer = (IntegerType)op1.real;
          pusheval(result);
          break;

        case CHARACTER:
          result.character = (CharacterType) op1.real;
          pusheval(result);
          break;

        default:
          fault(ILLEGALTYPE);
          break;
      }

  case CHARACTER:
      switch (instr.type & 0x0F) {

        case CHARACTER:
          break;

        case REAL:
          result.real = (RealType) op1.character;
          pusheval(result);
          break;
```

Figure 28 (continued)

```
            case INTEGER:
              result.integer = (IntegerType) op1.character;
              pusheval(result);
              break;

            default:
              fault(ILLEGALTYPE);
              break;
          }
      }
    }
  }
  else {
    popeval(&result);
    if (instr.mode == CRITICAL) {
      unsavedata(&op1, instr.type, &defined);
    }
    pusheval(op1);
  }
}


void link(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;

  if (getflag(FORWARD) == TRUE) {
    popeval(&tempdata);
    pushvariable(instr.data.integer, tempdata.address);
  }
  else {
    popvariable(instr.data.integer, &tempdata.address);
    pusheval(tempdata);
  }
}

void unlink(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popvariable(instr.data.integer, &tempdata.address);
    savedata(tempdata, ADDRESS, HIGHWORD);
  }
  else {
    unsavedata(&tempdata, ADDRESS, &defined);
    pushvariable(instr.data.integer, tempdata.address);
  }
}

void alloc(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;

  if (getflag(FORWARD) == TRUE) {
    tempdata.address = NULLDATAADDR;
    allocdata(&tempdata.address, data.integer);
    pusheval(tempdata);
  }
```

Figure 28 (continued)

```
  else {
    popeval(&tempdata);
    unallocdata(tempdata.address, data.integer);
  }
}

void unalloc(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;

  if (getflag(FORWARD) == TRUE) {
    popeval(&tempdata);
    if (instr.mode == CRITICAL) {
      savemem( tempdata.address, data.integer );
      savedata(tempdata, ADDRESS, HIGHWORD);
    }
    unallocdata(tempdata.address, data.integer);
  }
  else {
    tempdata.address = NULLDATAADDR;
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, ADDRESS, &defined);
    }
    allocdata(&tempdata.address, data.integer);
    if (instr.mode == CRITICAL) {
      unsavemem(tempdata.address, data.integer);
    }
    pusheval(tempdata);
  }
}

void inst(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  IntegerType size;

  if (getflag(FORWARD) == TRUE) {
    getvarsize(instr.data.integer, &size);
    tempdata.address = NULLDATAADDR;
    allocdata(&tempdata.address, size);
    if (tempdata.address != NULLDATAADDR) {
      pushvariable(instr.data.integer, tempdata.address);
    }
    else {
      fault(OUTMEM);
    }
  }
  else {
    popvariable(instr.data.integer, &tempdata.address);
    getvarsize(instr.data.integer, &size);
    unallocdata(tempdata.address, size);
  }
}

void uninst(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  IntegerType size;
  DataWord defined;
```

Figure 28 (continued)

```
  if (getflag(FORWARD) == TRUE) {
    getvaraddress(instr.data.integer, &tempdata.address);
    getvarsize(instr.data.integer, &size);
    if (instr.mode == CRITICAL) {
      savemem(tempdata.address, size);
    }
    unallocdata(tempdata.address, size);
    popvariable(instr.data.integer, &tempdata.address);
    if (instr.mode == CRITICAL) {
      savedata(tempdata, ADDRESS, HIGHWORD);
    }
  }
  else {
    tempdata.address = NULLDATAADDR;
    getvarsize(instr.data.integer, &size);
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, ADDRESS, &defined);
    }
    allocdata(&tempdata.address, size);
    pushvariable(instr.data.integer, tempdata.address);
    if (instr.mode == CRITICAL) {
      unsavemem(tempdata.address, size);
    }
  }
}


void br(Instruction instr, DataAddress address, DataValue data) {

  ProgAddress progaddr, progaddr2;

  if (getflag(FORWARD) == TRUE) {
    getlabeladdress(instr.data.integer, &progaddr2);
    if (getfault() == NOFAULT) {
      getpc(&progaddr);
      setppc(progaddr);
      setpc(progaddr2);
      setflag(BRANCH);
    }
  }
}

void brt(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;
  ProgAddress progaddr, progaddr2;

  if (getflag(FORWARD) == TRUE) {
    popeval(&tempdata);
    if (instr.mode == CRITICAL) {
      savedata(tempdata, BOOLEAN, HIGHWORD);
    }
    if (tempdata.boolean == TRUE) {
      getlabeladdress(instr.data.integer, &progaddr2);
      if (getfault() == NOFAULT) {
      getpc(&progaddr);
      setppc(progaddr);
      setpc(progaddr2);
      setflag(BRANCH);
```

Figure 28 (continued)

