

Implementing Prolog via Microprogramming a General Purpose Host Computer

Jeff Gee

Master's Report Plan II
Professor Yale Patt
Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

This report documents the implementation of a high performance Prolog system achieved by remicroprogramming a host general purpose computer. New microcode was added to a VAX 8600 computer to implement the Berkeley Programmed Logic Machine (PLM), a Prolog-specific architecture closely related to the Warren Abstract Machine. The mapping of the abstract resources of the PLM to the 8600 is described. Performance comparisons between this system and three other Prolog implementations are included. On average, this system performs three times better than compiled and twenty times better than interpreted systems available on the same hardware. In addition, this execution model provides 75% of the performance of the special purpose PLM coprocessor, after results are normalized to the cycle time of each machine.

December 14, 1987



Table of Contents

Chapter 1: Project Overview	
1. Introduction	1
2. Background	1
3. Project Goals	2
4. Outline of the Report	2
Chapter 2: Prolog Execution Models	
1. Introduction	4
2. The C-Prolog Interpreter	4
3. BIM Prolog	4
4. The PLM Special Purpose Coprocessor	5
5. Direct Execution of the PLM in VAX 8600 Microcode	6
Chapter 3: Details of the PLM, a Modified WAM	
1. Introduction	8
2. Data Types	8
3. Registers	8
4. Data Memory Allocation	9
5. Instruction Set	10
Chapter 4: Operating Environment	
1. 8600 System Architecture	12
2. 8600 Microarchitecture	13
3. Microprogramming Environment	14
4. Compilation and Assembly of Prolog Programs	14
Chapter 5: Implementation of the PLM Architecture on the 8600	
1. Data Representation	19
2. Register Allocation	20
3. Memory Allocation	20
4. Process Control	21
5. Implementation of the PLM Instruction Set	22
6. Implementation of the Prolog Built-in Predicates	23
6.1 New Instructions	23
6.2 Built-ins in C	23
6.3 Built-ins Emulated in Prolog	24
7. Implementation of Basic Prolog Routines	25
8. Sample Compilation Process	25
Chapter 6: Measurements and Analysis	
1. Measurement Philosophy	32
2. Performance Measurements	32

2.1 Implementations on General Purpose Computers	33
2.2 General vs. Special Purpose Implementations	34
Chapter 7: Conclusions	
1. Report Summary	37
2. Meeting the Project Goals	38
Acknowledgements	38
References	40
Appendices	
Appendix A: Microcode Listing	41
Appendix B: Instruction Formats	54
Appendix C: Source Code for Utility Routines	58

List of Illustrations

Figure 2.1: Prolog Execution Models	7
Figure 4.1: VAX 8600 Block Diagram	16
Figure 4.2: EBOX Data Path	17
Figure 4.3: Compilation and Assembly Process	18
Figure 5.1: Data Representation	28
Figure 5.2: Register Allocation	29
Figure 5.3: Memory Allocation	30
Figure 5.4: Sample Instruction Formats	31



Chapter 1

Project Overview

1. Introduction

The purpose of this project is to develop a high performance implementation of Prolog on a VAX¹ 8600 general purpose computer by emulating in microcode an architecture designed to support Prolog. The architecture is the Berkeley Programmed Logic Machine (PLM), developed by Tep Dobry and Barry Fagin [2,3]. The PLM is heavily influenced by the Warren Abstract Machine (WAM), conceived by David Warren [7]. New microcode was written for the VAX 8600 which directly interprets the instructions defined by the PLM abstract architecture. Performance results indicate that this system provides the fastest Prolog implementation available on the VAX 8600.

2. Background

The focus of the Aquarius project at Berkeley is to achieve large improvements in the execution speed of applications requiring intensive numerical calculation and substantial symbolic manipulation. The primary language of the system is the logic programming language Prolog. Prolog has gained wide acceptance as the language of choice for knowledge processing and expert systems.

Various execution models for the high performance execution of Prolog have been investigated. Each of these models compile a Prolog program into an architecture related to the Warren Abstract Machine as the first step. The WAM, analogous to p-code in Pascal, is the conventional intermediate form of Prolog. The WAM architecture consists of machine registers, four memory spaces, data structures, and forty instructions which carry out the semantics of the Prolog language.

After the Prolog program is compiled to WAM level instructions, the different execution models proceed in separate ways. Each WAM instruction can be interpreted by software written in the machine instructions of a general purpose computer, further compiled down to the machine architecture of a general purpose computer, or directly executed in the microcode of a special purpose Prolog coprocessor designed explicitly for the WAM.

¹ VAX is a trademark of the Digital Equipment Corporation.

To date, none of the above execution models provides efficient symbolic and numeric calculation. In a general purpose machine, a semantic gap exists between the symbolic operations of Prolog and the machine instructions available in the computer. A special purpose coprocessor normally does not contain the special hardware required for fast numeric calculation, although the ability to provide coprocessors for Prolog and numerics, which are in fact closely coupled, remains an important part of the Aquarius project.

The approach taken in the work reported here is to provide high performance symbolic and numeric execution in the same general purpose processor. A Prolog program is first compiled into the PLM architecture, a modified version of the Warren Abstract Machine. The semantic gap due to the host machine code level is eliminated by introducing microcode which interprets directly the PLM instruction set. Basic Prolog operations are provided at the microcode level, while fast numeric computations are provided by the native instruction set and hardware of the host. The VAX 8600 general purpose processor was chosen as the implementation vehicle. The resulting performance exceeds all other known VAX implementations and approaches the speed of existing special purpose coprocessors.

3. Project Goals

The major goals of this project are presented below. The remaining sections of this report document how these goals were met.

- (1) The PLM instruction set is to be translated into a form directly executable by the VAX 8600.
- (2) The new instructions added to the VAX 8600 must function in a multiprogramming environment. That is, a Prolog process must be interruptable and restartable.
- (3) The native VAX architecture must be preserved. Any software written in the VAX instruction set must execute correctly in this system.
- (4) The system should be the fastest Prolog implementation available on the VAX architecture.

4. Outline of the Report

This report is divided into seven chapters. Chapter 2 describes four different execution models for Prolog. Chapter 3 describes the PLM architecture. Chapter 4 describes the execution environment in the

VAX 8600. Chapter 5 discusses the implementation of the PLM architecture in the VAX 8600. Chapter 6 contains performance results and compares these measurements with what has been obtained with the other models discussed in Chapter 2. Finally, Chapter 7 offers some concluding remarks. In addition, several appendices are included which contain the new microcode introduced to the system, instruction formats for each PLM construct and other new instructions added to the VAX 8600, and source code for the utilities used in this project.

Chapter 2

Prolog Execution Models

1. Introduction

This chapter describes four uniprocessor Prolog systems, each of which represents a different execution model for Prolog. Three of these systems execute on the VAX 8600 general purpose computer; the fourth implements the PLM in the hardware of a special purpose Prolog coprocessor. The execution models for these systems are shown in figure 2.1.

2. The C-Prolog Interpreter

In the C-Prolog interpreter, a Prolog source program is first **translated** into an intermediate form, a structure-based representation of the original Prolog code. Two levels of interpretation are then employed. The intermediate form is interpreted by a program written in VAX machine language instructions, and each VAX instruction is in turn interpreted by the microinstructions and datapath of the VAX 8600.

There are several performance disadvantages to this approach. First there is the overhead required to evaluate the internal structure-based form and branch to the appropriate machine language routine. A more important problem is the semantic gap between the general purpose machine language instructions which form the interpreter and the basic symbolic operations of Prolog. Any high performance Prolog implementation must rapidly determine and branch on the state of a few select bits in the data word. This ability is not normally provided in the instruction set of a general purpose machine. Due to the above limitations, the performance of this model is expected to be low.

3. BIM Prolog

BIM Prolog compiles a Prolog program into machine language instructions of the VAX architecture. The Prolog program is first compiled into the instruction set of an abstract architecture closely related to the Warren Abstract Machine. Each abstract instruction is then macro-expanded into a sequence of VAX instructions. There is a single level of interpretation in BIM Prolog. Each VAX machine instruction in the new problem specification is interpreted by the hardware and microcode of the VAX 8600.

In contrast to interpreted Prolog, no decoding of an internal representation is necessary. The translation process continues through to the host ISA level. In the C-Prolog interpreter the translation from the internal form to machine code is done dynamically at runtime, slowing the execution process considerably. A study suggests the performance of a compiled system may be an order of magnitude greater than an interpreted implementation [8]. However, performance is still degraded by the semantic gap between the host machine code level and the primitive operations required of Prolog.

4. The PLM Special Purpose Coprocessor

The Berkeley PLM is a special purpose coprocessor which implements a variant of the WAM in hardware. This special purpose coprocessor directly executes the PLM version of a Prolog program. No further translation or compilation is required. The datapath of the special purpose coprocessor is optimized for PLM instructions and the basic symbolic operations of Prolog. In particular, support is provided for data tag test and manipulation. Processing of PLM instructions is expected to be optimal, due to the tailored hardware.

Certain Prolog built-in predicates require operations which are not supported by the PLM instruction set. For example, the `is` predicate requires complicated numeric functions such as multiplication, division, and modulo. The current version of the PLM does not have the hardware support necessary to perform these operations efficiently. Instead, the PLM would normally request a host general purpose machine (via the escape mechanism [2]) to perform these computations. The host processor retrieves the operands, performs the computation, and transfers the result back to the Prolog coprocessor. Performance is reduced due to the overhead required to transfer data between the host and coprocessor.

However, one should point out that the performance loss due to these external computations is a result of earlier work and not an inherent property of the special purpose coprocessor. The PLM and other existing Prolog coprocessors are only the first iterations of designs which are still evolving. The state of the art for specialized coprocessors is still young, and lessons have been learned from these initial implementations. Future designs will reduce or eliminate the overhead incurred with the escape mechanism, probably by adding hardware support to execute most built-in predicates internally.

5. Direct Execution of the PLM in VAX 8600 Microcode

Current implementations of the first three computation models possess shortcomings for either symbolic or numeric processing. This section describes an execution model for Prolog which supports both types of computation efficiently.

In this system, the instruction set of the VAX 8600 general purpose microprogrammable computer is extended via the addition of new microcode which executes the PLM instruction set. The VAX 8600 executes the PLM instructions as if they were part of its native architecture.

Symbolic operations of the PLM are implemented at the low level of the horizontal microcode and raw datapath of the host. By doing so, the semantic gap between these operations and the higher ISA level of the host is eliminated. Numeric operations are performed with general purpose machine instructions present in-line with the new PLM instructions, removing the time consuming escape mechanism associated with current special purpose coprocessors. Execution speed on the PLM constructs is expected to fall between the coprocessor and compiled models for Prolog. Performance on applications with significant amounts of numeric computation may even surpass existing special purpose coprocessors due to the in-line execution of numeric built-ins.

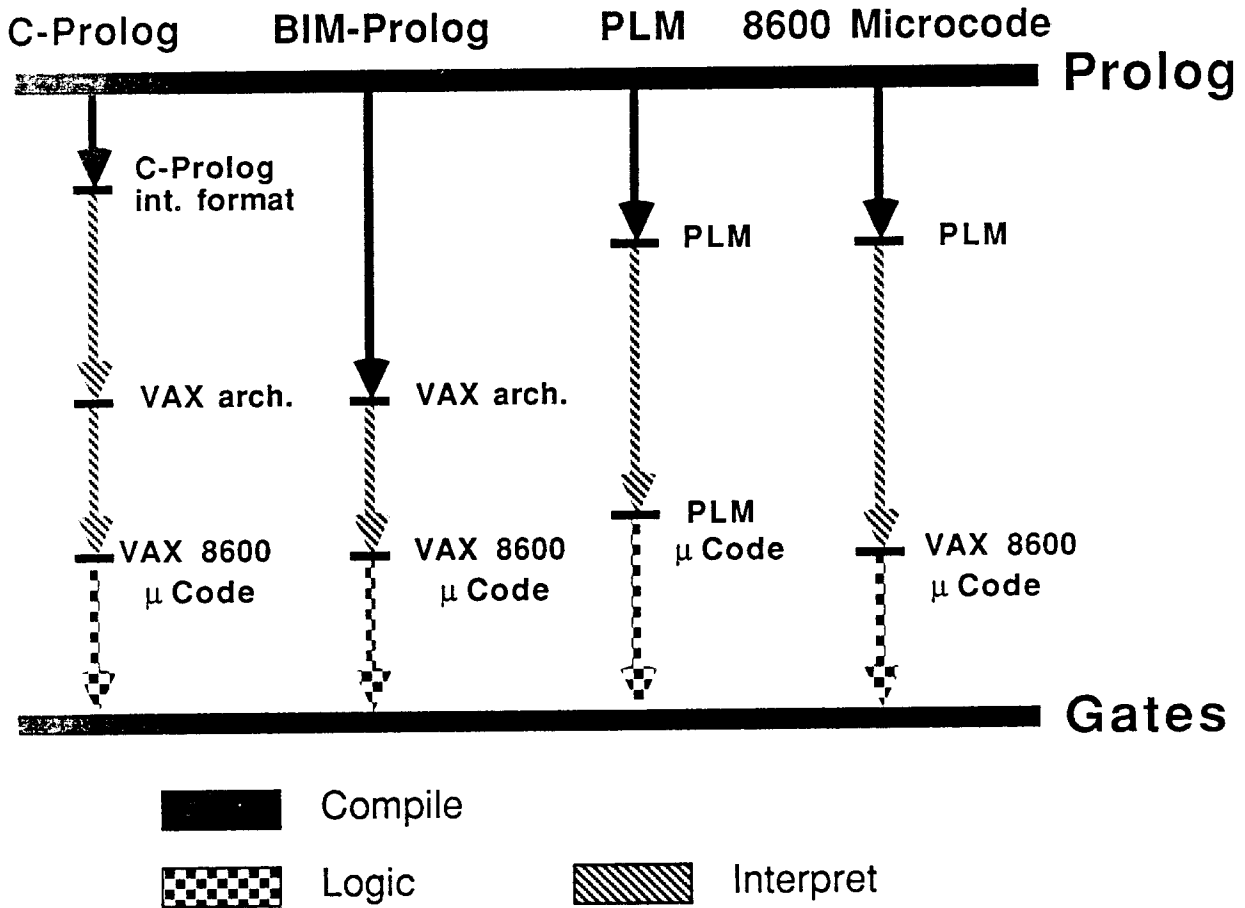


Figure 2.1: Prolog Execution Models

Chapter 3

Details of the PLM, a Modified WAM

1. Introduction

This chapter describes the PLM architecture implemented on the VAX 8600. The architecture is heavily based on the original Warren Abstract Machine [7] except for slight changes to the instruction set and processor registers. A few new instructions were added to the WAM to support the cdr-coding of lists (described in Chapter 5, section 1), the Prolog cut (!) operator, and Prolog built-in predicates. Several registers were added to the architecture to improve performance. Additional detail on the PLM architecture is provided in [2,3].

2. Data Types

Prolog manipulates four types of data: constants, variables, lists, and structures. Data in the PLM consists of a word containing a value and a tag. The tag determines the data type for the object; the value generally represents an address. Constants can be integers, atoms, and the special constant NIL. Variables point to the data to which they are bound. Unbound variables point to themselves. Lists reference the first element of the list. Structures are lists with principal functors. The first element of the list is the principal functor of the structure.

3. Registers

The architecture contains 18 special purpose registers:

- A1-A8: Argument registers, containing the arguments of a Prolog goal.
- P: Program counter, addressing the next instruction to execute.
- CP: Continuation pointer, where execution continues should the current goal succeed.
- E: the Environment pointer, references the current environment placed on the stack.
- B: the Backtrack pointer, contains the address of the current choice point placed on the stack.

- TR: the Trail pointer, pointing to the top of the trail.
- H: the Heap pointer, pointing to the top of the heap.
- HB: the Heap Backtrack pointer, the value of the heap pointer when the current choice point was placed on the stack.
- S: the Structure pointer, pointing to the current element of a list or structure being accessed.
- PDL: the Push Down List pointer, pointing to the last element placed on the push down list.
- N: the number of permanent variables in the current environment.

4. Data Memory Allocation

The data memory is partitioned into four spaces: the Stack, Heap, Trail, and Push Down List (PDL).

The stack is used to store control information necessary for the correct execution of a Prolog program. Choice points and environments are placed on the stack by special instructions which save data needed for backtracking.

An environment contains the saved state of a Prolog clause. It contains register values and "permanent variables" which must be retained between goals in a multi-goal clause. Permanent variables are variables whose use is not restricted to the first goal in a clause. Thus, if kept in argument registers, these variables may be overwritten during execution of a subsequent clause goal. These variables are stored on the stack and retrieved when the appropriate goal is invoked. In addition, an environment contains the CP, E, N, and B registers which are necessary to continue computation when the last goal in a clause succeeds.

A choice point contains the information necessary to restore the process state when a goal fails. Choice points are placed on the stack whenever a procedure contains more than one clause which can unify with the current goal. Choice points contain the following register values:

- A1..A8: the contents of the argument registers
- E: the location of the last environment
- CP: address to continue if the current goal succeeds
- B: location of the previous choice point

- TR: value of the trail pointer
- H: top of the heap
- N: number of permanent variables in the current environment
- L: address to continue should the current goal fail

The heap is used to store lists and structures. These data items are difficult to store on the stack. Instead, pointers to the lists and structures are stored on the stack. In addition, the heap is used to globalize variables on the stack which may become dangling references when an environment is deallocated [3].

The trail is used to store addresses of bindings which must be undone upon goal failure. When the current goal fails, the trail value saved in the current choice point is retrieved. All addresses in trail locations between this saved value and the current trail pointer are reset to unbound variables.

Finally, the PDL is a small stack used to unify nested structures and lists. Dangling references occur when unifying nested lists. During the depth first traversal of a nested list, pointers to the remainder of the higher levels of the list are lost. This occurs if the address of the cdr is not saved when a nested list is encountered. The Push Down List contains pointers to the remainder of a nested list. Unification resumes at the topmost PDL location during a depth first traversal. When the PDL is empty, the list has been traversed.

5. Instruction Set

The PLM instruction set is described in detail in [3].

In addition, the Berkeley PLM supports the built-in predicates of Prolog through the escape mechanism [2]. These predicates are not executed by normal WAM constructs, but are represented as a particular escape instruction. The escape sequences supported by our implementation include:

- | | |
|---|---|
| <p>Input/Output</p> <ul style="list-style-type: none"> • get • put • read | <ul style="list-style-type: none"> • write • tab • nl • see |
|---|---|

- seen
- tell
- told

Term Comparison

- >
- <
- ==
- ≡
- =<
- >=

Arithmetic

- +
- -
- *
- /
- mod

Metalogical

- var
- nonvar
- atom
- atomic
- number
- integer

- functor
- arg
- =..
- length
- name

System

- system

Chapter 4

Operating Environment

1. 8600 System Architecture

The PLM instructions are implemented on a VAX 8600 computer operating under 4.3 BSD UNIX.² The VAX 8600 is a 32 bit computer designed with ECL macrocell arrays. Figure 4.1 shows a block diagram of the 8600. The cycle time of the 8600 is 80 nanoseconds.

The 8600 is partitioned into four major units that work concurrently, each performing a different part of the overall execution of an instruction. The IBOX prefetches the instruction stream, processes operand specifiers, and passes operands and instruction-dependent control information to the EBOX, the main execution unit of the machine. The EBOX executes the VAX instruction set and supervises the entire system under exceptional conditions. The EBOX also contains the main data path and most of the microcode in the 8600. The MBOX performs memory accesses requested by the IBOX and EBOX. It contains the translation buffer, cache, and I/O sub-system interface. Finally, the FBOX is a floating point accelerator, containing special hardware to achieve a high performance computing capability. The FBOX is optional; the EBOX will execute all VAX floating point instructions if the FBOX is not present.

Sixteen general purpose registers are available to the programmer. Four copies of these registers are maintained to guarantee fast and flexible access to the data. Any modification updates, by means of special hardware, all copies of the registers.

The main interface signals between the four major units are shown in figure 4.1. All memory and I/O accesses occur via the Memory Data Bus (MD-Bus) which connects the MBOX to the IBOX. Memory operands are passed from the IBOX to the EBOX across the Operand Bus (OP-Bus). Operands in the general purpose registers are represented as GPR numbers passed across the IBGPR-Bus. Thus two operands can be passed from the IBOX to the EBOX in one cycle. Results from the EBOX or FBOX destined for memory are returned to the IBOX via the Write Bus (W-Bus). Any modifications to the general purpose registers are also broadcast across the Write Bus to update all other copies. The IBOX passes memory

² UNIX is a trademark of Bell Laboratories.

results to the MBOX via the Memory Data Bus. The EBOX and IBOX supply virtual 32 bit addresses to the MBOX across the EVA and IVA busses, respectively. The FA-Bus is used by the IBOX to send microcode entry points to the EBOX. The CC-Bus provides the IBOX with condition code information computed in the EBOX which the IBOX requires for the branch instructions.

2. 8600 Microarchitecture

All of the boxes are microprogrammed independently. Most of the microcode, including all instruction specific microcode, is contained in the EBOX. The EBOX was remicroprogrammed to execute the PLM instruction set and the IBOX decode RAM (DRAM) entries were augmented to recognize the previously reserved opcodes representing each PLM construct. The additional microcode performs the operations required for each of the PLM constructs and for several of the Prolog built-in functions which are normally represented as escape sequences. The IBOX and MBOX perform the duties of instruction prefetching, operand prefetching, and memory accesses. No IBOX microcode modifications were necessary other than the DRAM entries which are needed by the EBOX, since normal VAX addressing modes and the extended VAX opcodes (FD xy) are used to represent a PLM program.

