# 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


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.

# 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

APPROVAL

of a thesis submitted by

Michael Leigh Birch

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

6/12/90
Date

Rockford J. Ross
Chairperson, Graduate Committee

Approved for the Major Department

June 12, 1990
Date

J. Rudolph Starling
Head, Major Department

Approved for the College of Graduate Studies

June 13, 1990
Date

Henry L. Parsons
Graduate Dean

## 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 Michael J. Bird

Date 6/13/90

## 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.

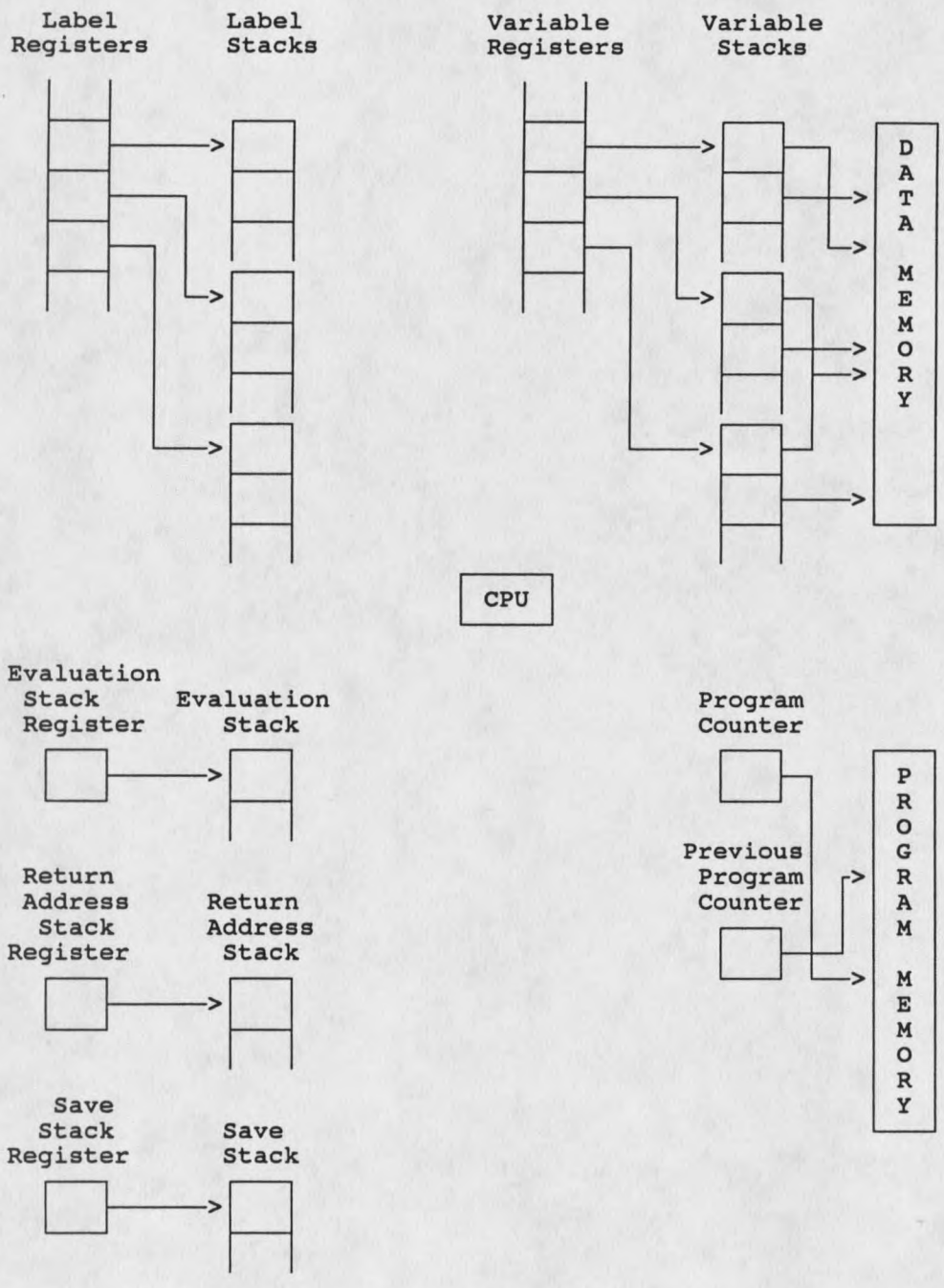


Figure 1
The E-machine

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.

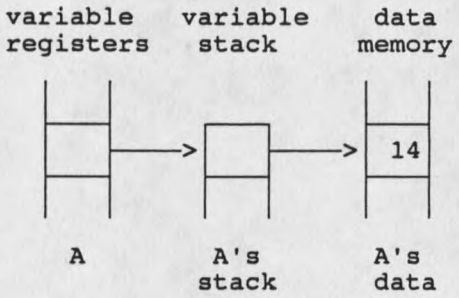


Figure 2
E-machine Global Variable Implementation

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