```
      }
    }
  }
  else {
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, BOOLEAN, &defined);
      pusheval(tempdata);
    }
    else {
      tempdata.boolean = TRUE;
      pusheval(tempdata);
    }
  }
}

void brf(Instruction instr, DataAddress address, DataValue data) {

  DataValue tempdata;
  DataWord defined;
  ProgAddress progaddr, progaddr2;

  if (getflag(FORWARD) == TRUE) {
    popeval(&tempdata);
    if (instr.mode == CRITICAL) {
      savedata(tempdata, BOOLEAN, HIGHWORD);
    }
    if (tempdata.boolean == FALSE) {
      getlabeladdress(instr.data.integer, &progaddr2);
      if (getfault() == NOFAULT) {
      getpc(&progaddr);
      setppc(progaddr);
      setpc(progaddr2);
      setflag(BRANCH);
      }
    }
  }
  else {
    if (instr.mode == CRITICAL) {
      unsavedata(&tempdata, BOOLEAN, &defined);
      pusheval(tempdata);
    }
    else {
      tempdata.boolean = FALSE;
      pusheval(tempdata);
    }
  }
}

void label(Instruction instr, DataAddress address, DataValue data) {

  ProgAddress progaddr;

  if (getflag(FORWARD) == TRUE) {
    getppc( &progaddr );
    pushlabel(instr.data.integer, progaddr);
  }
  else {
    getpc(&progaddr);
    setppc(progaddr);
    poplabel(instr.data.integer, &progaddr);
```

Figure 28 (continued)

```
    setpc(progaddr);
    setflag(BRANCH);
  }
}

void call(Instruction instr, DataAddress address, DataValue data) {

  ProgAddress progaddr;

  if (getflag(FORWARD) == TRUE) {
    getpc(&progaddr);
    setppc(progaddr);
    pushcall(progaddr+1);
    getlabeladdress(instr.data.integer, &progaddr);
    setpc(progaddr);
    setflag(BRANCH);
  }
  else {
  }
}

void returnf(Instruction instr, DataAddress address, DataValue data) {

  ProgAddress progaddr;

  if (getflag(FORWARD) == TRUE) {
    getpc(&progaddr);
    setppc(progaddr);
    popcall(&progaddr);
    setpc(progaddr);
    setflag(BRANCH);
  }
}
```

datamem.h

```
/* datamem.h: data memory header */

/* definitions for memory word types and sizes */
typedef unsigned char DataWord;
typedef unsigned int DoubleWord;
#define WORDSIZE 8
#define DBLWORDSIZE 16
#define HIGHWORD 0xFFFFu
#define HIGHDBLWORD 0xFFFFFFFFul


/* Enumeration of basic E-machine data types. */
typedef enum {
  BOOLEAN, INTEGER, REAL, ADDRESS, CHARACTER, LASTTYPE
} DataType;

/* Type definition for DataAddress, which is used to access data memory.
*/
typedef unsigned int DataAddress;

#define NULLDATAADDR (DataAddress)-1

/* Type definitions for basic E-machine data types.  Needed for casting
   void pointers.  These may be different on different machines.  i.e.
   32 bits is a good size for an integer which long int is for Turbo C
```

Figure 28 (continued)

```
   on an IBM PC, but is only int on a VAX. */
typedef enum { FALSE, TRUE } BooleanType;
typedef long int IntegerType;
typedef float RealType;
typedef DataAddress AddressType;
typedef char CharacterType;


/* DataValue is a union of all the basic E-machine types.  This allows
   for easy storage of all data types. */
typedef union {
  BooleanType   boolean;
  IntegerType   integer;
  RealType      real;
  AddressType   address;
  CharacterType character;
} DataValue;


int datasize( DataType );

/* writedata places DataValue at DataAddress and marks DataAddress as
   defined. */
void writedata( DataType, DataAddress, DataValue );

/* readdata returns DataValue from DataAddress. */
void readdata( DataType, DataAddress, DataValue *, DataWord * );

/* undefine marks the DataAddress as undefined. */
void undefine( DataAddress, DataValue );

/* clear clears all of the data memory space and marks it undefined. */
void clear( void );

/* allocate returns the DataAddress of a free area if one exists. */
void allocdata( DataAddress *, IntegerType /* size */ );

/* unallocate frees the data at DataAddress and marks it as undefined. */
void unallocdata( DataAddress, IntegerType /* size */ );