The EBOX contains 8K x 92 bits of writable control store. The horizontal microinstruction format facilitates the implementation of a simple, but flexible data path. This flexibility accounts for much of the power of this machine.

The EBOX data path, shown in figure 4.2, consists of a dual-ported 256 x 32 bit scratchpad register file, an ALU, and a barrel shift network. The scratchpad contains internal processor registers, temporary registers, constants, and architecturally defined general purpose registers.

The 8600 microcycle is 80 nanoseconds. In one microcycle, the machine can perform an ALU or shifter operation on two scratchpad elements and store the result back in the scratchpad. The barrel shifter works in parallel with the ALU and can select any 32 consecutive bits from a 64 bit value. Two scratchpad registers or one register concatenated with a memory operand supply this 64 bit value.

The MBOX contains a 16 Kbyte data cache to speed up memory accesses. A memory read takes two microcycles if the data is found in the cache, and seven cycles in the event of a cache miss.

3. Microprogramming Environment

The EBOX microcode source is divided into 20 separate files totaling approximately 75,000 lines. After assembly, roughly 500 lines of microcode are available for use. An additional 500 lines were gained by removing the microcode for PDP-11 compatibility mode. The microcode which implements the PLM is stored in a separate source file and assembled separately from the native microcode. The new microcode in this file will overlay the native code in unused locations.

The source files are stored on a MicroVAX II workstation running the VAX/VMS³ operating system. Assembly takes place on the MicroVAX using the MICRO2 assembler. The resulting microcode is converted into a binary format and transferred to the console disk pack of the VAX 8600. The microcode is then loaded into the writable control store of the 8600 before booting the UNIX operating system.

4. Compilation and Assembly of Prolog Programs

Three levels of translation are required to transform Prolog programs into an executable VAX 8600 object file. First, the Prolog program is compiled into its equivalent PLM form. An intermediate assembler then takes the output of the compiler and generates a VAX assembly language file. This file is then assembled into a VAX executable file, containing both VAX opcodes defined by the architecture and new VAX opcodes for the PLM constructs. Each of the PLM constructs is defined as a two byte VAX opcode of the form FD xy.

The Prolog compiler was developed at Berkeley as a Master's Thesis by Peter Van Roy [6]. The compiler is written in Prolog, and is invoked from a C-Prolog interpreter running under 4.3 BSD UNIX. Input to the compiler is a set of Prolog clauses and a query. The output is the equivalent translation into the PLM instruction set.

The intermediate assembler transforms the code generated by the compiler into a VAX assembly language file. It is written in C, and performs a one to one translation of PLM code to new VAX opcodes. Each PLM instruction corresponds to a single VAX instruction. An extended VAX opcode is defined to represent each of the PLM constructs. The operands of the new instructions are represented as normal

³ VAX/VMS is a trademark of the Digital Equipment Corporation.

VAX operand specifiers. The intermediate assembler is responsible for parsing the PLM file and generating the appropriate new VAX opcodes. The assembler also creates symbol and string tables which represent Prolog atoms, lists, and structures.

In addition, the implementation supports a number of built-in Prolog functions which are represented as escape sequences in the Berkeley PLM. These include input/output predicates such as write, read, and nl, arithmetic operations in the is predicate, metalogical predicates such as integer, functor, and arg, and term comparison operations such as ==, =<, >=, <, and >. The complete list of built-ins supported in this system is listed in the previous chapter. These built-in predicates are either implemented as new instructions, or in-line sequences of VAX code, or calls to subroutines written in C.

The VAX/UNIX assembler as generates executable VAX object code from the output of the PLM assembler.

The entire compilation and assembly process is shown in figure 4.3.

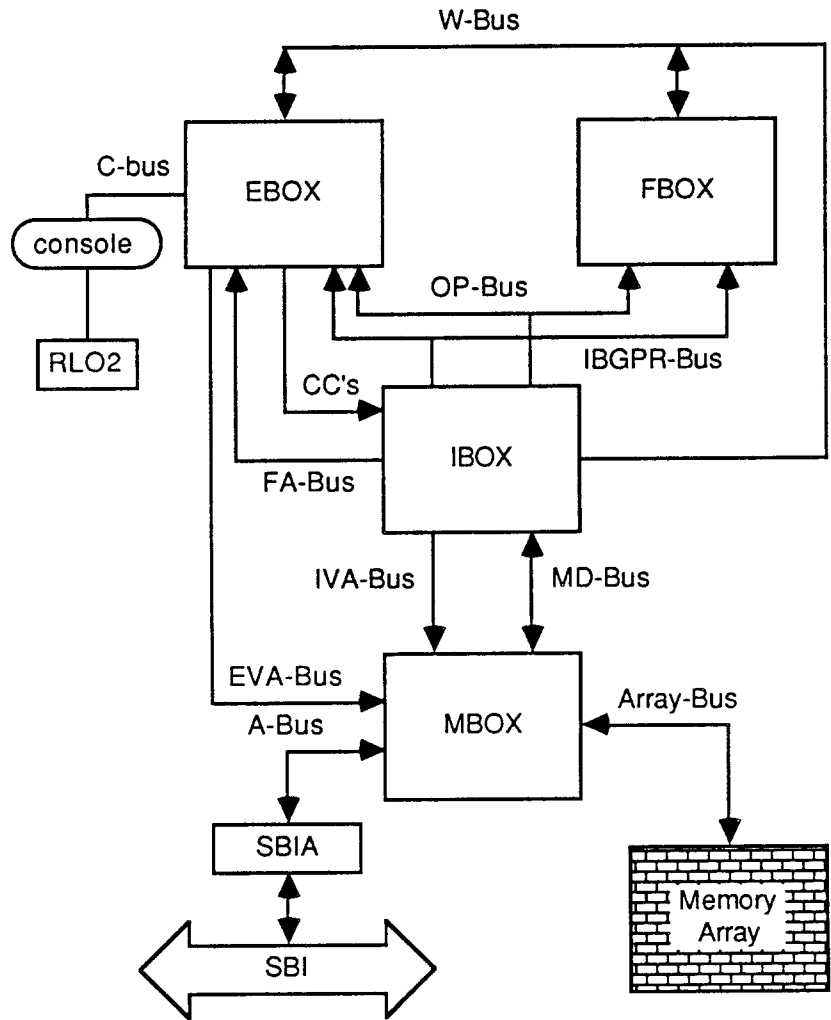


Figure 4.1: VAX 8600 Block Diagram

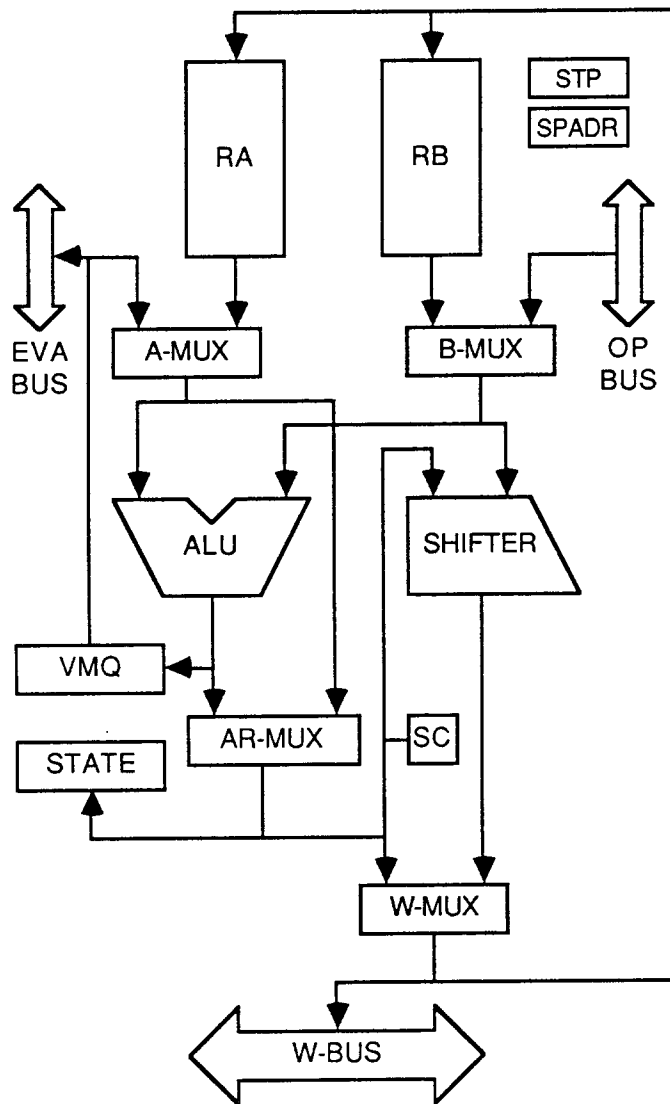


Figure 4.2: EBOX Datapath

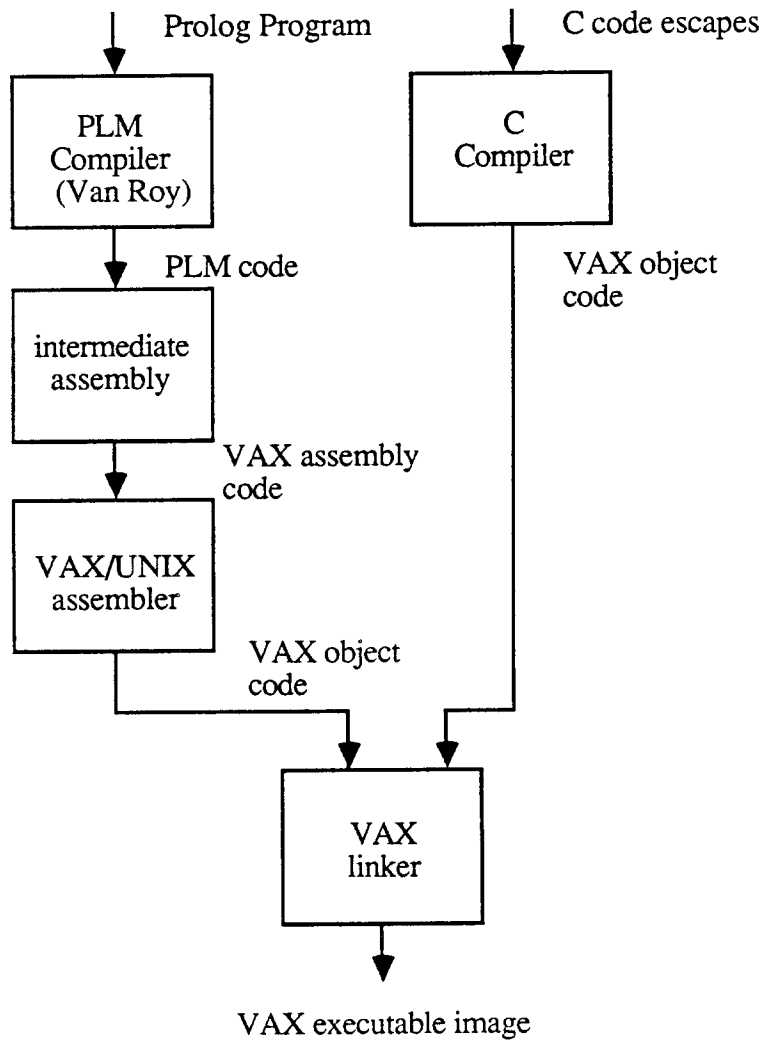


Figure 4.3: Prolog Compilation Process

Chapter 5

Implementation of the PLM Architecture on the VAX 8600

1. Data Representation

The method for implementing data tags is shown in figure 5.1. The two high order bits of a 32 bit data word specify the type of the data. The third bit supports the cdr-coding of lists, to be explained below. Another bit is allocated for garbage collection, which is not implemented in the current system.

Variables contain a 4 bit tag and a 28 bit address. Virtual addresses are 32 bits in the VAX architecture. The remaining high address bits are determined by the high bit of the 28 bit address. If 0, the data exists in heap space (hex 0 followed by the address). If 1, the data is in stack space (hex 7 followed by the address, see figure 5.3). This 32 bit address points to the data to which a variable is bound. For example, a variable bound to a constant contains the address of the constant in memory. Unbound variables address themselves, thus a bound variable can be unbound by modifying its value field to address itself.

Constants require two secondary tag bits which determine the type of constant. Values of constants are placed in the remaining 26 bits of the data word. Integer constants are stored in these bits. Constant atoms are represented by a unique identifier number which is its index in a symbol table. The identifier number provides sufficient information for Prolog unification operations, while the symbol table entry is required for the write predicate. The special constant NIL is represented by all 1's in the remaining bits.

Lists are represented as a data word containing a pointer to the first element of the list. Lists are cdr-coded. The car of the list is the first element; the cdr points to the remainder of the list. To improve memory efficiency, the cdr cell is not included if the rest of the list directly follows the car in memory. Otherwise, the cdr cell directly follows the car. A cdr bit in the data word detects this condition. If the cell following the car has its cdr bit set, it points to the rest of the list. Otherwise, it is the first element of the remainder of the list. A NIL constant ends a list.

Structures are identical to lists except the first element of a structure is the principle functor of the structure.

2. Register Allocation

The architectural registers of the PLM are mapped onto the sixteen VAX general purpose registers (GPR). Each PLM register is assigned to a VAX general purpose register, except for the trail and push down list registers. These registers share a VAX general purpose register, as 16 bits of address space were deemed sufficient for the trail and push down list. We should note that only six argument registers are provided, compared to eight in the PLM coprocessor, due to a shortage of VAX processor registers.

Several PLM instructions perform different functions depending on the state of two mode bits, the cut bit and the read/write bit. The cut bit determines the proper number of choice points to discard when the Prolog cut (!) operator is executed. Normally, all choice points above the B register value saved in the current environment are discarded. However, if the current procedure has placed a choice point on the stack, then one more choice point must be discarded. The cut bit is set when a choice point is placed on the stack and cleared by a call, execute, or proceed instruction. The read/write bit determines the mode for the unify instructions. In write mode, a list or structure is unified with an unbound variable, and a copy of the data is written on the heap. In read mode, two lists or structures are unified, and their elements on the heap are compared. The mode bit is set to read when the argument dereferences to a list or structure, and is set to write if the argument dereferences to a variable.

The read/write and cut bits are stored directly in the VAX Processor Status Longword (PSL). The PSL negative flag implements the read/write bit, and the the PSL overflow flag implements the cut bit. Condition codes in the PSL can be used freely as the PLM instructions do not depend on any condition codes defined by the native VAX architecture.

The register allocation scheme is shown in figure 5.2.

3. Memory Allocation

The VAX 8600 has 31 bits of process address space. Our Prolog implementation requires only 28 bits, due to the four bit tag in the data word. Half of this 28 bit address space is allocated to the code and heap space; the other half is used for the stack and trail space. The virtual address space is allocated according to figure 5.3.

The code space corresponds to the size of the individual Prolog program. The heap space begins where the code space ends and grows toward high memory. The heap boundary occurs when 27 bits of address space are used. The stack starts in VAX P1 space and grows towards low memory. No space is allocated for the Push Down List. Instead, the PDL is stored on top of the stack, as no choice points or environments will be placed on the stack while unifying two lists.

Memory locations 7FFF 0004 and 7FFF 0008 are reserved for process information which would be lost when executing PLM instructions. A Prolog program is invoked by an operating system call to procedure `main`, which performs some initialization and jumps to a subroutine which executes the Prolog code. The return address to the main procedure is stored in location 7FFF 0004. The frame pointer to the stack frame created by the operating system call is saved in location 7FFF 0008. These data must be saved as the PLM instructions do not follow the procedure call and stack frame semantics of the VAX architecture.

Nearly 64 KBytes of high memory are reserved for the trail. This portion of the memory stores addresses of bindings which must be undone upon goal failure. The uppermost portion of process P1 space is reserved for UNIX control information.

4. Process Control

It is intended that the Prolog system will execute within a multiprogramming environment. Thus the entire Prolog process state is stored in the sixteen VAX general purpose registers which are saved in the process control block.

In general, interrupts are handled between instruction boundaries. All process information is safely stored in the process control block when interrupts are executed. However, many PLM instructions execute in non-determinate time due to the usage of the dereference, unify, bind, fail, and trail routines. When the machine unifies long lists or traces through long dereference chains, any interrupt must wait for the operations to complete, which may cause unacceptable latency for certain real-time applications. Wherever these long loops occur in the microcode, the VAX first part done mechanism [10] is used to allow processing of interrupts within an instruction boundary. The process state is preserved and execution will later resume

where the instruction had left off.

Machine exceptions, such as page faults, are processed immediately. The instruction is restarted after the exception is processed, either at the beginning or at an intermediate state, depending on whether the first part done mechanism was in effect. In the first case, the processor registers are restored to their values before the instruction began execution, and the instruction is re-executed. In the second case, the processor registers are restored to their state at the time of interrupt, and processing is resumed from that point. In both cases, modifications to memory are not backed up. For all instructions the microcode is designed to insure that multiple writes are atomic or to order the writes such that if a fault occurs before the instruction completes, the process can resume without error.

5. Implementation of the PLM Instruction Set

Each of the PLM instructions is implemented as newly defined VAX instructions. Extended VAX opcodes represent each construct, with its associated microcode resident along with the host microcode. The decode RAM has been modified to provide correct fork address generation when the IBOX encounters one of the newly defined opcodes.

The operands of PLM instructions can be partitioned into four types: argument registers (X_i), permanent variables (Y_i), labels (L), and constant literals (N). Operands are encoded using VAX addressing modes and conveniently evaluated by the IBOX. Argument registers are specified with register mode; permanent variables in the current environment are specified via displacement mode from the current environment pointer; labels and constants form 32 bit literals in the instruction stream. Samples of the instruction format are shown in figure 5.4.

The instruction format contains some inefficiencies. Some contributing factors are the extended opcode (FD) byte, and the use of 32 bit literals for labels. The extended opcode byte requires an extra IBOX microcycle to decode, and increases the length of the instruction stream. Since most of the single byte opcodes are used by the native VAX instruction set little can be done about this limitation. Labels could be encoded more efficiently as displacements from the current program counter. One or two bytes of displacement may be sufficient in many cases to represent the target address. However, at present no two-

pass assembler for the PLM code necessary to produce such displacements has been written.

6. Implementation of the Prolog Built-in Predicates

The built-in functions of Prolog provide services not supported by the clause control and unification operations of pure Prolog. These services include input/output, arithmetic, metalogical, and program modification operations. Built-in functions are implemented in four ways, as macro expansions of PLM instructions, new VAX FD instructions, Prolog library procedures, and C functions. The macro expansion technique is implemented by the PLM compiler and won't be discussed here.

6.1. New Instructions

Certain built-in functions can be performed by the microcode and datapath of the VAX 8600. An efficient technique is to create new microcode and allocate a new FD instruction to each of these built-ins. The following operations are handled in this manner:

- Addition and subtraction in the `is` predicate
- Comparison of terms (`==`, `===`, `>`, `<`, `>=`, `=<`)
- Metalogical (`atom`, `integer`, `number`, `=..`, `length`)

Multiplication, division, and modulo in the `is` predicate are implemented with an in-line combination of new instructions and VAX instructions. The VAX instructions handle the arithmetic operation on untagged data, while the new instructions handle data tagging and unification.

6.2. Built-ins in C

Certain built-in predicates require services provided by the UNIX operating system or access to the symbol table containing string representations of Prolog atoms. These predicates are implemented as C functions. Some examples are:

- Input/Output (`get`, `put`, `write`, `nl`, `see`, `seen`, `tab`, `tell`, `told`)
- Metalogical (`name`)
- System (`system`)

Each built-in predicate is represented by a C function. Typed 32 bit longwords are the parameters passed to the functions. The C object code is linked with the PLM code, giving C functions access to the entire address space of the Prolog process.

Subroutines for the I/O built-ins used the standard library functions: `fprintf`, `fscanf`, `fopen`, and `fclose` to generate I/O for the current input and output files.

The `name` predicate poses some difficult problems, allowing new atoms to be created under certain circumstances. The symbol table created by the PLM to VAX assembler only accomodates atoms parsed in the PLM code. A data structure is maintained in the C code to store atoms created dynamically by `name`.

6.3. Built-ins Emulated in Prolog

A final method for handling built-in predicates is to emulate them in Prolog. Many Prolog functions can be synthesized by combinations of pure Prolog and other built-ins. Currently the following special predicates are implemented in Prolog:

- Input/Output (`read`)
- Metalogical (`arg`, `functor`)

`read` scans a Prolog term from the current input file, creating the structure form of the term on the heap. Two new built-in functions were added to support the emulation routine, `readln`, which scans the current line of input, and `gettoken`, which returns successive tokens to the emulator. Tokens can be atoms, variables, and punctuation. The Prolog code for `read` parses successive tokens returned by `gettoken` into a valid Prolog structure.

`arg` and `functor` are easily synthesized in Prolog given the availability of the `univ (=..)` and `length` functions in microcode.

The Prolog code for `read`, `arg`, and `functor` is compiled to a PLM code program. The PLM to VAX assembler merges the PLM code for built-ins with the PLM code for the user's application.

7. Implementation of Basic Prolog Routines

Several basic Prolog functions used by many of the PLM constructs are also implemented in microcode. These include the dereference, decdr, unify, and fail routines. Only the fail routine is directly accessible to the user to initiate backtracking. The dereference routine follows a chain of variables until a structure, list, constant, or unbound variable is encountered. The decdr routine supports the cdr-coding of lists, and insures that a list is traversed correctly.