/* save a block of memory on the save stack */
void savemem( DataAddress, IntegerType /* size */);

/* restore a block of memory from the save stack */
void unsavemem( DataAddress, IntegerType /* size */);
```

datamem.c

```
/* datamem.c: Data Memory Module */

#include "datamem.h"
#include "fault.h"

/* Array used to hold sizeof(data types) constants so they can
   be easily retrieved. */

int sizedata[LASTTYPE] = {
  sizeof(BooleanType),
  sizeof(IntegerType),
  sizeof(RealType),
```

Figure 28 (continued)

```
  sizeof(AddressType),
  sizeof(CharacterType)
};

/* MAXDATA should be a multiple of WORDSIZE so that the defined data
   structure will be big enough. */
#define MAXDATA (DataAddress)8192

/* Size of the free memory list structure. */
#define MAXFREEMEM 1024

/* Free memory list structure. */
struct {
  DataAddress lower,upper;
} freemem[MAXFREEMEM] = {0, MAXDATA - 1};
int lastentry = 0;


/* Data Memory structure. */
DataWord datamem[MAXDATA];

/* Data Defined structure. */
DataWord datadefined[MAXDATA/WORDSIZE] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

/* A procedure to provide sizeof(data types) to other modules. */
int datasize( DataType type ) {

  return(sizedata[type]);
}

/* Store data and mark it as defined. */
void writedata( DataType type, DataAddress address, DataValue data ) {

  void *dataptr;
  DoubleWord tempdef;

  if (address >= MAXDATA) {
    fault(BADDATAADDR);
  }
  else {
    (char *)dataptr = &(datamem[address]);
    switch(type) {

      case INTEGER:
        *((IntegerType *)dataptr) = data.integer;
        break;

      case BOOLEAN:
        *((BooleanType *)dataptr) = data.boolean;
        break;

      case REAL:
        *((RealType *)dataptr) = data.real;
        break;

      case ADDRESS:
        *((AddressType *)dataptr) = data.address;
        break;
```

Figure 28 (continued)

```
      case CHARACTER:
        *((CharacterType *)dataptr) = data.character;
        break;
    }
    /* This part marks all memory locations occupied by the data
       as defined. */
    tempdef = (DoubleWord)(HIGHDBLWORD << (DBLWORDSIZE - sizedata[type]))
>>
              (address % WORDSIZE);
    datadefined[(address/WORDSIZE) + 1] |= (DataWord)tempdef;
    datadefined[address/WORDSIZE] |= (DataWord)(tempdef >> WORDSIZE);
  }
}

/* Read data at a particular location.  All memory locations
   occupied by the data must be marked as defined otherwise
   the BooleanType will be marked as False and the contents
   of data is unknown. */
void readdata( DataType type, DataAddress address, DataValue *data,
               DataWord *defined ) {

  void *dataptr;
  int base, offset;
  DoubleWord def;

  if ((address >= MAXDATA) || (address < 0))
    { /* ERROR: bad address. */
    fault(BADDATAADDR);
  }
  else {
    base = address/WORDSIZE;
    offset = address % WORDSIZE;

    def = (DoubleWord)((DoubleWord)datadefined[base] << WORDSIZE |
             datadefined[base + 1]) >>
             (DBLWORDSIZE - offset - sizedata[type]);
    *defined  =  (DataWord)def  &  ((DataWord)HIGHWORD  >>  (WORDSIZE  -
sizedata[type]));
    (char *)dataptr = &(datamem[address]);
    switch(type) {

        case INTEGER:
          data->integer = *((IntegerType *)dataptr);
          break;

        case BOOLEAN:
          data->boolean = *((BooleanType *)dataptr);
          break;

        case REAL:
          data->real = *((RealType *)dataptr);
          break;

        case ADDRESS:
          data->address = *((AddressType *)dataptr);
          break;

        case CHARACTER:
          data->character = *((CharacterType *)dataptr);
          break;
```

Figure 28 (continued)

```
    }
  }
}


/* Allocate a block of data memory according to the free memory
   list or requested address. */
void allocdata( DataAddress *address, IntegerType size ) {

  int i,index;

  index = 0;
  if (*address == NULLDATAADDR) {
    /* Allocate any data memory block big enough. */
    while((index <= lastentry) && (freemem[index].upper
           - freemem[index].lower + 1 < size)) {
      index++;
    }
    /* If past the end of the free memory list, there is no
       block of memory large enough. */
    if (index > lastentry) {
      fault(OUTMEM); *address = NULLDATAADDR;
    }
    else {
      *address = freemem[index].lower;
      freemem[index].lower += size;
      if (freemem[index].lower == freemem[index].upper) {
        /* Adjust the free list if this block is now empty. */
        do {
          index++;
          freemem[index-1] = freemem[index];
        } while (index < lastentry);
        lastentry--;
      }
    }
  }
  else {
    /* Allocate a specific block in data memory. */
    if (*address + size > MAXDATA) size = MAXDATA - *address - 1;

    index = 0;
    while ((index <= lastentry) && (freemem[index].lower <= *address)) {
      index++;
    }
    index--;

    if    ((*address    <    freemem[index].lower)    ||    (*address    >
freemem[index].upper)
        || (freemem[index].upper - *address + 1 < size)) {
      /* attempted to allocate a specific area of memory, but
         at least part of it is already allocated. */
      fault(MEMALLOC); *address = NULLDATAADDR;
    }
    else {
      /* Specific block can be allocated, so adjust free table. */
      if (*address == freemem[index].lower) {
        /* Allocated block starts at the beginning of the free block,
           so the lower bound can be adjusted. */
        freemem[index].lower += size - 1;
      }
```

Figure 28 (continued)

```
      else if (*address == freemem[index].upper) {
        /* Allocated block ends at the end of the free block, so the
           upper bound can be adjusted. */
        freemem[index].upper = *address - 1;
      }
      else {
        /* Allocated block is in the middle of the free block, so
           the free table must be expanded to split this free block. */
      for (i = lastentry; i >= index; i--) {
          freemem[i + 1] = freemem[i];
      }
      lastentry++;
        freemem[index].upper = *address;
        freemem[index + 1].lower = *address + size;
      }
    }
  }
}


/* Make some previously allocated block available for allocation
   again. */
void unallocdata( DataAddress address, IntegerType size ) {

  int index, i;

  index = 0;
  /* Find where this block fits in the free memory list. */
  while ((index <= lastentry) && (freemem[index].lower <= address)) {
    index++;
  }
  if (index > lastentry) {
    lastentry++;
    freemem[lastentry].lower = address;
    freemem[lastentry].upper = address + size - 1;
  }
  else {
    if (address + size == freemem[index].lower) {
      freemem[index].lower = address;
    }
    else if (freemem[index-1].upper + 1 == address) {
      freemem[index-1].upper = address + size - 1;
    }
    else {
      for (i = lastentry; i >= index; i--) {
        freemem[i+1] = freemem[i];
      }
      lastentry++;
      freemem[index].lower = address;
      freemem[index].upper = address + size - 1;
    }
  }
}


void savemem( DataAddress address, IntegerType size) {

  DataWord def;
  int i, base, offset;
```

Figure 28 (continued)

```
  for (i = 0; i < size; i++) {
    base = (address + i)/WORDSIZE;
    offset = (address + i)%WORDSIZE;
    def = (1 << (WORDSIZE - offset)) & (datamem[base]);
    pushsave(&datamem[address+i], def, 1);
  }

/*  offset = address % WORDSIZE;*/

/*  /* Clear all bits except those that represent the data */
/*  def = datadefined[address/WORDSIZE] << offset >> offset;*/

  /* push first partial DATAWORD onto save stack */
/*  pushsave(&datamem[address], (DoubleWord)def, WORDSIZE - offset);*/

  /* push all full DATAWORD's onto save stack */
/*  for (i = 0; i < (size - WORDSIZE + offset)/WORDSIZE; i++) {
    pushsave(&datamem[address + i * 8],
          (DoubleWord)datadefined[address + i], WORDSIZE);
  }     */

  /* push last partial DATAWORD onto save stack */
/*  offset = (size - WORDSIZE + offset) % WORDSIZE;*/
/*  def = datadefined[address + i] >> (WORDSIZE - offset);*/
/*  pushsave(datamem[address + i * 8], def, offset);*/
}


void unsavemem( DataAddress address, IntegerType size ) {

  int tempsize, i, offset;
  DataWord def;

/*  tempsize = (address + size) % WORDSIZE;

  popsave(&datamem[address + size - tempsize], &def, tempsize);
  datadefined[(address + size - 1)/WORDSIZE] |= def <<
           (WORDSIZE - tempsize);

  for (i = (size - tempsize)/WORDSIZE; i >= 0; i--) {
    popsave(&datamem[address + i * WORDSIZE],
          &datadefined[(address/WORDSIZE) + i], WORDSIZE);
  }

  popsave(datamem[address], datadefined[address], WORDSIZE -
        (address % WORDSIZE)); */

  for (i = size - 1; i >= 0; i--) {
    popsave(&datamem[address + i], &def, 1);
    datadefined[(address +i)/WORDSIZE] |= def << (WORDSIZE - (address +
i)%WORDSIZE);
  }
}
```

variable.h