The unify routine performs the unification, binding, and trailing operations necessary when two Prolog variables are unified. The fail routine resets all trail addresses to unbound variables upon goal failure, and restores the state of the PLM registers from the last choice point placed on the stack.

Nearly 700 lines of microcode were added to the VAX 8600 to implement the PLM architecture.

8. Sample Compilation Process

In this section a Prolog example is followed through the compilation process.

The Prolog procedure `concat`, used to concatenate two lists, is shown below. The procedure consists of two clauses, each with three arguments. The first two arguments represent lists to be concatenated; the last represents the resulting list. The first clause provides the termination condition; a null list concatenated with list `L` is simply `L`. The second clause handles the general case; concatenating a list whose first element is `X` and remainder is `L1` to list `L2` is `X` followed by the concatenation of `L1` and `L2`.

```
concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

The first step in the transformation process involves compiling the Prolog source code to the instructions of the PLM architecture.

```
procedure concat/3
    switch_on_term _371,_372,fail
_373:
    try_me_else _374
_371:
    get_value X2,X3
    get_nil X1
```

```

    proceed
_374:
    trust_me_else fail
_372:
    get_list X1
    unify_variable X4
    unify_cdr X1
    get_list X3
    unify_value X4
    unify_cdr X3
    execute concat/3

end

```

The intermediate assembler translates each PLM instruction into a newly defined VAX instruction. To avoid modifying the VAX assembler as, the intermediate assembler generates `.byte`, `.word`, and `.long` directives followed by the opcode, operand specifier, or constant specified in hexadecimal. As recognizes only the native VAX assembly language instructions such as `movl`, `addl`, etc.

For example, the first `switch_on_term` instruction in procedure `concat` is represented by its two byte opcode `38fd`, followed by three labels. Labels are 32 bit addresses in the instruction stream (specified by the byte code `0x8f` followed by the longword address). The address `0xffffffff` represents a fail label.

```

concat_3:
    .word 0x38fd
    .byte 0x8f
    .long _371
    .byte 0x8f
    .long _372
    .byte 0x8f
    .long 0xffffffff