```
/* variable.h: variable register header */

/* This allows for 64K variable registers on on IBM PC.  The actual number
   of allowable registers is smaller than this, since only 64K bytes can
```

Figure 28 (continued)

```
   be used to store the set of registers. */
typedef IntegerType VariableReg;

/* getvaraddress returns the data address of the current instance of
   VariableReg. */
void getvaraddress( VariableReg, DataAddress * );

/* getvarsize returns the size of VariableReg. */
void getvarsize( VariableReg, IntegerType * );

/* pushvariable pushes a new data address onto the VarRegister stack. */
void pushvariable( VariableReg, DataAddress );

/* popvariable removes the top data address from the VarRegister stack. */
void popvariable( VariableReg, DataAddress * );
```

variable.c

```
/* variable.c: Variable Register/Stack Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "variable.h"
#include "fault.h"
#include <alloc.h>

/* Variable Stack structure. */
typedef struct vs {
  DataAddress address;
  struct vs *next;
} VarStack;

/* Variable Register Table. */
struct VR {
  IntegerType size;
  VarStack *stack;
};

static unsigned int lastreg = 0;

struct VR *varregs;

/* Returns current data memory address of variable. */
void getvaraddress( VariableReg varreg, DataAddress *address ) {

  if (varreg > lastreg) {
    fault(BADVARREG);
  }
  else {
    if (varregs[varreg].stack == NULL) {
      *address = NULLDATAADDR;
    }
    else {
      *address = varregs[varreg].stack->address;
    }
  }
}

/* Returns size of variable. */
```

Figure 28 (continued)

```
void getvarsize( VariableReg varreg, IntegerType *size ) {

  if (varreg > lastreg) {
    fault(BADVARREG);
  }
  else {
      *size = varregs[varreg].size;
  }
}

void pushvariable( VariableReg varreg, DataAddress address ) {

  VarStack *temp;

  if (varreg > lastreg) {
    fault(BADVARREG);
  }
  else {
    temp = malloc(sizeof(VarStack));
    if (temp == NULL) {
      fault(OUTMEM);
    }
    else {
      temp->address = address;
      temp->next = varregs[varreg].stack;
      varregs[varreg].stack = temp;
    }
  }
}

void popvariable( VariableReg varreg, DataAddress *address ) {

  VarStack *temp;

  if (varreg > lastreg) {
    fault(BADVARREG);
  }
  else {
    if (varregs[varreg].stack == NULL) {
      fault(VARNOTALLOC);
    }
    else {
      *address = varregs[varreg].stack->address;
      temp = varregs[varreg].stack;
      varregs[varreg].stack = varregs[varreg].stack->next;
      free(temp);
    }
  }
}


/* Load variable sizes from a file. */
void loadvarregs( int fileid ) {

  IntegerType size;
  VariableReg i;

  if (sizeof(size) != read(fileid, &size, sizeof(size))) {
    /* ERROR: bad file format. */
    fault(BADFILE);
```

Figure 28 (continued)

```
  }
  else {
    varregs = calloc(size, sizeof(Instruction));
    if (varregs == NULL) {
      /* Not enough system memory left. */
      fault(OUTMEM);
    }
    else {
      i = 0;
      while ((getfault() == NOFAULT) && (i < size)) {
      if (read(fileid, &(varregs[i].size), sizeof(IntegerType)) !=
          sizeof(IntegerType)) fault(BADFILE);
      }
      lastreg = size - 1;
    }
  }
}
```

label.h

```
/* label.h: Label Register/Stack header */

/* This allows for 64K variable registers on on IBM PC.  The actual number
   of allowable registers is smaller than this, since only 64K bytes can
   be used to store the set of registers. */
typedef IntegerType LabelReg;

/* getlabeladdress returns the program address at which Label resides. */
void getlabeladdress( LabelReg, ProgAddress * );

/* pushlabel pushes a program address onto the Label stack. */
void pushlabel( LabelReg, ProgAddress );

/* poplabel returns the top program address from the Label stack. */
void poplabel( LabelReg, ProgAddress * );
```

label.c

```
/* label.c: Label Register/Stack Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"
#include "label.h"
#include <alloc.h>

/* Label Stack structure. */
typedef struct ls {
  ProgAddress address;
  unsigned count;
  struct ls *next;
} LabelStack;

/* Label Register Table. */
struct LR {
  ProgAddress address;
  LabelStack *stack;
};
static unsigned int lastreg = 0;
```

Figure 28 (continued)

```
struct LR *labelregs;