_373:
    .word 0x3bfd
    .byte 0x8f
    .long _374

_371:
    .word 0x0bfd
    .byte 0x52
    .byte 0x51

    .word 0x09fd
    .byte 0x50

    .word 0x0ffd

_374:

```



```
.word 0x1ffd  
_372:  
.word 0x08fd  
.byte 0x50  
  
.word 0x28fd  
.byte 0x53  
  
.word 0x22fd  
.byte 0x50  
  
.word 0x08fd  
.byte 0x52  
  
.word 0x26fd  
.byte 0x53  
  
.word 0x22fd  
.byte 0x52  
  
.word 0x05fd  
.byte 0x8f  
.long concat_3
```

Reference

10	C	G	pointer
----	---	---	---------

Constant

11	C	G	XX	identifier
----	---	---	----	------------

XX = 00 - small integer
 01 - other numeric value
 10 - atom
 11 - NIL

Structure

01	C	G	pointer
----	---	---	---------

LIST

...	C	G	pointer
-----	---	---	---------

C = 0 - non-cdr
 1 - cdr

G = garbage collect

Figure 5.1: Data Representation

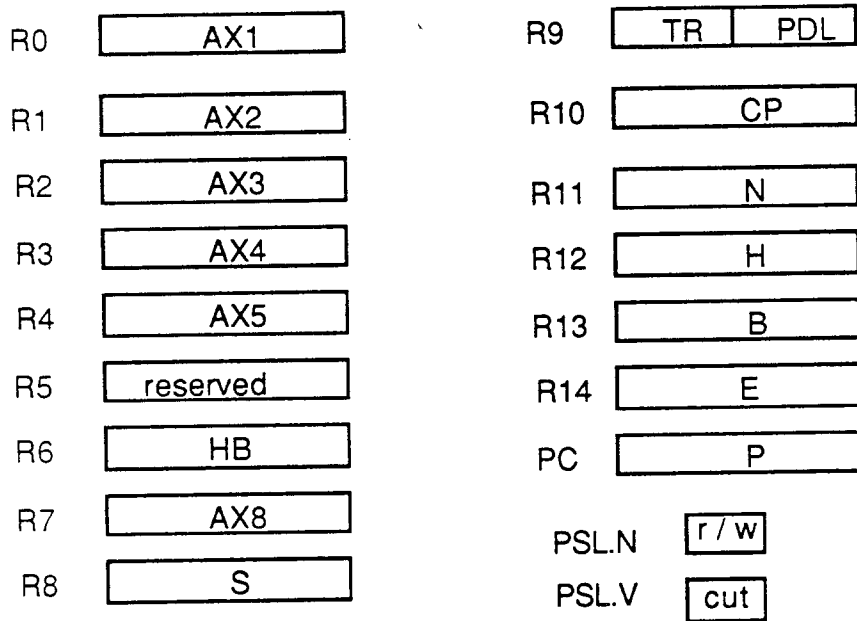


Figure 5.2: Register Allocation

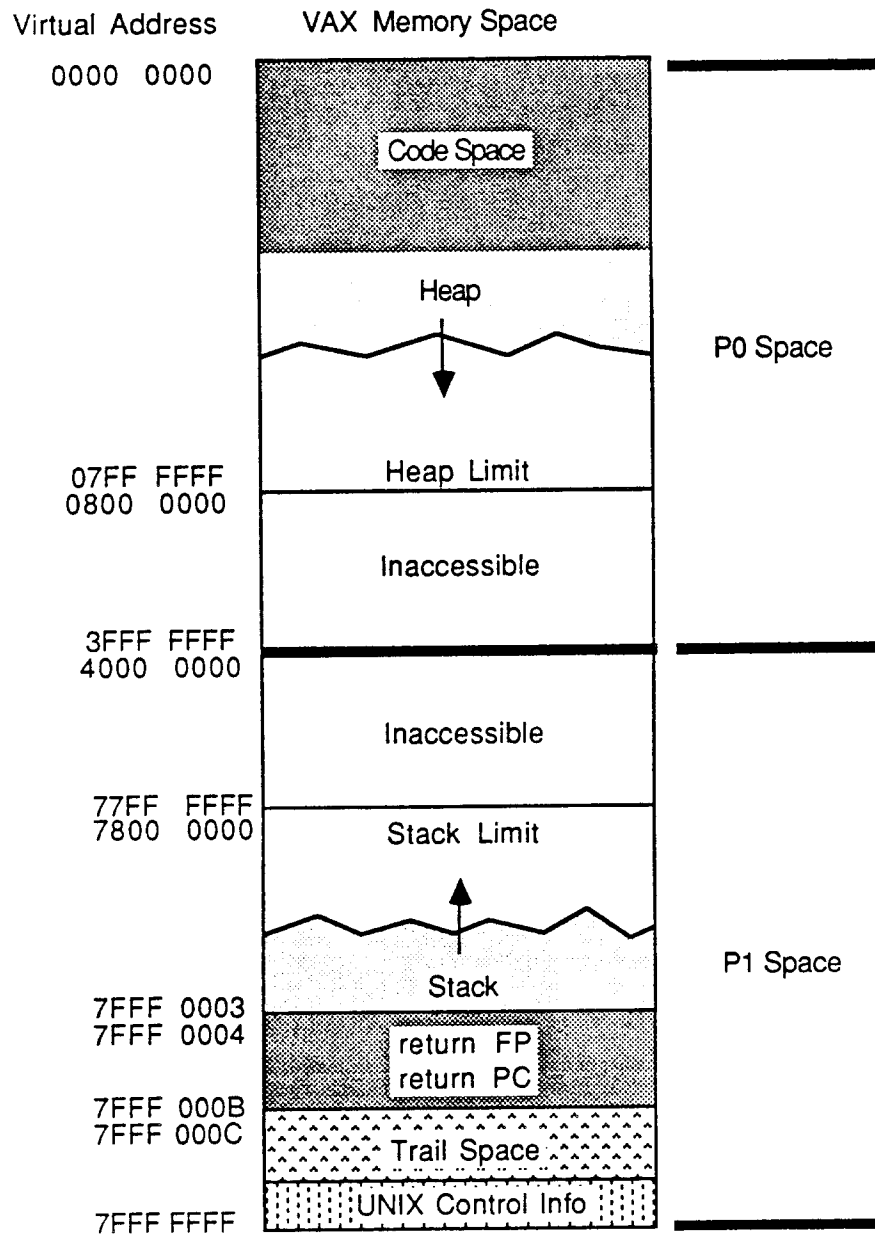


Figure 5.3: Virtual Memory Allocation

allocate

FD	00
----	----

call L,n

FD	01	8F	n	8F	L	L	L	L
----	----	----	---	----	---	---	---	---

get_constant c, Xi

FD	07	8F	c	c	c	c	5i
----	----	----	---	---	---	---	----

get_variable Yn, Xi

FD	0E	5i	CE	disp	disp
----	----	----	----	------	------

switch_on_term Lc, Li, Ls

FD	38	8F	Lc	Lc	Lc	Lc
----	----	----	----	----	----	----

8F	LI	LI	LI	LI
----	----	----	----	----

8F	Ls	Ls	Ls	Ls
----	----	----	----	----

Figure 5.4: Sample Instruction Formats

Chapter 6

Measurements and Analysis

1. Measurement Philosophy

Conventionally, the performance of a Prolog implementation is measured in logical inferences per second (LIPS). As mentioned previously, a logical inference represents the invocation of a Prolog procedure. Each PLM call, execute, or escape instruction executed is counted as a logical inference. Dividing the total inference count by execution time results in the LIPS measure for a particular benchmark.

Accounting for the execution time due to built-in predicates can be a problem. For example, a Prolog coprocessor may not be able to do I/O in the read or write predicates, or division in the is predicate, leaving these operations to its general purpose host. In the simulator for the PLM special purpose processor, no execution time is counted for built-ins which are performed externally.

The measurement philosophy taken in this report is to eliminate as much as possible inconsistencies in reported execution time due to built-in predicates. First, all occurrences of the write and nl built-ins, which are non-essential to the correct execution of the benchmarks, were removed. However, discrepancies due to the essential built-ins, such as the is predicate, still exist.

Four sets of performance numbers will be presented in the next section. Three sets belong to implementations available on the same general purpose machine, namely the VAX 8600. These results include the execution time for all the Prolog built-in predicates. The other set of results belong to the Berkeley PLM coprocessor, which executes most of the built-ins directly in its microcode. The PLM expects its host to perform the remaining built-ins, and attributes zero time for these operations.

2. Performance Measurements

The performance of the microcoded implementation was measured on fourteen common benchmarks. The standard technique of measuring the cpu time for multiple iterations of the benchmark was used. Multiple iterations are necessary to increase the accuracy of the measurement, especially for short benchmarks. Dividing the total cpu time by the number of iterations results in the execution time for the particular benchmark. The measurements were taken with the UNIX time facility.

2.1. Implementations on General Purpose Computers

The results are compared with the performance of the other systems available on the VAX 8600, interpreted C-Prolog and compiled BIM-Prolog.

Table 6.1 summarizes results on the benchmarks from these three Prolog systems. The first column corresponds to the microcoded implementation of Prolog. The second column corresponds to BIM-Prolog, which compiles to the native VAX architecture. The last column corresponds to the C-Prolog interpreter. Performance results for the BIM and C-Prolog systems were taken with the `cpuTime` built-in predicate.

Performance of Various VAX 8600 Prolog Systems			
Benchmark	u-coded 8600 klips	BIM_Prolog klips	C-Prolog klips
con1	106	42	5.6
con6	44	16	3.8
hanoi	122	38	5.5
mumath	83	26	5.3
pri2	118	8	3.1
queens	103	12	2.5
nrev	130	45	7.6
qs4	111	32	5.3
palin25	79	26	5.5
times10	56	16	3.5
div10	46	14	3.3
log10	59	16	2.6
ops8	70	20	3.8
query	95	43	2.7
averages	85	25	4.3

Table 1: Comparative Performance of VAX 8600 Prolog Implementations

The results show that our microcoded PLM interpreter provides the best performance of these three systems, averaging 85 kilolips over the fifteen benchmarks. The next fastest implementation is BIM-Prolog, at an average of 25 kilolips, followed by C-Prolog at 4 kilolips. Peak performance for all of the systems is on the naive reverse (`nrev`) benchmark, where the microcode, BIM, and C-Prolog perform at 131, 45, and 8 klips respectively. From the normalized results in Table 6.2, we see that the microcode is over three times faster than BIM Prolog, and nearly twenty times faster than C-Prolog.

Normalized Performance of Various VAX 8600 Prolog Systems			
Benchmark	u-coded 8600 klips	BIM_Prolog klips	C-Prolog klips
con1	1	.40	.05
con6	1	.36	.09
hanoi	1	.31	.05
mumath	1	.31	.06
pri2	1	.06	.03
queens	1	.12	.02
nrev	1	.35	.06
qs4	1	.29	.05
palin25	1	.33	.07
times10	1	.29	.06
div10	1	.30	.07
log10	1	.27	.04
ops8	1	.29	.05
query	1	.45	.03
averages	1	.29	.06

Table 6.2: Normalized Performance Ratios of VAX 8600 Prolog Implementations

As expected, the compiled systems (microcode and BIM) outperform the interpreted system (C-Prolog). On the primes benchmark (pri2), BIM Prolog is noticeably slower than average, being only twice as fast as C-Prolog. This program, which finds prime numbers with the sieve of Eratosthenes algorithm, executes a large number of modulo operations. Since none of the other benchmarks contains this operation, we suspect that modulo in BIM is not implemented in a particularly efficient manner, at least in the version we are using (version 2.0).

Overall, the results support the claim that symbolic computations are more efficiently implemented in microcode rather than the higher-level VAX instruction set. The microcode is over three times faster than BIM Prolog although both are based on the Warren abstract architecture. The results also support the notion that compiled Prolog systems can provide order of magnitude performance increases over interpreted systems. BIM-Prolog performs nearly an order of magnitude better than interpreted C-Prolog by eliminating the dynamic translation process from an internal form to VAX code.

2.2. General vs. Special Purpose Implementations

Table 6.3 compares results on the same programs between the microcoded 8600 implementation and the Berkeley PLM special purpose Prolog coprocessor. The performance results are not normalized to the

cycle time of each machine; a column in Table 6.3 provides normalized performance ratios. Essentially, the PLM outperforms the 8600 in all the benchmarks except for hanoi, pri2, and queens, although on average the 8600 performs within 15% on an unnormalized and 25% on a normalized basis.

8600 Microcode vs. PLM			
Benchmark	unnormalized		normalized 8600/PLM
	u-coded 8600 klips	PLM klips	
con1	106	185	.47
con6	44	49	.67
hanoi	122	104	.89
mumath	83	96	.66
pri2	118	107	.90
queens	103	75	1.00
nrev	130	185	.57
qs4	111	121	.71
palin25	79	95	.67
times10	56	54	.76
div10	46	47	.73
log10	59	61	.77
ops8	70	69	.75
query	95	123	.66
averages	85	98	.73

Table 6.3: Microcoded 8600 Prolog vs. PLM

Several comments on these results are in order. The measurements for the Berkeley PLM are simulated results, assuming perfect single cycle memory access. The PLM simulator does not model the execution time to perform certain "external" escapes, such as division and modulo in the `is` predicate. The results for the PLM essentially account for zero time to execute external escapes, rather than the substantial time required for the host to receive operands, generate results, and return the results to the PLM.

As an example, the query benchmark performs a total of 3528 logical inferences, of which 1900 are multiplications or divisions. The PLM simulator used in this report considers these functions as external escapes, and does not attribute execution time to these operations. Later versions of the PLM simulator will have multiplication available as a shift and add routine, which saves the overhead of an external escape but provides lower performance than a typical general purpose processor. The real performance of the PLM on this benchmark may likely be less than the microcoded 8600.

Finally, one should point out that the performance results compared to the PLM include the overhead associated with a real system in a real environment. That is, the VAX 8600 is a virtual memory machine operating in a multiprogramming environment. Thus the overhead due to address translation, page fault handling, and context switching is included in the measurements.

Chapter 7

Conclusions

1. Report Summary

This report describes an implementation of Prolog which provides for fast execution of both the numeric and symbolic constructs of Prolog. Symbolic operations are provided directly in microcode, eliminating the semantic gap associated with implementing Prolog on a general purpose machine. Numeric operations are executed with in-line VAX instructions, eliminating the time consuming escape mechanisms associated with current special purpose Prolog coprocessors requesting a general purpose host. In this system the host and the Prolog coprocessor are the same, and such overhead does not exist.

Performance results were presented for systems corresponding to four uniprocessor execution models for Prolog. These results indicate that the PLM special purpose coprocessor provides maximal performance, followed closely by the VAX 8600 microcoded implementation, then compiled and interpreted systems on the 8600. The results for the PLM do not include the substantial time required for the host to perform certain numeric computations, thus the microcoded Prolog system may indeed provide superior performance for certain applications.

On the other hand, one would be remiss to altogether dismiss the special purpose Prolog coprocessor for at least two important reasons. First, the state of the art for real Prolog coprocessors is still in its adolescence. There are bound to be improvements as people understand better how to optimize the data path for Prolog processing. Second, the large overhead incurred with the escape mechanism illustrates another area where research in overall system design of Prolog/numeric processing should result in reducing that overhead. Indeed, the Aquarius group at Berkeley is investigating that issue.

But until such better understanding occurs (and perhaps even then), the implementation described in this paper may prove to be the most cost-effective implementation method for handling computations that have substantial symbolic and numeric components.

2. Meeting the Project Goals

All four of the project goals were met.

- (1) The instruction set of the Berkeley PLM was translated one-for-one into newly defined VAX opcodes of the form FD xy. These new VAX instructions perform the operations required for each PLM instruction.
- (2) The system operates in a multiprogramming environment. Interrupts and exceptions are handled via normal VAX mechanisms. The process state is kept within the VAX process control block to insure that a process is restartable.
- (3) The native VAX architecture is preserved. New microcode is stored in unused microcode locations along with the native microcode of the 8600. PDP-11 compatibility mode microcode was removed to gain additional microcode locations, but this did not present any problems to users of the system.
- (4) The implementation outperforms compiled (BIM-Prolog) and interpreted (C-Prolog) Prolog systems on the VAX 8600. Performance is superior to any VAX 8600 Prolog implementations known to the author.

Acknowledgements

I would like to thank my advisor, Professor Yale Patt, for providing the ideas which made this work possible. He introduced me to Prolog, the Warren Abstract Machine, and the Berkeley PLM, and suggested a microcoded implementation of the PLM on the VAX 8600.

I am indebted to Steve Melvin, who provided many details on the VAX 8600 and assisted in the test and debug of the initial implementation. In addition, Steve also wrote the intermediate assembler and provided many useful ideas on implementing Prolog efficiently on the 8600.

I also wish to acknowledge the Digital Equipment Corporation for their generous support of this research, in particular Bill Kania for providing the VAX 8600 in order to enhance the ability to do research in microarchitecture and microprogramming; also, Fernando Colon Osorio, Mario Troiani, Nii Quaynor, Steve Ching and Harold Hubschman, from DEC's High Performance Systems and Clusters group in Marlboro.

This work in microarchitecture is part of a larger architectural research effort at Berkeley, the Aquarius Project. I acknowledge Professor Al Despain, who presides over the Aquarius Project, for the helpful suggestions which greatly improved this report. I also acknowledge colleagues in the Aquarius group for the stimulating research environment which they provide.

Finally, this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

References

- [1] Dobry, T., Despain, A.M., and Patt, Y.N., Performance Studies of a Prolog Machine Architecture, in: *Proceedings of the 12th Intl. Symposium on Computer Architecture*, 1985, pp. 180-190.
- [2] Dobry, T., A High Performance Architecture for Prolog, Ph.D. Dissertation, University of California, Berkeley, California, 1987.
- [3] Fagin, B., and Dobry, T., The Berkeley PLM Instruction Set: An Instruction Set for Prolog, UCB Research Report, CS Division, University of California, Berkeley, California, 1985.
- [4] Fossum, T., McElroy, J., and English, B., New VAX Squeezes Mainframe Power Into Mini Package, *Computer Design*, March 1985, pp. 173-181.
- [5] Shaefer, M.T., and Patt, Y.N., Improving the Performance of UCSD Pascal Via Microprogramming on the PDP-11/60, in: *Proceedings of the 16th Annual Microprogramming Workshop*, 1983, pp. 140-148.
- [6] Van Roy, P., A Prolog Compiler for the PLM, Master's Report, University of California, Berkeley, California, 1984.
- [7] Warren, D.H.D., An Abstract Prolog Instruction Set, Technical Report, SRI International, Menlo Park, California, 1983.
- [8] Warren, D.H.D, Applied Logic - Its Use and Implementation as a Programming Tool, Technical Note 209, SRI International, Menlo Park, California, 1983. Reprint of Ph.D. Thesis submitted in 1977 to the University of Edinburgh.
- [9] *VAX Architecture Handbook*, Digital Equipment Corporation, 1981, pp. 102-104.
- [10] *VAX Architecture Handbook*, Digital Equipment Corporation, 1981, pg. 100.

Appendix A

Microcode Listing

This appendix contains the new EBOX microcode and IBOX DRAM entries added to the VAX 8600 to implement the Warren Abstract Machine. The microcode is in hexadecimal format, prefaced by its address in control store.

EBOX Microcode

1b00:	00a19071c0086230700fdb00	1c08:	0801b0410008615860007dc9
1b42:	00c1b9b0800062b87e201c04	1c09:	00f199310408623461401e0c
1b43:	00c1b0708000627c7a001cf9	1c0a:	08219073000061b56d007dda
1b46:	00c19078800662b476601c1c	1c0b:	00a19031000062b46a501c66
1b48:	0881b0410000611060001c58	1c0c:	08c198990003713460001c0e
1b49:	00a190b8800662307e001c3c	1c0d:	08c190610008621867601c52
1b4a:	0801b0400000615860007dc9	1c0e:	08c198990003713c60001c14
1b4b:	00a190010000625061701c12	1c0f:	0801b0410008615860001c02
1b4e:	00a190708008623070005f00	1c10:	0801b0410008615860007dc9
1b50:	00a199300008627061705f00	1c11:	00a190310000623061531c9d
1b53:	00c19078800662b872601cd2	1c12:	0801b0410008611060007dc9
1b56:	00a19938800862707ed01c44	1c13:	00a190310000627061705f00
1b58:	00a19938800e62707ed01c48	1c14:	00a190718000623070001c16
1b5e:	0801b0400000611060201c18	1c15:	00a19039800062307150fe0a
1b62:	0041b930000862346a001cbc	1c16:	004198b1000972b44e001d0d
1b64:	00419c000000629065721cc3	1c17:	00a190b98006723070001c70
1b6a:	00a1d9300000627067805f00	1c18:	00a1b031800062707310fe2a
1b6c:	00a19930000862786cf21cd3	1c19:	00a199310000623861009cba
1b6e:	0881b0410000611060201c60	1c1a:	00a1b031800062707310fe2a
1b6f:	00419c00000062906cb21cf3	1c1b:	00a190010000625061509ce2
1b70:	00a19939800662707ef01cb0	1c1c:	00a119310008627067a01c1e
1b71:	00419430000862b06c001cc4	1c1d:	00c1b9b1800862f872501c75
1b72:	00419c00000062906cb05f00	1c1e:	0021b4330000623161001c24
1b76:	0801b0400000611060007dc9	1c1f:	00a19c310000627865601c74
1b77:	00a19031000062306a531c8e	1c20:	0821b0410000615861207dc9
1b7a:	0801b0410008611468007dc9	1c21:	00a199310000627865201e09
1b7b:	00a190010000621061701c10	1c22:	0801b0710008611060007ddb
1b7e:	00c1b9b1800862f872501c75	1c23:	00a190718000623070009d92
1b7f:	002194320000623361005f00	1c24:	00a19031000062306d509c53
1b7f:	000194000000628b74205f00	1c25:	00a19039800062307150fe0a
1c01:	00a190718000623070279e01	1c26:	00a190718000623070001c2c
1c02:	00a199310000623869007dc9	1c27:	00a190718000623070005f00
1c03:	00f199310408623461401e0c	1c28:	00a1b031800062707310fe2a
1c04:	094190410000615c67809c43	1c29:	00c190b1000862706f001e09
1c05:	00a19039800062307150fe0a	1c2a:	00a1b031800062f47330fe6a
1c06:	08a198990003713060001c0c	1c2b:	00a19931000062306d001e18
1c07:	00a190b98006723070001c51	1c2c:	00a1907b8006623176601c2e

1c2d: 00c19c01000062906c901cee
1c2e: 00019981000072a076601c34
1c2f: 00c1b9b1800862f872501c75
1c30: 0821b0710000611469207ddb
1c31: 00a19931000062346d201c32
1c32: 0821b0710000615c69207ddb
1c33: 00a19931000062386d001e19
1c34: 08019073000861b56c201c36
1c35: 00a19039800062307150fe0a
1c36: 00c19931000962b841001d0d
1c37: 00a1b0718008623071001c28
1c38: 0821b0710000611469207ddb
1c39: 00a19931000062346d001c3a
1c3a: 0821b0710000615c69207ddb
1c3b: 00a19931000062386d001e32
1c3c: 00a190b98006723070001c3e
1c3d: 08a19919000b613866001d8e
1c3e: 084190bb0000717d6a201c40
1c3f: 00c1b9b1800862f872501c75
1c40: 084190bb000671356a001c41
1c41: 00a190718000623070201c42
1c42: 08419073000961b54e201d0d
1c43: 08c19919000b61f466289c4e
1c44: 06a19051000b603060201c45
1c45: 00a190718000623070001c46
1c46: 00a19071c0086230700fdb00
1c47: 08c19959000361b461289c4e
1c48: 00a190718000623070001c49
1c49: 082190730000613161201c4a
1c4a: 062199310008613061201d0d
1c4b: 0801b0010000615860209c5d
1c4c: 00a199310000623462301c4d
1c4d: 08a1b033000861716ee01c50
1c4e: 08c19899000371b860001c0c
1c4f: 00419c01000862986cb01c06
1c50: 00a1b0718008623071209c05
1c51: 00a190710008627460601c4d
1c52: 08a19919000b617069001c54
1c53: 00a19079800e623076401c2e
1c54: 00a19931000062706cd01c55
1c55: 08a19919000b61706f201c56
1c56: 00c19071000862386e501c0f
1c57: 00a19079800e623072401c26
1c58: 00a1b031800062787330fe4a
1c59: 00a19039000e623861701c4c
1c5a: 00c1b9b1800862f872501c75
1c5b: 00c1b9b1800862f872501c75
1c5c: 00a1b031800062787ec01c4b
1c5d: 00a199310000623462101c02
1c5e: 00a190718000623070201c0a
1c5f: 00c1b9b1800862f872501c75
1c60: 00a1b031800062787330fe4a
1c61: 00c1b9b1800862f872501c75
1c62: 00a190310000627869701c08
1c63: 00a19931000062f861201c68
1c64: 00c1b9b1800862f872501c75
1c65: 00c1b9b1800862f872501c75
1c66: 00a1b9b1800862b476501c6e
1c67: 00a190b98006723070001c70
1c68: 0881b0410000615860007dd2
1c69: 00a190398006623871501c5e
1c6a: 00c1b9b1800862f872501c75
1c6b: 00c1b9b1800862f872501c75
1c6c: 00c1b9b1800862f872501c75
1c6d: 00a190b9800e627076001c7b
1c6e: 00a190398000623871509c65
1c6f: 00a190b9800e627072501c76
1c70: 00a190718000623070201c71
1c71: 08a1b033000861716ee01c72
1c72: 00a1b0718008623071009c15
1c73: 002199310c01627cc9201c9a
1c74: 00a199310000627065001e09
1c75: 00f199310408627464109c6d
1c76: 00a190718000623070201c78
1c77: 00a198b9000e62f061201c93
1c78: 0021943300016231c1201c79
1c79: 087190710c0861b860201c7a
1c7a: 00a190718009623050001d0d
1c7b: 08c190410000621867601c7c
1c7c: 082190bb0000717d69001c7e
1c7d: 008190598000722079701c7f
1c7e: 00a190b98006723070001c80
1c7f: 08a19051000b613460001c98
1c80: 00a190b98006723070201c81
1c81: 084190bb000671b166001c82
1c82: 084190bb000671356a201c83
1c83: 084190bb000671b56e001c84
1c84: 084190bb0006717d6a201c85
1c85: 084190bb0006717962001c88
1c86: 00a1b9b18008623871001c92
1c87: 00a1907180006230702008e1
1c88: 080190bb000e71fd6c001c89
1c89: 080190bb000e71b164001c8a
1c8a: 080190bb000e717d68201c8b
1c8b: 080190bb000e713568001c8c
1c8c: 080190730008617960201c8d
1c8d: 080190730008613160201c90
1c8e: 00a1b9b18008623072101c9b
1c8f: 00a1b9b18008623076501c9b
1c90: 084190410008621867601c91
1c91: 00a1b9b18008623871001c92
1c92: 00f199310408627469c09c73
1c93: 00a190718000623070001c94
1c94: 08a1b0330008617563101c96
1c95: 00c1b9b1800862f872501c75
1c96: 00a190398000623471509c7d
1c97: 00a190718009623050001d0d
1c98: 00a190718000623070201c99

1c99: 002198b10008627861061c86
1c9a: 00a190718009623050001d0d
1c9b: 00a190718000623070009c95
1c9c: 09719071000068b861001c9e
1c9d: 00a190718000623070231ca6
1c9e: 09a19071000066706f301ca8
1c9f: 00a19939800062707ef01ca1
1ca0: 00a190718000623070209cad
1ca1: 06a19051000b603060201ca2
1ca2: 088190990003713860201ca3
1ca3: 088190990003713460001ca4
1ca4: 088190990003713c60001ca5
1ca5: 008199310008623861201cae
1ca6: 002190710000623861001c9c
1ca7: 002199b10000623861001c9c
1ca8: 0881b0410000611060007dd2
1ca9: 00c1b9b1800862f872501c75
1caa: 00c1b9b1800862f872501c75
1cab: 00a1b0318000627873101df8
1cac: 00a1b9b18008623871201ca0
1cad: 00c1b9b1800862f872501c75
1cae: 008199310001627c41201d0d
1caf: 00a190718009623050001d0d
1cb0: 09719041000068d860001cb1
1cb1: 082190bb0006717961001cb2
1cb2: 00a190b98006723070001cb3
1cb3: 080190bb000e717960201cb4
1cb4: 080190730008613568001cb6
1cb5: 0041993100016234ca001cc2
1cb6: 080190730008617d68201cb8
1cb7: 00a190b9800e627072701cbd
1cb8: 09a19071000066706f301cb9
1cb9: 062199310008617861001ca8
1cba: 00a19071c0086230700fdb00
1cbb: 00a1b0318000627073101de8
1cbc: 00a119318000627077809cb5
1cbd: 008199310008627060401cc0
1cbe: 00a19071c0086230700fdb00
1cbf: 00a19071c0086230700fdb00
1cc0: 0021943300016231c1201cc1
1cc1: 00a190718009623050001d0d
1cc2: 00a190718009623050001d0d
1cc3: 08a19919000b613066001cde
1cc4: 00c199310000627c67a01cc5
1cc5: 00c199310000627467801cc6
1cc6: 00c19931000062f862701cc8
1cc7: 00a199398006623072201cda
1cc8: 00c19931000062f462501cc9
1cc9: 08719041000862f879601cca
1cca: 008199310008623860001ccb
1ccb: 008199310008623460201ccc
1ccc: 008199310008623c60201cce
1ccd: 00a190310000627061705f00
1cce: 00819931000862b060001cd0

1ccf: 00a199310000623062005f00
1cd0: 00819931000862bc60201cd1
1cd1: 00c199310009627845701d0d
1cd2: 00a190718000623070201cd4
1cd3: 08a19919000b613866201ce8
1cd4: 00a1907b8006623176401cd5
1cd5: 00019981000072a076601cd6
1cd6: 08019073000861b56c201cd8
1cd7: 00a199398006623072201ce0
1cd8: 00c19931000062b861001cd9
1cd9: 00a119310001627047801d0d
1cda: 00a190718000623070201cdb
1cdb: 08a1b033000861716ec01cdc
1cdc: 00a190718000623070209ccd
1cdd: 00a1b0718008623071201c1a
1cde: 00a190718000623070005f00
1cdf: 00c1b9b1800862f872501c75
1ce0: 00a190718000623070201ce1
1ce1: 08a1b033000861716ee01ce4
1ce2: 00c1b9b1800862f872501c75
1ce3: 00a199310000627861701de5
1ce4: 00a190718000623070009cdd
1ce5: 00a190718000623070009ced
1ce6: 00c1b9b1800862f872501c75
1ce7: 00a1b9b18008627871701ce5
1ce8: 00a190718000623070001ce9
1ce9: 004190b1000962b046001d0d
1cea: 00e19931000862bc62001ceb
1ceb: 08a1b033000861716ec01cec
1cec: 00a1b0718008623071009c25
1ced: 00c1b9b1800862f872501c75
1cee: 08a19919000b613866201cf0
1cef: 00a190718009623050001d0d
1cf0: 084190610000621867601cf1
1cf1: 08a19919000b613069201cf2
1cf2: 00c19c21000862d06ee01cf4
1cf3: 08a19919000b613066201cf8
1cf4: 08a19919000b617c6f201cf5
1cf5: 00a199310000627061001cf6
1cf6: 00c19071000862386e701cf8
1cf7: 00a199398006623072201cea
1cf8: 00a1d9318000627077a05f00
1cf9: 00c190310000627c69509cfa
1cfa: 00a190718000623070001b7e
1cfb: 00a190718000623070001b7e
1cfc: 00a19931000062386c001cfd
1cfd: 00819931000862b466201d00
1cfe: 00a199310000623862201d06
1cff: 00a199310000627064001cfc
1d00: 094190610008645c67801d01
1d01: 002190730008623449001d02
1d02: 0021993100016234c9201d04
1d03: 004190b1400862b0660fdb00
1d04: 00c190310000623c61601d05

1d05: 00a190718108623070001e09
1d06: 00a199310000623c6c001d08
1d07: 00c190b1400062706f0fdb00
1d08: 00819931000862b466201d09
1d09: 084190410008621867601c91
1d0a: 00a190718009623050001d0d
1d0b: 004190b1400862b0660fdb00
1d0c: 00a190718009623050221d0b
1d0d: 00a19071c0086230700fdb00
1d0e: 0021993100016278c1201d10
1d0f: 00c190b1400062706f0fdb00
1d10: 00c199310000623c61001d11
1d11: 00a119310008627067801d12
1d12: 00a190718009623050001d0d
1d13: 00a190b98006723070001d14
1d14: 00a1b9b38000623171201d16
1d15: 00a19079800e623874401d13
1d16: 082190730000617961209d15
1d17: 00a19079800e623876401d18
1d18: 00a190718000623070201d19
1d19: 08019073000861b56c201d1a
1d1a: 00c19931000962f841201d0d
1d1b: 00a190718000623070209d33
1d1c: 00a1b9b18008627071001d1b
1d1d: 00a119310008627067a01d20
1d1e: 00a190718000623070009d25
1d1f: 00a190718000623070209d2d
1d20: 00a190718009623050001d0d
1d21: 00a190718000623070009d3d
1d22: 082190710001613842201d0d
1d23: 00a19039800662b471701d24
1d24: 004190b10008723062201d26
1d25: 00c1b9b1800862f872501c75
1d26: 00a1b9b38000623171001d28
1d27: 00a190718009623050001d0d
1d28: 00a190718000623070209d55
1d29: 00c19c01000062d065401d2a
1d2a: 00a19931000062306d001d2b
1d2b: 0041d0b1000862b066201de5
1d2c: 0041dc01000062d065601d2e
1d2d: 00a190718009623050001d0d
1d2e: 004190b1000062b066005f00
1d2f: 00c1b9b1800862f872501c75
1d30: 00a1b171800062387e201d31
1d31: 00c19c11000862906cb09d6b
1d32: 084190610000621867401d34
1d33: 00a190718009623050001d0d
1d34: 08a19919000b613069201d35
1d35: 08a19919000b61f861201d36
1d36: 00a199310000627065201d3e
1d37: 00c1b9b1800862f872501c75
1d38: 00a1b031800062787330fe4a
1d39: 00c1b9b1800862f872501c75
1d3a: 00c1b9b1800862f872501c75
1d3b: 00a1b0318000627873101df8
1d3c: 00a1b9b18008623871001d21
1d3d: 00c1b9b1800862f872501c75
1d3e: 00c19071000862386e501d40
1d3f: 00a190718009623050001d0d
1d40: 004190b1000062b066005f00
1d41: 08a19919000b613066201d44
1d42: 00a190010000621061701d22
1d43: 00a1d9310008623866201de5
1d44: 00a190718000623070201d45
1d45: 004190b1000062b066005f00
1d46: 00c1b9b1800862f872501c75
1d47: 00c1b9b1800862f872501c75
1d48: 00a19051000362786d701d49
1d49: 00019041000072e07d505f00
1d4a: 00c1b9b1800862f872501c75
1d4b: 08a19919000b613066001d29
1d4c: 00a190718000623070001d4d
1d4d: 0021993100016230c1001d50
1d4e: 00e1d8b1000062fc67801d23
1d4f: 00c1b9b1800862f872501c75
1d50: 00a190718009623050001d0d
1d51: 00a190718009623050001d0d
1d52: 0ba190710000617467a01d53
1d53: 0021907b0000623489001d54
1d54: 00a190310006629461201d56
1d55: 00c1b9b1800862f872501c75
1d56: 08219073000061b085001d58
1d57: 00a190718009623050001d0d
1d58: 0021b4330000627d69201d59
1d59: 00a1b9b1800862f479209d75
1d5a: 00a190718000623070005f00
1d5b: 00a190010000621061509d65
1d5c: 00a190718000623070201d5d
1d5d: 08219073000961f9c5201d60
1d5e: 00a190718000623070005f00
1d5f: 00a190718000623070005f00
1d60: 00a190718009623050001d0d
1d61: 08f199390000619865201d62
1d62: 00a19971000862fc60401d63
1d63: 00a19071000862fc60609d8d
1d64: 00a190718000623070001d66
1d65: 00a190710008623869501d30
1d66: 08219073000061b56d001f55
1d67: 00a190718000623070005f00
1d68: 00a190718000623070009d9b
1d69: 00a190718000623070001d6a
1d6a: 00a1907b8006623176401d6c
1d6b: 00a190718000623070005f00
1d6c: 00819981000872a076401d6d
1d6d: 08019073000861b56c201d6e
1d6e: 00c19931000062f861001d70
1d6f: 08a19919000b61b866001d32
1d70: 0021993100016230c1001d71

1d71: 00a19310001627047801d0d
1d72: 09f1e041000860d86a009da3
1d73: 088190510003613060201d74
1d74: 08819899000b713860201d76
1d75: 00a1b9b1800862bc71209d7d
1d76: 08819899000b713460001d78
1d77: 00c1b9b1800862f872501c75
1d78: 08819899000b713c60001d79
1d79: 08819899000b71b060201d7a
1d7a: 08819899000b71bc60001d7b
1d7b: 08c198990003713860201d7c
1d7c: 08c198990003713c60201d7e
1d7d: 008190598000722072609d85
1d7e: 08c19899000371b460001d80
1d7f: 00a190b98006723070001d5c
1d80: 08c198990003713460001d81
1d81: 08c19899000371b060001d82
1d82: 08c19899000371b860001d83
1d83: 00419431000062b06c231dae
1d84: 00a190718000623070201d86
1d85: 00a1b9b18000723470201d63
1d86: 0021993100016230c1001d88
1d87: 00a1b9b18008623475001d61
1d88: 00a190718009623050001d89
1d89: 004198b1400872f8660fdb00
1d8a: 00e19931000862bc62201d8b
1d8b: 08a1b033000861716ee01d8c
1d8c: 00a1b0718008623071209c35
1d8d: 00a190718006623070001d56
1d8e: 084190610000621867601d90
1d8f: 00c1b9b1800862f872501c75
1d90: 08a19919000b613069001d91
1d91: 00c19c21000862d06ec01d94
1d92: 0021b93100096278c1201d68
1d93: 00a190718009623050001d0d
1d94: 08a19919000b617c6f001d95
1d95: 00c19071000862386e501d98
1d96: 0021b93100096234c9201d68
1d97: 0021b93100096230c1201d68
1d98: 0041d0b1000162b046001d0d
1d99: 00a190718000623070001d9a
1d9a: 004190b1000962b046001d0d
1d9b: 00a190718009623050001d0d
1d9c: 00c19c11000b62b06cb01d9d
1d9d: 08c190610008621867401d9e
1d9e: 08a19919000b617869201da0
1d9f: 00c1b9b1800862f872501c75
1da0: 00a199398000623079201da1
1da1: 00c19c11000b62b06cb01da2
1da2: 00a190718000623070201da4
1da3: 088199190003617066001d74
1da4: 004190b1000062b066001da5
1da5: 00c19071000162384e701d0d
1da6: 00a190718000623070201da8
1da7: 00c19979800862f471201d73
1da8: 08a1b033000861796ee01da9
1da9: 00a1b0718008627871009ddd
1daa: 00a190718000623070209ded
1dab: 00a1b9710000627060601dac
1dac: 002190b1000062b065209df5
1dad: 084190610000621867401db0
1dae: 08a1989b000b713080001d84
1daf: 08a198990003713060201d88
1db0: 08a19919000b613869001db1
1db1: 00c19c21000862d06ee01db2
1db2: 08a19919000b617c6f001db4
1db3: 08a19919000b613866001d99
1db4: 00c19071000862386e501db5
1db5: 00c1d931000162b045001d0d
1db6: 00819931000862f461201db8
1db7: 00a199398006623072001d8a
1db8: 00219433010062b48d001db9
1db9: 002199b3000862b44d001dba
1dba: 0971907100086ab86d201dbc
1dbb: 00a1b0318000627873309dc2
1dbc: 00c19c310000623c6d001dbd
1dbd: 00c19c310000623c62302008
1dbe: 00819931000862f469001dc0
1dbf: 00a199398006623072001d8a
1dc0: 00c19c310000627c65602008
1dc1: 00e19931000862b461201dc4
1dc2: 00a19011000b627861501d99
1dc3: 002190210000621861509dcd
1dc4: 08a1b033000861716ec01dc5
1dc5: 00a1b0718008623071209e15
1dc6: 00a19011000b627861501d99
1dc7: 00a19011000b627861501d99
1dc8: 00a1b0718008623071201dc9
1dc9: 00a1b0318000627073109e2a
1dca: 00a190718000623070201dcb
1dcb: 002194330008623161201dcc
1dcc: 00a1b0718008623071001dce
1dcd: 00a190710008623469501d9c
1dce: 00a190210008621061509e3a
1dcf: 00a19011000b627861501d99
1dd0: 00a190718000623070009e45
1dd1: 00a1b0718008627871201dd2
1dd2: 00a1b0318000627873309e4a
1dd3: 08a19919000b61b066201dab
1dd4: 00a190718000623070201dd5
1dd5: 002194330008627961001dd6
1dd6: 00a1b0718008627871201dd8
1dd7: 00a19939000e623c62001da6
1dd8: 00a190210008621861709e5a
1dd9: 00a190718000623070209e65
1dda: 00a1b071800862b47d201ddb
1ddb: 00a1b031800062f473309e6a
1ddc: 00a190718000623070001dde

1ddd: 00a190310000621871509de2
1dde: 00219433000862b56d001de0
1ddf: 00a1b9710000627060601daa
1de0: 00a1b071800862b47d201de1
1de1: 002190010008629471509e7a
1de2: 00a19939000e627c6f001da6
1de3: 08a19919000b61b066201dab
1de4: 00a190718000623070009e85
1de5: 00a1b0318000627073101de8
1de6: 00c1b9b1800862f872501c75
1de7: 00c1b9b1800862f872501c75
1de8: 00a190210008621061509e8d
1de9: 00a1b9718008627c76001dea
1dea: 084190610000621867409e93
1deb: 00219021000062506ee01dec
1dec: 00a19c11000b623869001dee
1ded: 002190b9000e627c69001da6
1dee: 0041b0210008625861601ef7
1def: 00c190b10001627049001d0d
1df0: 00a199398000627079001df1
1df1: 00219021000062506ee01df2
1df2: 00a19c11000b623869001df3
1df3: 0041b0210008625861601df4
1df4: 00c19071000862386e501ef7
1df5: 08a19099000b71b060201dab
1df6: 00a1b0318000627873101df8
1df7: 00a190718000623070021dfb
1df8: 00a190210008621861509e9d
1df9: 00a1b9718008627c76201dfa
1dfa: 084190610000621867409ea3
1dfb: 004190b1000172b046001d0d
1dfc: 00219021000062586ec01dfd
1dfd: 00a19c11000b623069001dfe
1dfe: 0041b0210008625861601ef7
1dff: 002190b1000872b065201dad
1e00: 00a199398000627079001e02
1e01: 00a190718000623070002008
1e02: 00219021000062586ec01e04
1e03: 00c199310000627061201dbe
1e04: 00a19c11000b623069201e06
1e05: 00a190718000623070002008
1e06: 0041b0210008625861601e08
1e07: 00819931000862b861201db6
1e08: 00c19071000862386e501ef7
1e09: 00f199310408623461401e0c
1e0a: 00619931000e72fc6f001dc1
1e0b: 00619931000e72fc6f001dc1
1e0c: 00a1b0718008623071001e0d
1e0d: 00a1b0718008627871009eaa
1e0e: 00a19071800062307020200a
1e0f: 00a19071800062307020200a
1e10: 00419971000062f469009f03
1e11: 00a190b98006723070001e12
1e12: 0021943b0006623569001e13
1e13: 0021943b0006627d69201e14
1e14: 08a1b033000861f16ec01e16
1e15: 00e1b9b1800062b471209e1a
1e16: 08a1b033000861f96ee09f0d
1e17: 00a190718000623070002002
1e18: 00a199310000627869001e1c
1e19: 00a199310000623069001e1c
1e1a: 00c1b9b1800862f872501c75
1e1b: 00a190718000623070009e25
1e1c: 00c199710008623862661f2e
1e1d: 00a190398006623871701e20
1e1e: 00c1b9b1800862f872501c75
1e1f: 00c1b9b1800862f872501c75
1e20: 082190730000613569201e21
1e21: 082190730000617d69201e22
1e22: 00c190110008625861601e23
1e23: 00c1b9b1800862b87e201e24
1e24: 002190b1000062f865209f23
1e25: 00c1b9b1800862f872501c75
1e26: 00419971000062b465201e28
1e27: 00a190718000623070002008
1e28: 00a19979800862f075201e29
1e29: 00a190b1000862f061201e2c
1e2a: 00a1b0718008623071202001
1e2b: 002190110000621061509e35
1e2c: 00a19011000b62b061701e2d
1e2d: 00a190b1000862b861001e30
1e2e: 00a1b0718008623071202001
1e2f: 00a1b0718008623071202001
1e30: 00a198b98008723070201e31
1e31: 00a19011000b62b861501e38
1e32: 00a199310000623069201e33
1e33: 00c190710008623862661f2e
1e35: 00a19079800e62b079701dca
1e37: 00a199398006623075001dca
1e3a: 00a1b0718008623071202001
1e3b: 00a1b9b1800862f079201dd0
1e3e: 00a1b0718008623071202001
1e3f: 00a1b0718008623071202001
1e45: 00a1b0318000627073101e2b
1e47: 00a1b0718008623071202001
1e4a: 00a190718000623070002001
1e4b: 002190110000621861709e55
1e4e: 00a190718000623070002002
1e4f: 00a190718000623070002004
1e55: 00a19079800e62b079701dd4
1e57: 00a199398006623075001dd4
1e5a: 00a190718000623070002001
1e5b: 00a1b9b1800862f079201dd9
1e5e: 00a190718000623070002002
1e5f: 00a190718000623070002004
1e65: 00a1b0318000627873301e4b
1e67: 00a190718000623070202003
1e6a: 00a1b071800862b47d202001

1e6b: 00a190310000629461709e75
1e6e: 00a1b071800862b47d202001
1e6f: 00a1b071800862b47d202001
1e75: 00a19079800e62bc79501ddc
1e77: 00a19939800662707d201ddc
1e7a: 00a1b071800862b47d202001
1e7b: 00e1b9b1800062707d001de4
1e7e: 00a1b071800862b47d202001
1e7f: 00a1b071800862b47d202001
1e85: 00a1b031800062f473301e6b
1e87: 00a1b071800862b47d202001
1e8d: 00a190710008623c69501de9
1e8f: 00a1b9b18008623c7c201dea
1e93: 08a19919000b613069201df0
1e97: 00a199398000627079201deb
1e9d: 00a190710008623c69701df9
1e9f: 00a1b9b18008623c7c001dfa
1ea3: 08a19919000b613869001e00
1ea7: 00a199398000627079201dfc
1eaa: 00a190718000623070009eda
1eab: 00a1b0318000627073309eb2
1eae: 00a190718000623070209ee2
1eaf: 00a1b9b18008623871009eea
1eb2: 00a190210008621061509e8d
1eb3: 00a1b0318000627873109ebd
1eb6: 00a190210008621061509e8d
1eb7: 00a190210008621061509e8d
1ebd: 00a1b9b18008623871209ec5
1ebf: 00a1b9718008623871009ecd
1ec5: 00a190718000623070009ed3
1ec7: 00a1b0318000627073101de8
1ecd: 00a1b0318000627873101df8
1ecf: 00a190718000623070009ed3
1ed3: 00a1b0318000627873101df8
1ed7: 00a1b0318000627073101de8
1eda: 00a190398006623071701e1d
1edb: 00a1b0318000627873101df8
1ede: 00c1b9b1800862f872501c75
1edf: 00c1b9b1800862f872501c75
1ee2: 00c1b9b1800862f872501c75
1ee3: 00a1b0318000627873101df8
1ee6: 00a190398006623071701e1d
1ee7: 00c1b9b1800862f872501c75
1eea: 00c1b9b1800862f872501c75
1eeb: 00a1b0318000627873101df8
1eee: 00c1b9b1800862f872501c75
1eef: 0041b0210008625861609ef5
1ef5: 00c1b9b1800862f872501c75
1ef7: 00c1b9b1800862b87e209efd
1efd: 09c190610000615c67801e10
1eff: 00a190718c09623050001d0d
1f00: 00a190718000623070005f00
1f03: 00c19979800e62b879001e11
1f07: 00a19979800e623c79001e11

1f0d: 00a199310000623465009f1d
1f0f: 00a190718000623070209f15
1f15: 00a199310000627c65001c32
1f17: 00a199310000627869001e19
1f1d: 00a199310000627c65201c30
1f1f: 0821b0710000611469201c2a
1f23: 00c19979800862f875001e29
1f27: 094190510008615c67801e26
1f2e: 00f199310408623461401e0c
1f2f: 00a1907180006230702008e1
1f40: 08419073000061348a001d0e
1f41: 00a190718000623070201419
1f42: 0021993100016230c1001d1d
1f44: 00a1b0718008623071207dc9
1f45: 00a1b0318000627073309d5a
1f46: 08a1909b000b617086001d4c
1f47: 00a190b9800e627076001d13
1f48: 00a1b0718008623071207dc9
1f49: 08a199090000611066221dbb
1f4a: 00419c11000062906cb21dd3
1f4b: 00c19079800e62b872401d69
1f4c: 0801b0710008611060007ddb
1f4d: 00a19039800662b471501d64
1f4e: 00c1b9b1800862b87e201d72
1f4f: 08a19919000b613066001d2c
1f50: 00a1b1b18000623871001d1b
1f51: 00a190710008627060401d1c
1f52: 00a1b9b18008623871001d1b
1f53: 00a1b9b18008627071001d1b
1f54: 0801b0710008611060007ddb
1f55: 002190210000629461001d52
1f56: 00a1b9b18008623871031d1e
1f60: 00a199310000623861221db3
1fc0: 00a199110001623061001d0a
1fc1: 000194310001622851201d0d
1fc2: 002194330008627b61005f00
1fc3: 008194010000628b74205f00
1fc4: 002194330008623361005f00
1fc5: 000194010008628b74205f00
1fc6: 00a199110001623061001d0c
1fc7: 000194310001622851221d03
1fc8: 0801b0410008610b74007dc9
1fc9: 00f1d1b1800062f877809d42
1fcc: 0801b0610000614b74207ddb
1fcd: 00a1b031800062f473109d4a
1fce: 008199310008626b71207dc8
1fcf: 00a19931000062b861001c20
1fd0: 002194330008627b61207dc8
1fd1: 00a19931000062b861001c20
1fd2: 0001943140006228760fdb00
1fd3: 0001943140006268715fdb00
1fd4: 0881b0410000610b74201d38
1fd6: 000194310008622871201d41
1fd8: 0021943b0000627b61001d48

1fda: 08a1909b000b617366201d51
1fdc: 002194330008623769201c22

902: 0000000000000000000516
903: 0000000000000000000001
904: 0000000000000000000001
905: 0000000000000000000001
906: 0000000000000000000001
907: 0000000000000000000583
908: 0000000000000000000583
909: 000000000000000000008a
90a: 0000000000000000000516
90b: 0000000000000000000001
90c: 0000000000000000000001
90d: 0000000000000000000001
90e: 0000000000000000000001
90f: 0000000000000000000583
910: 0000000000000000000001
911: 00000000000000000000a4
912: 0000000000000000004d87
913: 0000000000000000000001
914: 0000000000000000000001
915: 0000000000000000000001
916: 0000000000000000000001
917: 0000000000000000000001
918: 0000000000000000000001
919: 00000000000000000000a4
91a: 0000000000000000004d87
91b: 0000000000000000000001
91c: 0000000000000000000001
91d: 0000000000000000000001
91e: 0000000000000000000001
91f: 0000000000000000000001
920: 0000000000000000000583
921: 00000000000000000042839
922: 0000000000000000000520
923: 0000000000000000000001
924: 0000000000000000000001
925: 0000000000000000000001
926: 0000000000000000000001
927: 0000000000000000000583
928: 0000000000000000000583
929: 000000000000000000052c
92a: 0000000000000000000001
92b: 0000000000000000000001
92c: 0000000000000000000001
92d: 0000000000000000000001
92e: 0000000000000000000001
92f: 0000000000000000000583
930: 0000000000000000000583
931: 00000000000000000042839
932: 0000000000000000000508
933: 0000000000000000000001
934: 0000000000000000000001
935: 0000000000000000000001
936: 0000000000000000000001
937: 0000000000000000000583

938: 0000000000000000000583
939: 00000000000000000042839
93a: 0000000000000000000508
93b: 0000000000000000000001
93c: 0000000000000000000001
93d: 0000000000000000000001
93e: 0000000000000000000001
93f: 0000000000000000000583
940: 0000000000000000000583
941: 000000000000000000002f
942: 0000000000000000004d87
943: 0000000000000000000001
944: 0000000000000000000001
945: 0000000000000000000001
946: 0000000000000000000001
947: 0000000000000000000583
948: 0000000000000000000583
949: 000000000000000000002f
94a: 0000000000000000004d87
94b: 0000000000000000000001
94c: 0000000000000000000001
94d: 0000000000000000000001
94e: 0000000000000000000001
94f: 0000000000000000000583
950: 0000000000000000000583
951: 00000000000000000042839
952: 000000000000000000058a
953: 0000000000000000000001
954: 0000000000000000000001
955: 0000000000000000000001
956: 0000000000000000000001
957: 0000000000000000000583
958: 0000000000000000000001
959: 0000000000000000000531
95a: 0000000000000000000001
95b: 0000000000000000000001
95c: 0000000000000000000001
95d: 0000000000000000000001
95e: 0000000000000000000001
95f: 0000000000000000000001
960: 0000000000000000000001
961: 00000000000000000042839
962: 00000000000000000044901
963: 0000000000000000000001
964: 0000000000000000000001
965: 0000000000000000000001
966: 0000000000000000000001
967: 0000000000000000000001
968: 0000000000000000000001
969: 000000000000000000002a
96a: 00000000000000000044992
96b: 0000000000000000000001
96c: 0000000000000000000001
96d: 0000000000000000000001

Appendix B

Instruction Formats

This appendix contains the format of each newly defined VAX instruction which implements a WAM instruction.

allocate
|FD|00|

call L,n
|FD|01|8F|n|8F|L|L|L|L|

cut
|FD|02|

cutd L
|FD|03|8F|L|L|L|L|

deallocate
|FD|04|

execute L
|FD|05|8F|L|L|L|L|

fail
|FD|06|

get_constant c,Xi
|FD|07|8F|c|c|c|c|c|5i|

get_list Xi
|FD|08|5i|

get_nil Xi
|FD|09|5i|

get_structure F,Xi
|FD|0A|8F|F|F|F|F|5i|

get_value Xn,Xi
|FD|0B|5i|5n|

get_value Yn,Xi
|FD|0C|5i|CE|d|d| word displacement
|FD|0C|5i|AE|d| byte displacement

get_variable Xn,Xi
|FD|0D|5i|5n|

get_variable Yn,Xi
|FD|0E|5i|CE|d|d| word displacement
|FD|0E|5i|AE|d| byte displacement

proceed
|FD|0F|

escape_integer
|FD|10|

escape_atom
|FD|11|

escape_gt
|FD|12|

escape_lt
|FD|13|

escape_ge
|FD|14|

escape_le
|FD|15|

trail_X1
|FD|18|

escape_in
|FD|1D|

escape_out
|FD|1E|

trust_me_else fail
|FD|1F|

escape_eq
|FD|20|

escape_neq
|FD|21|

unify_cdr Xi
|FD|22|5i|

unify_cdr Yi
|FD|23|CE|d|d| word displacement
|FD|23|AE|d| byte displacement

unify_constant c
|FD|24|8F|c|c|c|c|

unify_nil

|FD|25|

unify_value Xi

|FD|26|5i|

unify_value Yi

|FD|27|CE|d|d| word displacement

|FD|27|AE|d| byte displacement

unify_variable Xi

|FD|28|5i|

unify_variable Yi

|FD|29|CE|d|d| word displacement

|FD|29|AE|d| byte displacement

unify_void n

|FD|2A|8F|n|n|n|n|n|

reset

|FD|2B|

put_constant c,Xi

|FD|2C|8F|c|c|c|c|c|5i|

put_list Xi

|FD|2D|5i|

put_nil Xi

|FD|2E|5i|

put_structure F,Xi

|FD|2F|8F|F|F|F|F|5i|

retry L

|FD|34|8F|L|L|L|L|

retry_me_else L

|FD|35|8F|L|L|L|L|

switch_on_constant mask

|FD|36|8F|m|

switch_on_structure mask

|FD|37|8F|m|

switch_on_term Lc,Li,Ls

|FD|38|8F|Lc|Lc|Lc|Lc|

|8F|Li|Li|Li|Li|

|8F|Ls|Ls|Ls|Ls|

trust L

|FD|39|8F|L|L|L|L|

try L

|FD|3A|8F|L|L|L|L|

try_me_else L

|FD|3B|8F|L|L|L|L|

put_unsafe_value Yn,Xi

|FD|3C|CE|d|d|5i| word displacement

|FD|3C|AE|d|5i| byte displacement

put_value Xn,Xi

|FD|3D|5n|5i|

put_value Yn,Xi

|FD|3E|CE|d|d|5i| word displacement

|FD|3E|AE|d|5i| byte displacement

put_variable Xn,Xi

|FD|57|5n|5i|

put_variable Yn,Xi

|FD|58|CE|d|d|5i| word displacement

|FD|58|AE|d|5i| byte displacement

escape_length

|FD|59|

is_out

|FD|5A|

escape_univ

|FD|5B|

escape_plus

|FD|5C|

escape_minus

|FD|5D|

is_in

|FD|5E|

Appendix C

Source Code for Utility Routines

This appendix contains the source code for the C and Prolog routines which emulate certain built-in procedures, and the C source code for the WAM to VAX translator.