/* Get the program address of the label instruction */
void getlabeladdress( LabelReg labelreg, ProgAddress *address ) {

  if ((labelreg > lastreg) || (labelreg < 0)) {
    /* ERROR: bad label register number. */
    fault(BADLABELREG);
  }
  else {
    *address = labelregs[labelreg].address;
  }
}

/* Save the address from which the label was reached. */
void pushlabel(LabelReg labelreg, ProgAddress address) {
  LabelStack *temp;

  if ((labelreg > lastreg) || (labelreg < 0)) {
    /* ERROR: bad label register number. */
    fault(BADLABELREG);
  }
  else {
    if ((labelregs[labelreg].stack == NULL) ||
      (labelregs[labelreg].stack->address != address)) {
      temp = malloc(sizeof(LabelStack));
      if (temp == NULL) {
      fault(OUTMEM);
      }
      else {
      temp->address = address;
      temp->count = 1;
      temp->next = labelregs[labelreg].stack;
      labelregs[labelreg].stack = temp;
      }
    }
    else {
      labelregs[labelreg].stack->count++;
    }
  }
}

/* Return the address from which a label was reached. */
void poplabel(LabelReg labelreg, ProgAddress *address) {

  LabelStack *temp;

  if ((labelreg > lastreg) || (labelreg < 0)) {
    /* ERROR: bad label register number. */
    fault(BADLABELREG);
  }
  else {
    *address = labelregs[labelreg].stack->address;
    labelregs[labelreg].stack->count--;
    if (labelregs[labelreg].stack->count == 0) {
      temp = labelregs[labelreg].stack;
      labelregs[labelreg].stack = labelregs[labelreg].stack->next;
      free(temp);
    }
  }
```

Figure 28 (continued)

```
}


/* Load label addresses from a file. */
void loadlabelregs( int fileid ) {

  IntegerType size;
  LabelReg i;

  if (sizeof(size) != read(fileid, &size, sizeof(size))) {
    /* ERROR: bad file format. */
    fault(BADFILE);
  }
  else {
    labelregs = calloc(size, sizeof(Instruction));
    if (labelregs == NULL) {
      /* Not enough system memory left. */
      fault(OUTMEM);
    }
    else {
      i = 0;
      while ((getfault() == NOFAULT) && (i < size)) {
      if (read(fileid, &(labelregs[i].address), sizeof(ProgAddress)) !=
          sizeof(ProgAddress)) fault(BADFILE);
      }
      lastreg = size - 1;
    }
  }
}
```

eval.h

```
/* eval.h: Evaluation Stack Header */

/* pusheval places DataValue on top of the evaluation stack */
extern void pusheval( DataValue );

/* popeval returns the top DataValue from the evaluation stack. */
extern void popeval( DataValue * );
```

eval.c

```
/* eval.c: Evaluation Stack Module */

#include "datamem.h"
#include "eval.h"
#include "fault.h"

/* Evaluation stack size. */
#define MAXEVAL 512

/* Evaluation Stack structure. */
DataValue evalstack[MAXEVAL];
static int evaltop = 0;

void pusheval( DataValue data ) {

  if (evaltop >= MAXEVAL) {
    fault(EVALOVRFLW);
  }
```

Figure 28 (continued)

```
  else {
    evalstack[evaltop++] = data;
  }
}

void popeval( DataValue *data ) {

  if (evaltop == 0) {
    fault(EVALEMPTY);
  }
  else {
    *data=evalstack[--evaltop];
  }
}
```

call.h

```
/* call.h: call stack header */

/* pushcall places a program address onto the call stack. */
extern void pushcall( ProgAddress );

/* popcall returns the top program address on the call stack. */
extern void popcall( ProgAddress * );
```

call.c

```
/* call.c: Call Stack Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "fault.h"

#define MAXCALL 1024

/* Call Stack structure. */
ProgAddress callstack[MAXCALL];
static unsigned int calltop = 0;

void pushcall( ProgAddress address ) {

  if (calltop >= MAXCALL) {
    fault(CALLOVRFLW);
  }
  else {
    callstack[calltop++] = address;
  }
}

void popcall( ProgAddress *address ) {

  if (calltop == 0) {
    fault(CALLEMPTY);
  }
  else {
    *address = callstack[--calltop];
  }
}
```

Figure 28 (continued)

save.h

```
/* save.h: save stack header */

/* savedata pushes DataValue onto the save stack. */
void savedata( DataValue, DataType, DataWord /* defined */ );

/* unsavedata returns the top DataValue from the top of the
   save stack. */
void unsavedata( DataValue *, DataType, DataWord * );
```

save.c

```
/* save.c: Save Stack Module */

#include "datamem.h"
#include "fault.h"
#include "flags.h"
#include "save.h"