```
% file "builtin.pro"
%
% Prolog Routines to Emulate the Read, Arg, and Functor Predicates
%

% reads a prolog term, expecting a '.' to end it.
read(X) :- readln, getdata(Y), parse(X,Y,Rem), Rem == [], !.

% build a list consisting of the input tokens. Tokens are atoms, special
% atoms (like ':'-'), variables, and punctuation
getdata(Y) :- gettoken(T), (T == '.'!, Y = []; Y = [T|Y1], getdata(Y1)).

% parse a general prolog term, it can be one of three things:
%   an expression with no or only * and / operators present (factor)
%   an expression which may have + and - in addition to * and / (expr)
%   a compound term (clause)
parse(X,Y,Rem) :- factor(Z,Y,Rem1), expr(X,Rem1,Z,Rem).
parse(X,Y,Rem) :- factor(X,Y,Rem).
parse(X,Y,Rem) :- clauses(X,Y,Rem).

% parse arithmetic expressions only, no clause terms. Used to parse
% within a list or structure, where only arithmetic terms are valid
aparse(X,Y,Rem) :- factor(Z,Y,Rem1), expr(X,Rem1,Z,Rem).
aparse(X,Y,Rem) :- factor(X,Y,Rem).

% expr(X,Y,Z,Rem): X is the returned expression, Y is the current token list,
%                  Z is the expression we have so far, Rem is the remainder of
%                  the list when we are done.
expr(X,[Y|Rest],Z,Rem) :- logicop(Y), factor(B,Rest,Rem1), C =.. [Y,Z,B],
                        expr(X,Rem1,C,Rem), !.
expr(X,[Y|Rest],Z,Rem) :- logicop(Y), factor(B,Rest,Rem),
                        X =.. [Y,Z,B].

% Read a prolog compound clause term, of the form:
%   x :- x,y,z;t ...
% or x,y,z,t;r ...
% Facts (like a(X).) are simple structures and fall under the factor designation
clauses(X,Y,Rem) :- term(A,Y,[B|Rem1]), atomic(B),
                    name(B,":-"), dclauses(C,Rem1,Rem), X =.. [B,A,C], !.
clauses(X,Y,Rem) :- dclauses(X,Y,Rem).

% read a prolog disjunct clause term, of the form:
%   X;Y;D ... where X,Y,and D can be compound
```



```

dclauses(X,Y,Rem) :- cclauses(A,Y,[B|Rem1]), B == ';',
                    dclauses(C,Rem1,Rem), X =.. [B,A,C], !.
dclauses(X,Y,Rem) :- cclauses(A,Y,[B|Rem1]), B == ';',
                    cclauses(C,Rem1,Rem), X =.. [B,A,C], !.
dclauses(X,Y,Rem) :- cclauses(X,Y,Rem).

% read a prolog compound clause term, of the form:
%   x,y,d ... where there are at least two subgoals, or one
%   subgoal followed by a disjunction
cclauses(X,[B|Rest],Rem) :- B == '(', dclauses(A,Rest,[C,D|Rem1]),
                            C == ')', D == ',', cclauses(E,Rem1,Rem),
                            X =.. [D,A,E], !.
cclauses(X,Y,Rem) :- aparse(A,Y,[B|Rem1]), B == ';',
                    cclauses(C,Rem1,Rem), X =.. [B,A,C], !.
cclauses(X,Y,Rem) :- aparse(X,Y,Rem), !.
cclauses(X,[B|Rest],Rem) :- B == '(', dclauses(X,Rest,[C|Rem]), C == ')'.

factor(X,Y,Rem) :- multerm(A,Y,Rem1), zfactor(X,Rem1,A,Rem), !.
factor(X,Y,Rem) :- multerm(X,Y,Rem).

zfactor(X,[Y|Rest],Z,Rem) :- addop(Y), multerm(B,Rest,Rem1),
                            C =.. [Y,Z,B], zfactor(X,Rem1,C,Rem), !.
zfactor(X,[Y|Rest],Z,Rem) :- addop(Y), multerm(B,Rest,Rem),
                            X =.. [Y,Z,B].

multerm(X,Y,Rem) :- modterm(A,Y,Rem1), zmulterm(X,Rem1,A,Rem), !.
multerm(X,Y,Rem) :- modterm(X,Y,Rem).

zmulterm(X,[Y|Rest],Z,Rem) :- multop(Y), modterm(B,Rest,Rem1),
                            C =.. [Y,Z,B], zfactor(X,Rem1,C,Rem), !.
zmulterm(X,[Y|Rest],Z,Rem) :- multop(Y), modterm(B,Rest,Rem),
                            X =.. [Y,Z,B].

modterm(X,Y,Rem) :- term(A,Y,Rem1), zmod(X,Rem1,A,Rem), !.
modterm(X,Y,Rem) :- term(X,Y,Rem).

zmod(X,[Y|Rest],Z,Rem) :- Y == 'mod', term(B,Rest,Rem1),
                        C =.. [Y,Z,B], zmod(X,Rem1,C,Rem), !.
zmod(X,[Y|Rest],Z,Rem) :- Y == 'mod', term(B,Rest,Rem),
                        X =.. [Y,Z,B].

term(X,[LB|Rest],Rem) :- LB == '[', !, getlist(X,Rest,Rem).
term(X,[LP|Rest],Rem) :- LP == '(', !, parse(X,Rest,Rem1), !,
                        Rem1 = [RPIRem], RP == ')'.
term(X,[Atom,LP|Rest],Rem) :- atom(Atom), LP == '(', !, C = [Atom|B],
                        getstruct(B,Rest,Rem), X =.. C.
term(X,[X|Rest],Rest).

getlist([],RB|Rem,Rem) :- RB == ']', !.
getlist([X|Y],Rest,Rem) :- aparse(X,Rest,[Comma|NRem]), Comma == ',', !,
                        getlist(Y,NRem,Rem).
getlist([X],Rest,Rem) :- aparse(X,Rest,[RB|Rem]), RB == ']''.

getstruct(_,RPIRem,Rem) :- RP == ')', !.

```

```

getstruct([X|Y],Rest,Rem) :- aparse(X,Rest,[Comma|NRem]), Comma == ',', !,
    getstruct(Y,NRem,Rem).
getstruct([X],Rest,Rem) :- aparse(X,Rest,[RP|Rem]), RP == ')'.

```

```

addop(Y) :- Y == '+',!.
addop(Y) :- Y == '-',!.

```

```

multop(Y) :- Y == '*',!.
multop(Y) :- Y == '/',!.

```

```

logicop(Y) :- Y == '>',!.
logicop(Y) :- Y == '<',!.
logicop(Y) :- Y == '>=',!.
logicop(Y) :- Y == '<=',!.
logicop(Y) :- Y == '==',!.
logicop(Y) :- Y == '==',!.
logicop(Y) :- Y == 'is'.

```

```

%gettoken(X) :- read(X).           % stdin for C routine.
%readln.                          % stdin for C routine.

```

```

% builtin functions arg/3 and functor/3 implemented in Prolog.
% can be read into compiler after source code to add these utilities
% without paying the price of added microcode

```

```

arg(I,X,Y) :- list(X), I == 1, !, X = [Y|_].
arg(I,X,Y) :- list(X), I == 2, !, X = [_|Y].
arg(I,T,X) :- integer(I), T =.. Y, I2 is I+1, arg1(I2,Y,X).

```

```

arg1(1,[X|_],X) :- !.
arg1(I,[_|Y],X) :- I1 is I-1, arg1(I1,Y,X).

```

```

functor(T,F,N) :- list(T), !, F = '.', N = 2.
functor(T,F,N) :- T =.. [F|X], !, length(X,N).
functor(T,F,N) :- atom(F), integer(N), functor1(N,L), T =.. [F|L].

```

```

functor1(1,[_]) :- !.
functor1(N,[_|L]) :- N1 is N - 1, functor1(N1,L).

```

```

/* escape.h */

/* declarations and data structures used by escape.c, a group of
 * C functions which handle certain Prolog built-in predicates
 */

#include <stdio.h>

#define unix 1
#ifdef vms
#   include <types.h>
#   include ssdef
#   include descrip
#else
#   include <sys/types.h>
#endif

#define HEAPSIZ 4000000 /* size of allocated heap */
#define CMASK 0x03ffffff /* masks out constant tag */
#define TRAILPRE 0x7fff0000 /* OR to create trail pointer */

/* C escape routines will place heap and trail increments in the following
 addresses for certain routines which make bindings, etc. Currently these
 routines are name_2, retract_1, and access_3 */
#define TRAILINC 0x7fff0010
#define HEAPINC 0x7fff0014

/* macros for bit manipulation of Prolog data elements */
#define signextend(x) (((x >> 27) & 1) ? (0x70000000 + (x & 0x0ffffff)) : (x & 0x0ffffff))
#define cdr(x) (x & 0x20000000)
#define nil(x) ((x & 0xcc000000) == 0xcc000000)
#define list(x) ((x >> 30) == 0)
#define structure(x) ((x >> 30) == 1)
#define var(x) ((x >> 30) == 2)
#define const(x) ((x >> 30) == 3)
#define atom(x) ((x >> 26) == 0x32) /* tag for atom: 110010 */
#define number(x) ((x >> 26) == 0x30) /* tag for int: 110000 */
#define numeric(x) ((x >= '0') && (x <= '9'))
#define lowercase(x) ((x >= 'a') && (x <= 'z'))
#define uppercase(x) ((x >= 'A') && (x <= 'Z'))
#define alphanumeric(x) (numeric(x) || lowercase(x) || uppercase(x))
#define special(x) ((x >= '!') || (x <= '~'))

/* macros which check if a structure functor is arithmetic for printing */
#define add(x) ((x[0] == '+') || (x[0] == '-'))
#define mult(x) ((x[0] == '/') || (x[0] == '*') || (x[0] == '^'))

/* global variables */
FILE *outfile,*infile; /* for see, seen, tell, told, get, put, write */
int staticatom; /* count of static atoms, set by init() */
int dynamicatom; /* index of next available dynamic atom */
int varcount; /* count of variables found in current read */
extern char *atomlist[]; /* external reference to WAM code atoms */
char dynamiclist[100][80]; /* stores atoms created dynamically by "name" */

```

```
struct {
    char str[80];      /* stores variables found during read */
    unsigned int val; /* in case they recur in the term */
} varlist[20];       /* recurring vars bound to same place */
/* can support twenty of these */
char string[80];    /* to support read(X) */
int ptr;            /* to support read(X) */

/* forward definitions for non-integer functions */
unsigned int deref();
```

```

/* escape.c */

/*
 * C subroutines which implement several of the Prolog built-in functions.
 * The main escape routines are prefaced by "plm" with the arity of the
 * built-in as the suffix. For example, to execute the write(X) built-in
 * the plm_write_1(X) subroutine is called.
 */

#include "escape.h"

/***** Utilities used by the Main Escape Routines *****/

/*
 * argwrite: writes a specific argument of a structure
 *
 */
argwrite(ptr,arg,nest,lst)
unsigned int *ptr;
int arg,nest,lst;
{
    if (cdr(*ptr))
        ptr = (unsigned int *) signextend(*ptr);
    while (arg) {
        ptr++;
        if (cdr(*ptr))
            ptr = (unsigned int *) signextend(*ptr);
        arg--;
    }
    dowrite(*ptr,nest,lst);
}

/*
 * copylist: handles simple case of plm_name_2, where an atom is turned into
 * a string list, i.e. name(atom,Var).
 */
copylist(atum,name,heap)
unsigned int atum,name,*heap;
{
    int offset, heapgrowth = 0;
    char *str;

    offset = atum & CMASK;
    /* check if atom is static (in code) or dynamic (created on fly) */
    if (offset < staticatom)
        str = atomlist[offset];
    else
        str = dynamiclist[offset - staticatom];
    /* bind variable 'name' to a listpointer to the heap */
    *((unsigned int *)signextend(name)) = (unsigned int) heap;
    while (*str != ' ') {
        heapgrowth += 4;
    }
}

```

```

        *heap++ = (*str) | 0xc0000000;
        str++;
    }
    *heap = 0xefffffff; /* finish list with a NIL */
    return(heapgrowth+4); /* account for NIL */
}

/*
 * deref: dereferences prolog data word x and returns the dereferenced
 * value of x.
 */
unsigned int deref(x)
unsigned int x;
{
    for (;;) {
        switch (x >> 30) {
            case 0: /* list */
            case 1: /* structure */
            case 3: /* constant */
                return(x);
            case 2: /* variable */
                x = signextend(x);
                if (x == signextend*((unsigned int*) x))
                    return*((unsigned int *) x); /* unbound */
                x = *(unsigned int *) x; /* bound: loop */
                break;
        }
    }
}

/*
 * dowrite: essentially performs the functions of the write(X) predicate
 * except that a nest flag is incorporated to
 * print arithmetic expressions nicely. If nest is set, then dowrite
 * is being called recursively with a parent functor of * or /,
 * thus any chile + or - structures must be surrounded by brackets.
 * Nest is set when * and / functors are found and cleared when
 * + and - functors are found.
 */
dowrite(x,nest,lst)
unsigned int x;
int nest,lst;
{
    unsigned int functor;
    char *ptr;

    switch (x >> 30) {
        case 0: /* list */
            x = signextend(x);
            fprintf(outfile,"["");
            printlist((unsigned int*) x);
            fprintf(outfile,"]");
            break;
        case 1: /* structure */

```

```

x = signextend(x);
functor = (*(unsigned int *) x) & CMASK;
if (functor < staticatom)
    ptr = atomlist[functor];
else
    ptr = dynamiclist[functor - staticatom];
switch (ptr[0]) {
case '-':
case '+': /* write with paren if nested */
    if (nest) fprintf(outfile, "(");
    argwrite((unsigned int *) (x + 4), 0, 0, 0);
    printconstant(*(unsigned int *) x);
    argwrite((unsigned int *) (x + 4), 1, 0, 0);
    if (nest) fprintf(outfile, ")");
    fflush(outfile);
    break;

case '^':
case '**':
case '/':
case '=': /* don't need paren, but nested */
    argwrite((unsigned int *) (x + 4), 0, 1, 0);
    printconstant(*(unsigned int *) x);
    argwrite((unsigned int *) (x + 4), 1, 1, 0);
    fflush(outfile);
    break;

case ':':
case ';':
    argwrite((unsigned int *) (x + 4), 0, 0, lst);
    printconstant(*(unsigned int *) x);
    argwrite((unsigned int *) (x + 4), 1, 0, lst);
    fflush(outfile);
    break;

case ',':
    if (lst) fprintf(outfile, "(");
    argwrite((unsigned int *) (x + 4), 0, 0, 0);
    printconstant(*(unsigned int *) x);
    argwrite((unsigned int *) (x + 4), 1, 0, 0);
    if (lst) fprintf(outfile, ")");
    fflush(outfile);
    break;

default:
    if (strcmp(ptr, "mod") == 0) {
        argwrite((unsigned int *) (x + 4), 0, 1, 0);
        fprintf(outfile, " ");
        printconstant(*(unsigned int *) x);
        fprintf(outfile, " ");
        argwrite((unsigned int *) (x + 4), 1, 1, 0);
        fflush(outfile);
        break;
    } else {
        printconstant(*(unsigned int *) x);
        fprintf(outfile, "(");
        fflush(outfile);
        printlist((unsigned int *) (x + 4));
    }
}

```

```

                fprintf(outfile,"");
                fflush(outfile);
                break;
            }
        }
        break;
    case 2:                /* variable */
        x = signextend(x);
        if (x == signextend*((unsigned int*) x)) {
            fprintf(outfile,"_%x",*(unsigned int*)x & 0xffff);
        } else {
            dowrite*((unsigned int*) x, nest, lst);
        }
        break;
    case 3:                /* constant */
        printconstant(x); /* constants never need nest or list */
        break;
    }
    return(1);
}

/*
 * init: called by main() before any WAM instructions are executed.
 * Prints message, sets default I/O files, and initializes heap space
 */
init()
{
    printf("VAX 8600 PLM, Version 1.00);
    outfile = stdout;
    infile = stdin;

    /* get count of how many atoms we have statically */

    for (staticatom = 0; *atomlist[staticatom] != 0; staticatom++);
    dynamicatom = staticatom; /* next available index */
    return((int) malloc(HEAPSIZE));
}

/*
 * matchlist: called by plm_name_2 to handle the case where name is called
 * with a bound atom and list. The string representing the atom
 * is unified with the list. The difficulty here is that all
 * bindings made must be trailed, as a fail should undo them.
 */
matchlist(atum, name, trailptr)
unsigned int atum, *name, *trailptr;
{
    int offset, trailinc = 0;
    char *str;
    unsigned int data;

    offset = atum & CMASK;
    if (offset < staticatom)
        str = atomlist[offset];

```



```

else
    str = dynamiclist[offset-staticatom];
while (*str != ' ') {
    if (nil(*name)) {
        return(0);
    } else if (cdr(*name) && (var(*name))) {
        return(0);
    } else if (cdr(*name)) {
        name = (unsigned int *) signextend(*name);
    } else {
        data = *name++;
        data = deref(data);
        if (number(data)) {
            if (!(char)(data & 0xff) == *str++)
                return(0);
        } else if (var(data)) {
            *(unsigned int *) signextend(data) =
                ((unsigned int) *str++) | 0xc0000000;
            *trailptr++ = data;
            trailinc++;
        } else
            return(0);
    }
}
if (nil(*name))
    return(trailinc+1); /* at least 1, for success */
else
    return(0);
}

/*
 * nest: evaluates possibly nested structures in "is_2" statements
 */
unsigned int nest(x)
unsigned int x;
{
    unsigned int *structptr;
    unsigned int result, val1, val2;
    char *operation;
    int index;

    x = deref(x);

    if (x >> 26 == 0x30) { /* just return any integers */
        return(x);
    }
    else if (!structure(x)) /* else fail if not structure */
        return(0);

    structptr = (unsigned int *) signextend(x);
    index = structptr[0] & CMASK;
    if (index < staticatom)
        operation = atomlist[index];
    else

```

```
operation = dynamiclist[index - staticatom];
```

```
/* now lets evaluate operands of the functor, but only if the
   functor is a valid operation */
```

```
switch (operation[0]) {
    case 'm': if (strcmp(operation,"mod"))
                return(0);
    case '+':
    case '-':
    case '*':
    case '/':
                if (!(val1 = nest(structptr[1])))
                    return(0);
                if (!(val2 = nest(structptr[2])))
                    return(0);
                break;
    default:
                return(0);
}
```

```
/* now we have operands, lets get the result */
```

```
switch (operation[0]) {
    case 'm':
                result = 0xc0000000 +
                (CMASK & ((CMASK & val1) % (CMASK & val2)));
                return(result);
                break;
    case '+':
                result = 0xc0000000 +
                (CMASK & ((CMASK & val1) + (CMASK & val2)));
                return(result);
                break;
    case '-':
                result = 0xc0000000 +
                (CMASK & ((CMASK & val1) - (CMASK & val2)));
                return(result);
                break;
    case '*':
                result = 0xc0000000 +
                (CMASK & ((CMASK & val1) * (CMASK & val2)));
                return(result);
                break;
    case '/':
                result = 0xc0000000 +
                (CMASK & ((CMASK & val1) / (CMASK & val2)));
                return(result);
                break;
    default: /* this shouldn't happen */
                return(0);
}
}
```

```
/*
```

```
* printconstant: used by plm_write_1 to print a character constant
```

```

*/
printconstant(x)
unsigned int x;
{
    int index;

    switch ((x >> 26) & 3) {
    case 0:
        fprintf(outfile,"%d", (x & CMASK));
        break;
    case 1:
        fprintf(outfile,"%f",*((float *) (x & CMASK)));
        break;
    case 2:
        index = x & CMASK;
        if (index < staticatom)
            fprintf(outfile,"%s",atomlist[(index)]);
        else
            fprintf(outfile,"%s",dynamiclist[(index-staticatom)]);
        break;
    case 3:
        fprintf(outfile,"[]");
        break;
    }
}

```

```

/*
* printlist: used by plm_write_1 to print out a list. Recursive: calls
* plm_write_1 again to print each element of the list
*/

```

```

printlist(x)
unsigned int *x;
{
    int first;
    unsigned int y;

    first = 1;
    for (;;) {
        if (cdr(*x)) {
            y = deref(*x);
            if (nil(y)) {
                return;
            } else if (var(y)) {
                fprintf(outfile,"l_%x",y & 0x0fffffff);
                return;
            } else if (list(y)) {
                x = (unsigned int *) signextend(y);
            } else {
                fprintf(outfile,"l");
                dwrite(y,0,1);
                return;
            }
        } else {
            if (!first) {

```

```

        fprintf(outfile,");
    } else {
        first = 0;
    }
    dwrite(*x,0,1); /* list is on */
    x++;
}
}

/*
 * searchtable: used by plm_name_2 to handle case where list is
 * instantiated and the atom is a var to be bound.
 * In this case the list must not contain any variables
 * and must consist only of integer constants.
 */
searchtable(atum,name)
unsigned int atum,*name;
{
    char str[80],*temp;
    unsigned int data;
    int i;

    /* convert list pointed to by 'name' into a ascii string */
    temp = str;
    for (;;) {
        if (nil(*name)) {
            *temp = ' ';
            break;
        } else if (cdr(*name) && (var(*name))) {
            return(0);
        } else if (cdr(*name)) {
            name = (unsigned int *) signextend(*name);
        } else {
            data = *name++;
            data = deref(data);
            if (!number(data))
                return(0);
            else *temp++ = (char) (data & 0xff);
        }
    }
    /* now scan for all atoms in the table */
    for (i = 0; atomlist[i][0] != 0; i++) {
        if (strcmp(atomlist[i],str) == 0) {
            *(unsigned int *) signextend(atum) = 0xc8000000 + i;
            return(1);
        }
    }
    /* not found, must create new atom */
    i = dynamicatom - staticatom; /* get proper index into dynamiclist */
    strcpy(dynamiclist[i],str); /* make new entry */
    *(unsigned int *) signextend(atum) = 0xc8000000 + dynamicatom++;
    return(1);
}

```

```

}

/*
 * writeno: called if end result is a fail
 */
writeno()
{
    printf("0o0);
    return;
}

/*
 * writeyes: called if end results is a success
 */
writeyes()
{
    printf("0es0);
    return;
}

/***** Main Escape Routines *****/

/*
 * get: get a character from the current input file and unify it's ascii
 *      value with x
 */
plm_get_1(x)
unsigned int x;
{
    int c;

    c = fgetc(infile);      /* get the character regardless */
    while (c == '0)
        c = fgetc(infile);
    c = c | 0xc0000000;     /* convert it to a constant */

    x = deref(x);          /* dereference x */
    switch (x >> 30) {     /* what is x? */
    case 0:                /* list and structures fail */
    case 1:
        return(0);
    case 2:                /* variable */
        x = signextend(x); /* get its address */
        *(unsigned int *)x = c; /* bind x to c, x has been trailed */
        return(1);
    case 3:                /* constant */
        if (x == c) {
            return(1);
        } else {
            return(0);
        }
    }
    break;
}

```

```

    }
    return(0);
}

/*
 * gettoken: instantiates variable represented by tokenptr to the next
 *           token in the input string. Tokens can be atoms, variables,
 *           and punctuation combining to form a valid Prolog term.
 */
plm_gettoken_1(tokenptr)
unsigned int tokenptr;
{
    char temp[80];
    int i = 0, j;

    /* skip blanks in the input */
    while ((string[ptr] == ' ') || (string[ptr] == '0')) ptr++;

    /* handle integers first, convert string to a value and
       bind the variable parameter to a Prolog numeric atom */
    if (numeric(string[ptr])) {
        temp[i++] = string[ptr++];
        while (numeric(string[ptr]))
            temp[i++] = string[ptr++];
        temp[i] = ' ';
        *(unsigned int *) signextend(tokenptr) =
            0xc0000000 + atoi(temp);
        return(1);
    }

    /* handle variables next, check if variable token has been seen
       previously and bind this token to previous token if true, else
       do nothing as token
       is already instantiated to a variable */
    else if (uppercase(string[ptr]) || (string[ptr] == '_')) {
        temp[i++] = string[ptr++];
        while (alphanumeric(string[ptr]))
            temp[i++] = string[ptr++];
        temp[i] = ' ';
        for (i = 0; i < varcount; i++) {
            if (strcmp(varlist[i].str, temp) == 0) {
                *(unsigned int *) signextend(tokenptr) =
                    varlist[i].val;
                return(1);
            }
        }
        /* otherwise var has not been seen */
        strcpy(varlist[varcount].str, temp);
        varlist[varcount++].val = tokenptr;
        return(1);
    }

    /* handle atoms last */
    else {
        if (lowercase(string[ptr])) {
            temp[i++] = string[ptr++];

```

```

    while (alphanumeric(string[ptr]))
        temp[i++] = string[ptr++];
    temp[i] = ' ';
}
else if (string[ptr] == '"') {
    ptr++;
    while (string[ptr] != '"')
        temp[i++] = string[ptr++];
    ptr++;
    temp[i] = ' ';
}
else if (special(string[ptr])) {
    switch(string[ptr]) {
    case '-': if ((string[ptr+1] == '-') &&
                (string[ptr+2] == '>')) {
                strcpy(temp, "-->");
                ptr += 3;
            } else {
                temp[0] = '-';
                temp[1] = ' ';
                ptr++;
            }
            break;
    case ':':
    case '?': if (string[ptr+1] == '-') {
                temp[0] = string[ptr];
                temp[1] = '-';
                temp[2] = ' ';
                ptr += 2;
            } else {
                temp[0] = string[ptr];
                temp[1] = ' ';
                ptr++;
            }
            break;
    default: temp[0] = string[ptr];
            temp[1] = ' ';
            ptr++;
            break;
    }
} else return(0);