/* Save Stack and Packet Queue size */
#define MAXSAVE 4096
#define MAXPACK 1024

/* Save Stack structure */
DataWord savestack[MAXSAVE];
static unsigned int savetop = 0, savebottom = 0;

/* Save Stack defined bits structure */
DataWord savedefined[MAXSAVE >> 3];

/* Packet Queue structure */
unsigned int packetqueue[MAXPACK];
static unsigned int packettop = 0, packetbottom= 0;


void pushsave( DataWord *data, DoubleWord defined,
               int size ) {


  int base, offset, i;
  DoubleWord def;
  BooleanType ok;

  base = savetop/WORDSIZE;
  offset = savetop % WORDSIZE;

  /* Set up data defined bits to match save stack
     defined bit positions. */
  def = defined << (DBLWORDSIZE - offset);

  /* Clear save stack defined bits so they can be properly
     set when ored with the data defined bits. */
  savedefined[base] &= (DataWord)((def ^ HIGHDBLWORD)
                       >> WORDSIZE);
  savedefined[base + 1] &= (DataWord)(def ^ HIGHDBLWORD);

  /* Or the data defined bits with the cleared save stack
     defined bits so the defined status of the data is
     also save on the save stack. */
```

Figure 28 (continued)

```
  savedefined[base] |= (DataWord)(def >> WORDSIZE);
  savedefined[base + 1] |= (DataWord) def;

  /* Now push the actual data onto the save stack. */
  i = 0;
  ok = TRUE;
  while (i < size && ok == TRUE) {
    savestack[savetop++] = data[i];
    savetop %= MAXSAVE;
    i++;
    if (savebottom == savetop) {
      if (packettop == packetbottom) {
        ok = FALSE;
        savetop = 0;
        savebottom = 0;
      }
      else {
        savebottom = packetqueue[packetbottom++];
        packetbottom %= MAXPACK;
      }
    }
  }
}


void savedata( DataValue data, DataType type, DataWord defined ) {

  defined >>= WORDSIZE - datasize(type);
  pushsave((DataWord *)&data, (DoubleWord)defined,
           datasize(type));
}


void popsave( DataWord *data, DataWord *defined, int size ) {

  int base, offset, i;


  base = (savetop - size)/WORDSIZE;
  offset = (savetop - size)%WORDSIZE;

  *defined = ((savedefined[base] << WORDSIZE) | savedefined[base+1])
            >> (WORDSIZE - offset - size);

  for (i = size-1; i >= 0; i--) {
    data[i] = savestack[--savetop];
    savetop %= MAXSAVE;
  }
}


void unsavedata( DataValue *data, DataType type, DataWord *defined ) {

     popsave((DataWord *)data, defined, datasize(type));
}


void newpacket() {

  packettop++;
```

Figure 28 (continued)

```
  packettop %= MAXPACK;
  if (packettop == packetbottom) {
    savebottom = packetqueue[packetbottom];
    savebottom++;
    savebottom %= MAXPACK;
  }
  packetqueue[packettop] = savetop;
}
```

pachet.h

```
/* packet.h: Packet Module header */

typedef struct {
  ProgAddress startaddr, endaddr;
  int startline, startcol, endline, endcol;
  SymbolEntry scope;
} Packet;

typedef unsigned int PacketNum;

void findpacket(ProgAddress, Packet *, PacketNum *);

void getpacket(PacketNum, Packet *);
```

packet.c

```
/* packet.c: Packet Module */

#include "datamem.h"
#include "decex.h"
#include "progmem.h"
#include "variable.h"
#include "symbol.h"
#include "packet.h"

/* Packet table structure */
Packet *packettable;

PacketNum lastpacket = 0;

void findpacket( ProgAddress address, Packet *pack,
                 PacketNum *num ) {

  int top, bottom;

  top = lastpacket;
  *num = lastpacket;
  bottom = 0;
  while (top > bottom) {
    *num = (top + bottom)/2;
    if (address > packettable[*num].endaddr) {
      bottom = *num + 1;
    }
    else {
      top = *num;
    }
  }
  if (address >= packettable[top].startaddr &&
      address <= packettable[top].endaddr) {
```

Figure 28 (continued)

```
    *pack = packettable[top];
    *num = top;
  }
  else {
    pack->startaddr = NULLPROGADDR;
    *num = -1;
  }
}


void getpacket( PacketNum num, Packet *pack ) {

  if (num > lastpacket || num < 0) {
    pack->startaddr = NULLPROGADDR;
  }
  else {
    *pack = packettable[num];
  }
}
```

fault.h