/* get atom value by searching atomlist */
for (i = 0; atomlist[i][0] != 0; i++) {
    if (strcmp(atomlist[i],temp) == 0) {
        *(unsigned int *) signextend(tokenptr) =
            0xc8000000 + i;
        return(1);
    }
}
/* didn't find atom, check dynamic list */
j = dynamicatom - staticatom;
for (i = 0; i < j; i++) {
    if (strcmp(dynamiclist[i],temp) == 0) {

```

```

        *(unsigned int *) signextend(tokenptr) =
            0xc8000000 + staticatom + i;
        return(1);
    }
}
/* didn't find again, make new atom */
strcpy(dynamiclist[j],temp);
*(unsigned int *) signextend(tokenptr) =
    0xc8000000 + dynamicatom++;
return(1);
}
}

/*
 * plm_is_2: escape function to evaluate structured "is" statements.
 *
 * The first parameter is the
 * value in X1 and can be one of two things: an unbound var, in which case
 * the result is stored at that location; or a constant, which is unified
 * with the result of the "is" function.
 *
 * The second parameter is the value in X2 and should be a structure.
 * The structure functor should be an atom which identifies the operation.
 * Operands follow the functor.
 *
 * This structptr returns 1 in r0 upon success and 0 upon failure.
 */
plm_is_2(x2,x1)
unsigned int x1, x2;
{
    unsigned int *structptr, *dest;
    unsigned int result, nest();
    char *operation;
    int index;

    /* get value of expression in x2 */
    x2 = deref(x2);
    if ((result = nest(x2)) == 0)
        return(0); /* bad expression */

    /* unify value with x1 */
    x1 = deref(x1);
    if (!var(x1)) { /* not a var, unify */
        if (result ^ x1) { /* Bitwise XOR, false if equal */
            return(0); /* not equal */
        }
        else
            return(1); /* equal */
    } else { /* var, assignment only */
        dest = (unsigned int*) signextend(x1);
        *dest = result;
        return(1); /* var has been trailed */
    }
}

```



```

/*
 * name: unifies ascii string of list "name" with ascii string representing
 * atom "atum". Returns two values: in 7fee0010 the value to increment
 * the trail register by, in 7fee0014 the value to increment the
 * heap pointer by. This value should be a multiple of 4.
 */
plm_name_2(trail,heap,name,atum)
unsigned int trail, *heap, name, atum;
{
    int flag;
    unsigned int *trailptr, *inc_count;

    inc_count = (unsigned int *) TRAILINC;          /* store # of trails */
    atum = deref(atum);
    name = deref(name);
    /* create trail ptr */
    trailptr = (unsigned int *) ((trail >> 16) | TRAILPRE);
    if (atom(atum) && var(name)) {
        *trailptr = name;          /* trail new list */
        *inc_count = 0x00040000;   /* inc trail by 1 */
        flag = copylist(atum,name,heap); /* create new list */
        inc_count = (unsigned int *) HEAPINC; /* store heap offset */
        *inc_count = flag; /* count of written bytes on heap */
        return(1);
    } else if (var(atum) && list(name)) {
        if (searchtable(atum,signextend(name))) {
            *trailptr = atum;      /* trail new atom */
            *inc_count = 0x00040000; /* inc trail by 1 */
            inc_count = (unsigned int *) HEAPINC;
            *inc_count = 4;
            return(1);
        } else
            return(0);
    } else if (atom(atum) && list(name)) {
        flag = matchlist(atum,signextend(name),trailptr);
        if (flag) { /* flag = # of bindings+1 */
            *inc_count = 0x00040000 * (flag - 1);
            inc_count = (unsigned int *) HEAPINC;
            *inc_count = 4;
            return(1);
        } else
            return(0);
    } else
        return(0);
}

/*
 * nl: writes a newline to the current output file
 */
plm_nl_00
{
    fprintf(outfile,"0");
    return(1);
}

```

```

/*
 * put: writes out a character to the current output file. The character
 * must be expressed as an integer constant representing an ASCII value
 */
plm_put_1(x)
unsigned int x;
{
    x = deref(x);
    switch (x >> 30) {
    case 0:          /* fails for lists, structures, and unbound variables */
    case 1:
    case 2:
        return(0);
    case 3:          /* constant */
        if (x & 0x0c000000) {
            fprintf(stderr,"Out: not an integer0);
            return(0);
        } else {
            fprintf(outfile,"%c", x & CMASK);
        }
        break;
    }
    return(1);
}

```

```

/*
 * readln: assists in emulating read(X) by reading the next
 * Prolog term (ending in '.') as a string.
 */
plm_readln_00
{
    char c;
    int i,j;

    if (infile == stdin)
        printf("0:");

    /* Read in the term, ended by a '.' */

    for (i = 0, c = getc(infile);; i++) {
        if ((c == '0') && (infile == stdin))    printf("l:");
        string[i] = c;
        c = getc(infile);
        if ((string[i] == '.') && ((c == ' ') || (c == '0')))
            break;
    }
    ptr = 0;
    varcount = 0;
    return(1);
}

```

```

/*
 * see: sets the atom represented by fvar to the current input file

```

```

*/
plm_see_1(fvar)
unsigned int fvar;
{
    char *fname;
    int index;

    fvar = deref(fvar);
    if (!atom(fvar))
        return(0);
    index = fvar & CMASK;
    if (index < staticatom)
        fname = atomlist[(index)];
    else
        fname = dynamiclist[(index-staticatom)];
    infile = fopen(fname,"r");
    return(1);
}

/*
 * seen: sets the current output file back to stdout
 */
plm_seen_00
{
    if (infile == stdin) {
        printf("Seen: input is stdin0);
        return(0);
    }
    fclose(infile);
    infile = stdin;
    return(1);
}

/*
 * system: convert list into an ascii string and use routine as in
 *         C-Prolog
 */
plm_system_1(command)
unsigned int command;
{
    unsigned int *string, data;
    char commandstring[256], *tmp;

#ifdef vms
    struct dsc$descriptor_s s_d;
#endif
    data = deref(command);    /* should be list of ascii codes */
    if (!list(data))
        return(0);          /* else exit with fail */
                            /* set up pointer to ascii list */
    string = (unsigned int *) signextend(data);
    tmp = commandstring;    /* set up pointer to create string */
    for (;;) {
        if (nil(*string)) {

```

```

        *tmp = ' ';
        break;
    } else if (cdr(*string) && (var(*string))) {
        return(0);
    } else if (cdr(*string)) {
        string = (unsigned int *) signextend(*string);
    } else {
        data = *string++;
        data = deref(data);
        if (!number(data))
            return(0);
        else *tmp++ = (char) (data & 0xff);
    }
}
#endif
#ifdef unix
    system(commandstring);
#endif
#ifdef vms
    s_d.dsc$w_length = strlen(commandstring);
    s_d.dsc$b_dtype = DSC$K_DTYPE_T;
    s_d.dsc$b_class = DSC$K_CLASS_S;
    s_d.dsc$a_pointer = commandstring;
    lib$spawn(&s_d);
#endif
    return(1);
}

/*
 * tab: writes sp spaces to the current output file
 */
plm_tab_1(sp)
unsigned int sp;
{
    int i,count;

    sp = deref(sp);
    if (!number(sp))
        return(0);
    count = (sp & CMASK);
    for (i=0; i< count; i++)
        fprintf(outfile, " ");
    return(1);
}

/*
 * tell: sets the atom represented by fvar to the current output file.
 */
plm_tell_1(fvar)
unsigned int fvar;
{
    char *fname;
    int index;

    fvar = deref(fvar);

```

```

    if (!atom(fvar))
        return(0);
    index = fvar & CMASK;
    if (index < staticatom)
        fname = atomlist[(index)];
    else
        fname = dynamiclist[(index-staticatom)];
    outfile = fopen(fname,"w");
    return(1);
}

/*
 * told: sets the current output file back to stdout
 */
plm_told_0()
{
    if (outfile == stdout) {
        printf("output is already stdout");
        return(0);
    }
    fclose(outfile);
    outfile = stdout;
    return(1);
}

/*
 * write: used dwrite() to write out the prolog data item
 * represented by x to the current output file.
 * The third parameter of dwrite() helps to reduce the number of
 * parentheses to a minimum when printing a arithmetic expression.
 * The second parameter signifies whether certain structures
 * such as "x,y" should be enclosed within parentheses (if
 * it is within another structure or list). Initially these
 * flags are set to zero, since we are printing from the root of
 * the expression.
 */
plm_write_1(x)
unsigned int x;
{
    dwrite(x,0,0); /* set nest, list to zero, printing from the root */
    return(1);
}

```

```
/* plmas.h */
```

```
/* C header file for the PLM to VAX assembler */
```

```
/* definitions for opcode values for the opcode table */
```

```
#define ALLOCATE      0
#define CALL         1
#define CUT          2
#define CUTD         3
#define DEALLOCATE   4
#define EXECUTE      5
#define FAIL         6
#define GET_CONSTANT 7
#define GET_LIST     8
#define GET_NIL      9
#define GET_STRUCTURE 10
#define GET_VALUE    11
#define GET_VARIABLE 13
#define PROCEED      15
#define ESCAPE_INTEGER 16
#define ESCAPE_ATOM   17
#define ESCAPE_GT     18
#define ESCAPE_LT     19
#define ESCAPE_GE     20
#define ESCAPE_LE     21
#define TRAIL_X1      24
#define ESC_IN        29
#define ESC_OUT       30
#define TRUST_ME_ELSE 31
#define ESCAPE_EQ     32
#define ESCAPE_NEQ    33
#define UNIFY_CDR      34
#define UNIFY_CONSTANT 36
#define UNIFY_NIL     37
#define UNIFY_VALUE    38
#define UNIFY_VARIABLE 40
#define UNIFY_VOID     42
#define RESET         43
#define PUT_CONSTANT  44
#define PUT_LIST      45
#define PUT_NIL       46
#define PUT_STRUCTURE 47
#define RETRY         52
#define RETRY_ME_ELSE 53
#define SWITCH_ON_CONSTANT 54
#define SWITCH_ON_STRUCTURE 55
#define SWITCH_ON_TERM 56
#define TRUST         57
#define TRY           58
#define TRY_ME_ELSE   59
#define PUT_UNSAFE_VALUE 60
#define PUT_VALUE     61
#define PUT_VARIABLE  87
#define ESCAPE_LENGTH 89
```

```

#define IS_OUT          90
#define ESCAPE_UNIV    91
#define PLUS           92
#define MINUS          93
#define IS_IN          94

/* definitions for operand patterns */
#define NONE 50          /* no operands */
#define LABEL_OP 51     /* one label operand */
#define XI_OP 52        /* one Xi operand */
#define CONST_OP 53     /* one constant operand */
#define N_OP 54         /* one n operand */
#define FAIL_OP 55      /* a "fail" operand */
#define XYI_OP 56       /* Xi or Yi */
#define CONST_XI 57     /* a constant and then an Xi */
#define FUNCT_XI 58     /* a functor and then an Xi */
#define CMASK_LABEL 59  /* a mask and then a label, then a table */
#define SMASK_LABEL 66  /* a mask and then a label, then a table */
#define XYN_XI 60       /* Xn or Yn, then Xi */
#define YN_XI 61        /* Yn and then Xi */
#define XYN_XI_REV 62   /* Xn or Yn, then Xi, reverse for output */
#define LABEL_N_REV 63  /* a label and then n, reverse for output */
#define LABEL_LABEL_LABEL 64 /* three label operands */
#define TWO_XI 65       /* two operands, either Xi,Yi, or N */
                        /* followed by Xi */

```

```

struct {
    char *instruction;
    int opcode;
    int operandpattern;
} optable[] = {
    "allocate", ALLOCATE,          NONE,
    "call",     CALL,             LABEL_N_REV,
    "cut",      CUT,              NONE,
    "cutd",     CUTD,             LABEL_OP,
    "deallocate", DEALLOCATE,     NONE,
    "execute",  EXECUTE,         LABEL_OP,
    "fail",     FAIL,            NONE,
    "get_constant", GET_CONSTANT,  CONST_XI,
    "get_list", GET_LIST,        XI_OP,
    "get_nil",  GET_NIL,         XI_OP,
    "get_structure",GET_STRUCTURE,  FUNCT_XI,
    "get_value",GET_VALUE,       XYN_XI_REV,
    "get_variable", GET_VARIABLE,  XYN_XI_REV,
    "proceed",  PROCEED,         NONE,
    "put_constant", PUT_CONSTANT,  CONST_XI,
    "put_list", PUT_LIST,        XI_OP,
    "put_nil",  PUT_NIL,         XI_OP,
    "put_structure",PUT_STRUCTURE,  FUNCT_XI,
    "put_unsafe_value",PUT_UNSAFE_VALUE,  YN_XI,
    "put_value", PUT_VALUE,       XYN_XI,
    "put_variable", PUT_VARIABLE,  XYN_XI,
    "retry",    RETRY,            LABEL_OP,
    "retry_me_else",RETRY_ME_ELSE,  LABEL_OP,

```

```

"switch_on_constant",SWITCH_ON_CONSTANT,  CMASK_LABEL,
"switch_on_structure",SWITCH_ON_STRUCTURE, SMASK_LABEL,
"switch_on_term",SWITCH_ON_TERM, LABEL_LABEL_LABEL,
"trust",    TRUST,          LABEL_OP,
"trust_me_else",TRUST_ME_ELSE,    FAIL_OP,
"try",     TRY,           LABEL_OP,
"try_me_else",  TRY_ME_ELSE,    LABEL_OP,
"unify_cdr",UNIFY_CDR,        XYI_OP,
"unify_constant",UNIFY_CONSTANT,  CONST_OP,
"unify_nil" ,UNIFY_NIL,        NONE,
"unify_value",  UNIFY_VALUE,     XYI_OP,
"unify_unsafe_value",UNIFY_VALUE, XYI_OP,
"unify_variable",UNIFY_VARIABLE, XYI_OP,
"unify_void",   UNIFY_VOID,      N_OP,
"reset",    RESET,            NONE,
"escape",   ESC_IN,          NONE,
"plus",     PLUS,            TWO_XI,
"minus",    MINUS,           TWO_XI,
"trail_x1", TRAIL_X1,        NONE,
"",         0,               0,
};

```



```

/* plmas.c */

/*
 * converts one or more Warren Abstract Machine files into VAX assembly
 * language files
 */

#include <stdio.h>
#include "plmas.h"

/* Definitions for token types to be found in the input stream.
 */
#define INSTRUCTION 11
#define ESCAPE 12
#define LABEL 13
#define PROCEDURE 14
#define END 15

/* Definition of a token structure. A token is a sequence of printable
 * characters found in the input stream in between spaces, newlines, tabs
 * or other non-printable characters. The maximum token length is MAXTOK-1.
 */
#define MAXTOK 81
typedef struct {
    int type;
    char string[MAXTOK];
} token;

/* Symbol table structure. This structure is updated when atoms are found
 * in the input stream and then at the end of the output file, the symbol
 * table is inserted.
 */
struct symbol {
    char string[MAXTOK];
    struct symbol *next;
} *symboltable;

/* definition for switch_on_constant and switch_on_structure tables */
struct {
    int index;
    char string[MAXTOK];
} table_array[256];

/* global variables */
FILE *infile; /* the input file */
FILE *outfile; /* the file for sending the assembler output */
int inputlinenumber = 1; /* line count of input for printing errors */
int errorcount = 0; /* number of errors found so far */
#define MAXERROR 10 /* maximum number of errors before quitting */
int repeatcount = 1; /* number of times to repeat the plm program */
int morefiles; /* flag signalling more files to be assembled */
/* determines when to output dummy allocate */

```

```

/* forward definitions for non-integer functions */
token gettoken();
unsigned int getconstantvalue();

main(argc,argv)
int argc;
char **argv;
{
    char **destfile; /* finds the output file, the last parameter */
    int argcount;

    /* note the command line syntax is:
    *   plmas <infile> [<infiles>] <outfile>
    */
    switch (argc) {
    case 1:
    case 2:
        fprintf(stderr,"usage: plmas <input file(s)> <output file>0);
        exit();
    default: /* open the destination file */
        argcount = argc;
        destfile = argv;
        while (--argcount) destfile++; /* destination file is last */
        if ((outfile = fopen(*destfile,"w")) == NULL) {
            fprintf(stderr,"plmas: can't create %s0,*argv);
            exit();
        }
        argc--; /* keeps parser from trying to read dest file */
        break;
    }

    /* initialize the symbol table structure */
    symboltable = (struct symbol *) malloc(sizeof(struct symbol));
    symboltable->string[0] = ' ';
    symboltable->next = 0;

    /* print header information in assembly output file, including _main
    * definition and the call to _initmsg and _doplms
    */
    printhead();

    /* Parse the input file(s) and write to the output file. This is a one
    * pass assembler. Since the output is then sent to the VAX assembler,
    * forward labels are OK. The symbol table is created during the
    * parse phase and written out at the end.
    */
    while (--argc) {
        morefiles = argc - 1; /* argc is 1 for the last file */
        if ((infile = fopen(*++argv,"r")) == NULL) {
            fprintf(stderr,"plmas: can't open %s0,*argv);
            exit();
        }
    }
}

```

```

    parser();
    fclose(infile);
}

/* Put the symbol table definition into the output file. This table
 * contains the definitions for printing out atoms and will be accessed
 * by the _plm_write function in escape.c
 */
outputsymboltable();
}

```

```

/* This procedure implements the main loop for the parser. It scans the
 * input file for a token in between instructions, which must either be
 * a label (which ends in a ':'), the word "procedure" to start a procedure
 * definition, the word "escape" to start an escape function definition,
 * the word "end" to signify the end of the file, or something else, which
 * is assumed to be an instruction keyword.
 */

```

```

parser()
{
    int end;
    token nexttoken;

    end = 0;
    while (!end) {
        nexttoken = gettoken();
        switch (nexttoken.type) {
            case INSTRUCTION:
                end = getinstruction(nexttoken);
                break;
            case ESCAPE:
                end = getescape();
                break;
            case LABEL:
                putlabel(nexttoken);
                break;
            case PROCEDURE:
                end = getprocedure();
                break;
            case END:
                end = 1;
                break;
        }
    }
}

```

```

/* This procedure is called when an inter-instruction token is not a
 * label, or one of the keywords "procedure", "escape" or "end". The
 * string is matched against the list of instruction keywords and then
 * the instruction specific operands are processed. A 0 is returned

```

```

* if everything is OK, a 1 is returned if input processing should stop.
*/
getinstruction(nexttoken)
token nexttoken;
{
    int index;

    index = getopcode(nexttoken.string);
    if (index == -1) {
        fprintf(stderr,
            "plmas: line %d: unknown instruction keyword: %s0,
            inputlinenumber,nexttoken.string);
        if (++errorcount >= MAXERROR) {
            fprintf(stderr,"plmas: too many errors, goodbye0);
            return(1);
        }
    } else {
        getoperands(index);
    }
    fprintf(outfile,"0);
    return(0);
}

```

```

/* This procedure processes the operands from the input file and puts the
* appropriate definitions into the output file.
*/

```

```

getoperands(index)
int index;
{
    int mask, i, xoperand, yoperand, symindex, tblindex, count;
    int structtable; /* discerns between struct and const hash */
    token oprn1, oprn2;

    if ((optable[index].operandpattern == XYI_OP) ||
        (optable[index].operandpattern == XYN_XI) ||
        (optable[index].operandpattern == XYN_XI_REV)) {
        oprn1 = gettoken();
        if (oprn1.string[0] == 'X') {
            fprintf(outfile,".word0x%02xfd0,
                optable[index].opcode);
            xoperand = 1;
        } else {
            fprintf(outfile,".word0x%02xfd0,
                optable[index].opcode+1);
            xoperand = 0;
        }
    } else {
        fprintf(outfile,".word0x%02xfd0,
            optable[index].opcode);
    }
    switch (optable[index].operandpattern) {
    case NONE: /* no operands */

```

```

    break;
case LABEL_OP:      /* one label operand */
    oprn1 = gettoken();
    fixlabel(oprn1.string);
    fprintf(outfile, ".byte0x8f0.long%s0,
        oprn1.string);
    break;
case XI_OP:         /* one Xi operand */
    oprn1 = gettoken();
    if ((oprn1.string[0] != 'X') ||
        (oprn1.string[1] < '1') ||
        (oprn1.string[1] > '8') ||
        (oprn1.string[2] != ' ')) {
        fprintf(stderr, "plmas: line %d: expected X1 - X80,
            inputlinenumber);
        errorcount++;
    } else {
        fprintf(outfile, ".byte0x5%d0,
            oprn1.string[1] - '1');
    }
    break;
case CONST_OP:     /* one constant operand */
    oprn1 = gettoken();
    fprintf(outfile, ".byte0x8f0.long0x%08x0,
        getconstantvalue(oprn1.string));
    break;
case N_OP:         /* one n operand */
    oprn1 = gettoken();
    fprintf(outfile, ".byte0x8f0.long0x%08x0,
        atoi(oprn1.string));
    break;
case FAIL_OP:      /* a "fail" operand */
    oprn1 = gettoken();
    if (strcmp(oprn1.string, "fail") != 0) {
        fprintf(stderr, "plmas: line %d: expected
            inputlinenumber);
        errorcount++;
    }
    break;
case XYI_OP:       /* Xi or Yi */
    /* already have operand */
    if ((oprn1.string[0] != 'X') && (oprn1.string[0] != 'Y')) {
        fprintf(stderr,
            "plmas: line %d: expected X<n> or Y<n>0,
            inputlinenumber);
        errorcount++;
    } else {
        if (xoperand) {
            fprintf(outfile, ".byte0x5%d0,
                oprn1.string[1] - '1');
        } else {
            yoperand = atoi(oprn1.string + 1) - 1;
            get_Yop(yoperand);
        }
    }

```