```
/* fault.h: fault handler header */

typedef enum {
  NOFAULT, EVALOVRFLW, EVALEMPTY, CALLOVRFLW, CALLEMPTY, ILLEGALINSTR,
  ILLEGALTYPE, BADVARREG, BADLABELREG, BADPROGADDR, BADDATAADDR,
  BADFILE, OUTMEM, VARNOTALLOC, ILLEGALFLAG, SAVEUNDRFLW,
  UNDEFDATA, ILLEGALMODE, DIVBYZERO, MEMALLOC, LASTFAULT
} FaultType;

void fault( FaultType );

void faultmsg( FaultType, char * );

FaultType getfault( void );
```

fault.c

```
/* fault.c: Fault Handler Module */

#include "fault.h"
#include <stdlib.h>

/* Holds the last fault that occurred. */
FaultType lastfault;

/* Messages associated with faults. */
static char *faultmsgs[LASTFAULT] = {
  "No fault.",
  "Evaluation stack overflow.",
  "Evaluation stack underflow.",
  "Call stack overflow.",
  "Call stack underflow.",
  "Illegal instruction.",
  "Illegal data type for instruction.",
  "Bad variable register.",
  "Bad label register.",
  "Bad program address.",
  "Bad data address.",
  "Error in input file.",
```

Figure 28 (continued)

```
  "Out of data memory.",
  "Variable not allocated.",
  "Save stack underflow, no more backup allowed."
  "Undefined data accessed.",
  "Illegal address mode for opcode.",
  "Divide by zero.",
  "Attempted to allocate already allocated memory."
};

/* Register fault occurrence. */
void fault( FaultType faultnum ) {

  lastfault = faultnum;
}

/* Return pointer to message explaining fault. */
void faultmsg( FaultType faultnum, char *msg ) {

  if (faultnum >= LASTFAULT) {
    msg = NULL;
  }
  else {
    msg = faultmsgs[faultnum];
  }
}

FaultType getfault() {

  return(lastfault);
}
```

flags.h

```
/* flags.h: Flags Module header */

/* These are the various flags used in the E-machine. */
typedef enum {
  SAVEEMPTY, FAULT, FORWARD, BRANCH, LASTFLAG
} FlagType;


void setflag( FlagType );

void resetflag( FlagType );

BooleanType getflag( FlagType );
```

flags.c

```
/* flags.c: Machine Flag Module */

#include "datamem.h"
#include "flags.h"
#include "fault.h"

/* Flag Set structure. */
static BooleanType flags[LASTFLAG];

/* Set a flag value to TRUE. */
void setflag( FlagType flag ) {
```

Figure 28 (continued)

```
  if (flag >= LASTFLAG) {
    fault(ILLEGALFLAG);
  }
  else {
    flags[flag] = TRUE;
  }
}

/* Set a flag value to FALSE. */
void resetflag( FlagType flag ) {

  if (flag >= LASTFLAG) {
    fault(ILLEGALFLAG);
  }
  else {
    flags[flag] = FALSE;
  }
}


/* Return the value of a flag. */
BooleanType getflag( FlagType flag ) {

  if (flag >= LASTFLAG) {
    fault(ILLEGALFLAG); return(FALSE);
  }
  else {
    return(flags[flag]);
  }
}
```

symbol.h

```
/* symbol.h: Symbol Table header */

typedef unsigned int SymbolEntry;

enum { PROCEDURE = 10, HEADER, END, RECORD };

extern struct {
  char *name;
  IntegerType upperbound, lowerbound;
  IntegerType offset;
  SymbolEntry parent, child;
  VariableReg varreg;
} *symboltable;

extern SymbolEntry lastsym;
```

symbol.c

```
/* symbol.c: symbol table load */
#include "sybmol.h"

typedef unsigned int SymbolEntry;

struct ST {
  char *name;
  IntegerType upperbound, lowerbound;
  IntegerType offset;
  int type;
  SymbolEntry parent, child;
```

Figure 28 (continued)

```
  VariableReg varreg;
}  *symboltable;

SymbolEntry lastsym;

/* Load variable sizes from a file. */
void loadsymboltable( int fileid ) {

  IntegerType size;
  VariableReg i;

  if (sizeof(size) != read(fileid, &size, sizeof(size))) {
    /* ERROR: bad file format. */
    fault(BADFILE);
  }
  else {
    symboltable = calloc(size, sizeof(ST));
    if (symboltable == NULL) {
      /* Not enough system memory left. */
      fault(OUTMEM);
    }
    else {
      i = 0;
      while ((getfault() == NOFAULT) && (i < size)) {
      if (read(fileid, &(symboltable[i]), sizeof(ST)) !=
          sizeof(ST)) fault(BADFILE);
      }
      lastsym = size - 1;
    }
  }
}
```