```

    }
    break;
case CONST_XI: /* a constant and then an Xi */
    opr1 = gettoken();
    fprintf(outfile, ".byte0x8f0.long0x%08x0,
        getconstantvalue(opr1.string));
    opr2 = gettoken();
    if ((opr2.string[0] != 'X') ||
        (opr2.string[1] < '1') ||
        (opr2.string[1] > '8') ||
        (opr2.string[2] != ' ')) {
        fprintf(stderr, "plmas: line %d: expected X1 - X80,
            inputlinenumber);
        errorcount++;
    } else {
        fprintf(outfile, ".byte0x5%d0,
            opr2.string[1] - '1');
    }
    break;
case FUNCT_XI: /* a functor and then an Xi */
    opr1 = gettoken();
    fixfunctor(opr1.string);
    fprintf(outfile, ".byte0x8f0.long0x%08x0,
        getconstantvalue(opr1.string));
    opr2 = gettoken();
    if ((opr2.string[0] != 'X') ||
        (opr2.string[1] < '1') ||
        (opr2.string[1] > '8') ||
        (opr2.string[2] != ' ')) {
        fprintf(stderr, "plmas: line %d: expected X1 - X80,
            inputlinenumber);
        errorcount++;
    } else {
        fprintf(outfile, ".byte0x5%d0,
            opr2.string[1] - '1');
    }
    break;
case CMASK_LABEL: /* a mask and then a label, then a table */
case SMASK_LABEL:
    if (optable[index].operandpattern == SMASK_LABEL)
        structtable = 1; /* functor arities must be removed */
    else
        structtable = 0; /* constants, don't check for arity */
    opr1 = gettoken();
    mask = atoi(opr1.string) >> 1;
    fprintf(outfile, ".word0x%02x8f0,mask);
    /* skip two labels to get to beginning of table */
    opr1 = gettoken();
    opr1 = gettoken();
    /* initialize the table array structure */
    for (i = 0; i <= mask; i++) {
        table_array[i].index = 0xffffffff;
    }
    /* read in the table and update the table array */

```

```

/* (note table is of the form: <symbol> [tcdr] <label> ...) */
for (i = 0; i <= mask; i++) {
    oprn1 = gettoken();
    if (strcmp(oprn1.string,"fail") == 0) {
        /* skip the tcdr and next fail */
        oprn1 = gettoken();
        oprn1 = gettoken();
    } else {
        if (structtable) fixfunctor(oprn1.string);
        symindex = getconstantvalue(oprn1.string);
        tblindex = symindex & mask;
        while (table_array[tblindex].index != 0xffffffff) {
            tblindex = (tblindex + 1) % (mask + 1);
        }
        table_array[tblindex].index = symindex;
        oprn1 = gettoken();
        if (strcmp(oprn1.string,"tcdr") == 0) {
            oprn1 = gettoken();
        }
        fixlabel(oprn1.string);
        if (strcmp(oprn1.string,"fail") == 0) {
            table_array[tblindex].string[0] = ' ';
        } else {
            strcpy(table_array[tblindex].string,oprn1.string);
        }
    }
}
/* print out table */
for (i = 0; i <= mask; i++) {
    if (table_array[i].index == 0xffffffff) {
        fprintf(outfile, ".long00);
        fprintf(outfile, ".long00);
    } else {
        fprintf(outfile, ".long0x%08x0,
            table_array[i].index);
        fprintf(outfile, ".long%s0,
            table_array[i].string);
    }
}
break;
case XYN_XI:          /* Xn or Yn, then Xi */
/* already have first operand */
oprn2 = gettoken();
if ((oprn1.string[0] != 'X') && (oprn1.string[0] != 'Y')) {
    fprintf(stderr,
        "plmas: line %d: expected X<n> or Y<n>0,
        inputlinenumber);
    errorcount++;
} else {
    if ((oprn2.string[0] != 'X') ||
        (oprn2.string[1] < '1') ||
        (oprn2.string[1] > '8') ||
        (oprn2.string[2] != ' ')) {
        fprintf(stderr,

```

```

        "plmas: line %d: expected X1 - X80,
        inputlinenumber);
        errorcount++;
    } else {
        if (xoperand) {
            fprintf(outfile,
                ".byte0x5%d0.byte0x5%d0,
                oprn1.string[1] - '1',
                oprn2.string[1] - '1');
        } else {
            yoperand = atoi(oprn1.string+1) -1;
            get_Yop(yoperand);
            fprintf(outfile, ".byte0x5%d0,
                oprn2.string[1] - '1');
        }
    }
}
break;
case YN_XI: /* Yn and then Xi */
    oprn1 = gettoken();
    oprn2 = gettoken();
    if (oprn1.string[0] != 'Y') {
        fprintf(stderr,
            "plmas: line %d: expected Y<n>0,
            inputlinenumber);
        errorcount++;
    } else {
        if ((oprn2.string[0] != 'X') ||
            (oprn2.string[1] < '1') ||
            (oprn2.string[1] > '8') ||
            (oprn2.string[2] != ' ')) {
            fprintf(stderr,
                "plmas: line %d: expected X1 - X80,
                inputlinenumber);
            errorcount++;
        } else {
            yoperand = atoi(oprn1.string+1) -1;
            get_Yop(yoperand);
            fprintf(outfile,
                ".byte0x5%d0,oprn2.string[1] - '1');
        }
    }
break;
case XYN_XI_REV: /* Xn or Yn, then Xi, reverse for output */
/* already have first operand */
    oprn2 = gettoken();
    if ((oprn1.string[0] != 'X') && (oprn1.string[0] != 'Y')) {
        fprintf(stderr,
            "plmas: line %d: expected X<n> or Y<n>0,
            inputlinenumber);
        errorcount++;
    } else {
        if ((oprn2.string[0] != 'X') ||
            (oprn2.string[1] < '1') ||

```



```

        (oprn2.string[1] > '8') ||
        (oprn2.string[2] != ' ') {
            fprintf(stderr,
                "plmas: line %d: expected X1 - X80,
                inputlinenumber);
            errorcount++;
        } else {
            if (xoperand) {
                fprintf(outfile,
                    ".byte0x5%d0.byte0x5%d0,
                    oprn2.string[1] - '1',
                    oprn1.string[1] - '1');
            } else {
                fprintf(outfile, ".byte0x5%d0,
                    oprn2.string[1] - '1');
                yoperand = atoi(oprn1.string+1) - 1;
                get_Yop(yoperand);
            }
        }
    }
    break;
case LABEL_N_REV: /* a label and then n, reverse for output */
    oprn1 = gettoken();
    fixlabel(oprn1.string);
    oprn2 = gettoken();
    if (atoi(oprn2.string) < 64)
        fprintf(outfile, ".byte0x%02x0,atoi(oprn2.string));
    else
        fprintf(outfile, ".byte0x8f0.byte0x%02x0,
            atoi(oprn2.string));
    fprintf(outfile, ".byte0x8f0.long%s0,
        oprn1.string);
    break;
case LABEL_LABEL_LABEL: /* three label operands */
    oprn1 = gettoken();
    if (strcmp(oprn1.string, "fail") == 0) {
        fprintf(outfile,
            ".byte0x8f0.long0xffffffff0);
    } else {
        fixlabel(oprn1.string);
        fprintf(outfile, ".byte0x8f0.long%s0,
            oprn1.string);
    }
    oprn1 = gettoken();
    if (strcmp(oprn1.string, "fail") == 0) {
        fprintf(outfile,
            ".byte0x8f0.long0xffffffff0);
    } else {
        fixlabel(oprn1.string);
        fprintf(outfile, ".byte0x8f0.long%s0,
            oprn1.string);
    }
    oprn1 = gettoken();
    if (strcmp(oprn1.string, "fail") == 0) {

```

```

        fprintf(outfile,
            ".byte0x8f0.long0xfffffffff0);
    } else {
        fixlabel(oprn1.string);
        fprintf(outfile, ".byte0x8f0.long%s0,
            oprn1.string);
    }
    break;
case TWO_XI: /* Two Xi, Yi, or N operands followed by an Xi */
    for (count = 1; count <= 3; count++) {
        oprn1 = gettoken();
        switch (oprn1.string[0]) {
            case 'X':
                fprintf(outfile, ".byte0x5%d0,
                    oprn1.string[1] - '1');
                break;
            case 'Y':
                fprintf(outfile, ".byte0xce0);
                fprintf(outfile, ".word0x%04x0,
                    -((atoi(oprn1.string + 1) - 1) * 4)
                    & 0xffff);
                break;
            case '&':
                fprintf(outfile, ".byte0x8f0);
                fprintf(outfile, ".long0x%08x0,
                    atoi(oprn1.string + 1) ! 0xc0000000);
                break;
        }
    }
    break;
}
}
}

```

/* This procedure is used to process a Yn operand. There are three cases:

- * 1) operand is Y1, represented as (r14)
- * 2) operand is <= Y64, represented as B^D(r14)
- * 3) operand is > Y64, represented as W^D(r14)

```

*/
get_Yop(yvalue)
int yvalue;
{
    /* This doesn't work if mode = ASRC
    if (yvalue == 0)
        fprintf(outfile, ".byte0x6e0);
    else */
    if (yvalue <= 31) {
        fprintf(outfile, ".byte0xae0);
        fprintf(outfile, ".byte0x%02x0, -(yvalue * 4) & 0xff);
    }
    else {
        fprintf(outfile, ".byte0xce0);
        fprintf(outfile, ".word0x%04x0, -(yvalue * 4) & 0xffff);
    }
}
}

```

```

/* This procedure is called when a label is detected in the input file.
 * Since the output is processed by the VAX assembler, labels can just
 * be passed through to the output file.
 */

```

```

putlabel(nexttoken)
token nexttoken;
{
    fprintf(outfile,"%s0,nexttoken.string);
    return(0);
}

```

```

/* This procedure is called when the keyword "procedure" is found in
 * the input stream. The next token is assumed to be a label that
 * is then put into the output file.
 *
 * If there are more WAM files left to parse (signified by 'morefiles')
 * and the procedure name is "allocate_dummy", then all processing of the
 * current WAM file is stopped by returning 1. Each WAM file has an
 * identical allocate_dummy procedure which should only be output once.
 * We do so while processing the last WAM file.
 */

```

```

getprocedure()
{
    token procname;

    procname = gettoken();

    if (morefiles && (strcmp(procname.string,"allocate_dummy/0") == 0))
        return(1);
    fixlabel(procname.string);
    fprintf(outfile,"%s:0,procname.string);
    return(0);
}

```

```

/* This procedure is called when the keyword "escape" is found in
 * the input file. The next token is assumed to be the name of
 * a C level procedure to be called. The last character of the
 * token must be a number indicating the number of parameters to
 * be passed to the C procedure.
 */

```

```

getescape()
{
    int i, argc, logical;
    token escapename;

    escapename = gettoken();

    /* the logical escapes are now implemented as new instructions
     * in VAX 8600 microcode
     */
}

```

```

if (strcmp(escapename.string, "=/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_EQ);
} else if (strcmp(escapename.string, "\=/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_NEQ);
} else if (strcmp(escapename.string, "=./2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_UNIV);
} else if (strcmp(escapename.string, ">/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_GT);
} else if (strcmp(escapename.string, "</2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_LT);
} else if (strcmp(escapename.string, ">=/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_GE);
} else if (strcmp(escapename.string, "<=/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_LE);
} else if (strcmp(escapename.string, "integer/1") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_INTEGER);
} else if (strcmp(escapename.string, "number/1") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_INTEGER);
} else if (strcmp(escapename.string, "atom/1") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_ATOM);
} else if (strcmp(escapename.string, "length/2") == 0) {
    fprintf(outfile, ".word0x%02xfd0,ESCAPE_LENGTH);
} else if (strcmp(escapename.string, "plus/3") == 0) {
    fprintf(outfile, ".word0x%02xfd0,PLUS);
} else if (strcmp(escapename.string, "minus/3") == 0) {
    fprintf(outfile, ".word0x%02xfd0,MINUS);
} else if (strcmp(escapename.string, "mult/3") == 0) {
    fprintf(outfile, ".word0x%02xfd0,IS_IN);
    fprintf(outfile, "nop0);          /* timing problem */
    fprintf(outfile, "mull2r3,r10); /* mull2 r3,r1 */
    fprintf(outfile, ".word0x%02xfd0,IS_OUT);
} else if (strcmp(escapename.string, "div/3") == 0) {
    fprintf(outfile, ".word0x%02xfd0,IS_IN);
    fprintf(outfile, "nop0);          /* timing problem */
    fprintf(outfile, "divl2r3,r10); /* divl2 r3,r1 */
    fprintf(outfile, ".word0x%02xfd0,IS_OUT);
} else if (strcmp(escapename.string, "mod/3") == 0) {
    fprintf(outfile, ".word0x%02xfd0,IS_IN);
    fprintf(outfile, "nop0);          /* timing problem */
    fprintf(outfile, "divl3r3,r1,r20); /* divl3 r3,r1,r2 */
    fprintf(outfile, "mull2r3,r20); /* mull2 r3,r2 */
    fprintf(outfile, "subl2r2,r10); /* subl2 r2,r1 */
    fprintf(outfile, ".word0x%02xfd0,IS_OUT);
} else {

```

/* the non-logical escapes are done in C */

```
fixlabel(escapename.string);
```

```
/* get argument count */
```

```
i = 0;
```

```
while (escapename.string[i] != ' ') i++;
```

```
argc = escapename.string[i - 1] - '0';
```

```
if ((argc > 8) || (argc < 0)) {
```

```

        argc = 0;
    }

    /* first do the microcode escape instruction saving PSL */
    fprintf(outfile, ".word0x%02xfd0,ESC_IN);

    /* trail X1 if the escape is get_1 or is_2*/
    if ( (strcmp(escapename.string, "get_1") == 0) ||
        (strcmp(escapename.string, "is_2") == 0) )
        fprintf(outfile, ".word0x%02xfd0,TRAIL_X1);

    /* generate call to escape routine */
    fprintf(outfile, "cmplsp,fp0.word0x08190);
    fprintf(outfile, "movlsp,(fp)0);
    fprintf(outfile, "movlfp,sp0.word0x0c110);
    fprintf(outfile, "movlsp,w^-1024(sp)0);
    fprintf(outfile, "subl2$1024,sp0pushr$0x3f0);
    for (i = 0; i < argc; i++) {
        fprintf(outfile, "pushlr%d0,i);
    }

    /* push heap and trail if escape is name_2 */
    if (strcmp(escapename.string, "name_2") == 0) {
        fprintf(outfile, "pushlap0);
        fprintf(outfile, "pushlr90);
        argc = 4;
    }
    fprintf(outfile, "calls$d,_plm_%s0,argc,escapename.string);
    fprintf(outfile, "tsl0);
    fprintf(outfile, ".word0x09120);
    fprintf(outfile, "popr$0x3f0movl(sp),sp0);
    fprintf(outfile, ".word0x%02xfd0,ESC_OUT);
    fprintf(outfile, ".word0x06fd0);
    fprintf(outfile, "popr$0x3f0movl(sp),sp0);

    /* if escape is name_2 */
    /* increment heap and trail by the values stored */
    /* in reserved locations of memory */
    if (strcmp(escapename.string, "name_2") == 0) {
        fprintf(outfile, "addl20x7fff0014,ap0);
        fprintf(outfile, "addl20x7fff0010,r90);
    }
    fprintf(outfile, ".word0x%02xfd0,ESC_OUT);
}
return(0);
}

```

```

/* This procedure is called to remove the arity associated with a
 * functor. Only the functor itself and not it's arity should be used
 * to determine it's encoding.
 */
fixfunctor(string)

```

```

char *string;
{
    while ((string[0] != '/') || (string[1] == '/'))
        string++;
    *string = ' ';
}

/* This procedure is called to replace all occurrences of '/' in a
 * file to '_'. This is done so that labels will be valid for the
 * assembler.
 *
 * In addition, this procedure fixes up the logical escape calls
 * replacing '=' with 'e', '<' with 'l', and '>' with 'g'.
 */

```

```

fixlabel(string)
char *string;
{
    while (*string != ' ') {
        switch (*string) {
            case '/': *string = '_'; break;
            case '<': *string = 'l'; break;
            case '>': *string = 'g'; break;
            case '=': *string = 'e'; break;
            default: break;
        }
        string++;
    }
}

```

```

/* This procedure searches the optable structure for a match of the
 * token with one of the instruction keywords. If a match is found,
 * the appropriate index into the optable structure is returned, otherwise
 * -1 is returned
 */

```

```

getopcode(string)
char *string;
{
    int i;

    for (i = 0; optable[i].instruction[0] != ' '; i++) {
        if (strcmp(optable[i].instruction, string) == 0) {
            return(i);
        }
    }
    return(-1);
}

```

```

/* This procedure gets a token from the input file. A token is a sequence
 * of printable characters separated by non-printable characters or by
 * a comma. If the end of file is detected, END is returned and if the
 * token ends in ':', LABEL is returned, if the token is "escape",

```

* "procedure" or "end" then ESCAPE, PROCEDURE or END are returned
 * respectively. Otherwise, the value INSTRUCTION is assigned to the type
 * field.

```

*/
token gettoken()
{
    int i, c;
    static token result;

    i = 0;
    c = Getc();
    while ((c != EOF) && ((c <= ' ') || (c > '~'))) {
        c = Getc();
    }
    /* now check for special case constant delimited by single quotes */
    /* some of these may even have an arity following */
    if (c == '"') {
        c = Getc(); /* skip the single quote */
        while ((c != EOF) && ((c >= ' ') && (c <= '~')) && (c != '"')) {
            result.string[i++] = c;
            c = Getc();
        }
        c = Getc(); /* skip comma or check for a functor */
        if (c == '/') {
            result.string[i++] = c; /* yes, take '/' */
            result.string[i++] = Getc(); /* and arity */
            c = Getc(); /* now skip comma */
        }
    } else {
        while ((c != EOF) && ((c > ' ') && (c <= '~')) && (c != ',')) {
            result.string[i++] = c;
            c = Getc();
        }
    }
    result.string[i] = ' ';

    if (c == EOF) {
        result.type = END;
    } else if (result.string[i-1] == ':') {
        result.type = LABEL;
    } else if (strcmp(result.string, "procedure") == 0) {
        result.type = PROCEDURE;
    } else if (strcmp(result.string, "escape") == 0) {
        result.type = ESCAPE;
    } else if (strcmp(result.string, "end") == 0) {
        result.type = END;
    } else {
        result.type = INSTRUCTION;
    }

    return(result);
}

Getc()

```

```

{
    static char c;
    char d;

    d = c;
    c = getc(infile);
    if (d == '0') {
        inputlinenumber++;
    }
    return(c);
}

```

```

/* This procedure prints out header information into the output file.
* A comment line is printed followed by assembler directives to create
* the global _main and to call _initmsg and _doplm.
*/

```

```

printhead()
{
    fprintf(outfile, ".text0.globl _main0);
    fprintf(outfile, "_main:0.word00);
    fprintf(outfile, "movlfp,0x7fff00080);
    fprintf(outfile, "calls$0,_init0);
    fprintf(outfile, "movlr0,_end+80);
    fprintf(outfile, "movl$d,_end+40,repeatcount);
    fprintf(outfile, "_plm.repeat:0movl_end+8,ap0);
    fprintf(outfile, "movl$0x7fff0008,sp0);
    fprintf(outfile, "jsb_doplm0);
    fprintf(outfile, "subl2$1,_end+40);
    fprintf(outfile, "bneq_plm.repeat0);
    fprintf(outfile, "movlr0,r00beql_main10);
    fprintf(outfile, "calls$0,_writeyes0brb_main20);
    fprintf(outfile, "_main1:calls$0,_writeno0);
    fprintf(outfile, "_main2:movl0x7fff0008,fp0ret0);
    fprintf(outfile, "0doplm:0.word0x%02xfd0,RESET);
}

```

```

/* This procedure prints out the symbol table to the output file.
*/

```

```

outputsymboltable()
{
    int i;
    struct symbol *next;

    fprintf(outfile, ".globl _atomlist0atomlist:0);
    i = 0;
    next = symboltable;
    while (next->string[0] != ' ') {
        fprintf(outfile, ".long__sym%d0,i++);
        next = next->next;
    }
}

```



```

fprintf(outfile, ".long__endsym0);
i = 0;
next = symboltable;
while (next->string[0] != ' ') {
    fprintf(outfile, "__sym%d:0.asciz
        next->string);
    next = next->next;
}
fprintf(outfile, "__endsym:0.byte00);

```

```

/* reserve two longwords for the malloc heap pointer and loop count */

```

```

/* statements needed for VMS Macro
fprintf(outfile, "_end:0.long00.long00);
fprintf(outfile, ".end main0);
*/
}

```

```

/* This procedure puts a string into the symbol table if it isn't already
* there and returns a new index. If the string is already there, it
* returns the old index.
*/

```

```

unsigned int getconstantvalue(string)
char *string;
{
    int i;
    struct symbol *next;

    if (strcmp(string, "[ ]") == 0) {
        return(0xffffffff);
    } else if (string[0] == '&') {
        return(0xc0000000 + atoi(string + 1));
    }
    i = 0;
    next = symboltable;
    while (next->string[0] != ' ') {
        if (strcmp(next->string, string) == 0) {
            return(0xc8000000 + i);
        }
        i++;
        next = next->next;
    }
    next->next = (struct symbol *) malloc(sizeof(struct symbol));
    strcpy(next->string, string);
    next->next->string[0] = ' ';
    next->next->next = 0;
    return(0xc8000000 + i);
}

```

