This report should be referenced as:

Brown, A.L., Carrick, R., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., Morrison, R. & Munro, D.S. “The Persistent Abstract Machine Version 10 / Napier88 (Release 2.0)”. Universities of St Andrews and Adelaide (1994).

The Persistent Abstract Machine

Version 10 / Napier'88 (Release 2.0)

A.L. Brown†, R. Carrick, Q.I. Cutts, R.C.H. Connor
A. Dearle†, G.N.C. Kirby, R. Morrison, D.S. Munro

Department of Mathematical and Computational Sciences,
University of St. Andrews,
North Haugh, St Andrews
KY16 9SS, Scotland

†Department of Computer Science,
University of Adelaide,
South Australia 5005,
Australia.

PAM v10 Abstract

In recent years, research into persistent programming systems has led to the design of sophisticated database programming languages such as Galileo, PS-algol, and Napier. These languages provide a wide range of abstraction facilities such as abstract data types, polymorphism and first class procedures that are integrated within a single persistent store. The development of these systems has required the design of a variety of new implementation techniques. For example, the development of the Napier system required the design of reusable compiler componentry, an intermediate language, an abstract machine and a persistent object store, all of which are integrated into a highly modular layered architecture. Here we present a description of the Persistent Abstract Machine.

Contents

1 Introduction

In recent years, research into persistent programming systems has led to the design of sophisticated database programming languages such as Galileo[1], PS-algol[2], and Napier[3]. These languages provide a wide range of abstraction facilities such as abstract data types, polymorphism and first class procedures that are integrated within a single persistent store. The development of these systems has required the design of a variety of new implementation techniques. For example, the development of the Napier system required the design of reusable compiler componentry[4], an abstract machine and a persistent object store, all of which are integrated into a highly modular layered architecture[6]. Here we present a description of the Persistent Abstract Machine.

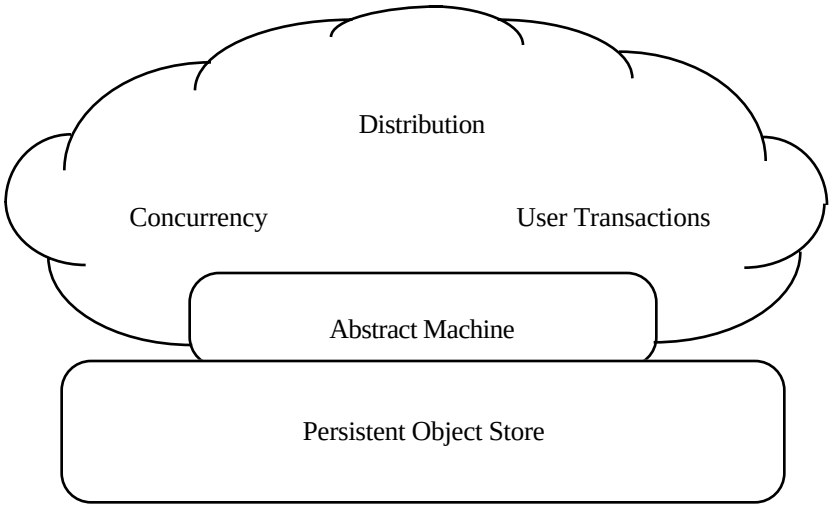


Figure 1. The major architecture components.

The Persistent Abstract Machine is primarily designed to support the Napier programming language. It is closely based on the PS-algol abstract machine[7], which in turn evolved from the S-algol abstract machine[8]. Due to the modularity of its design and implementation, it may be used to support any language with no more than the following features: persistence, polymorphism, subtype inheritance, first class procedures, abstract data types and block structure. Other features, such as object-oriented programming in the Smalltalk style[9] and lazy evaluation, may be modelled at a higher level using the same support mechanisms as first class procedures[10,11]. This covers most algorithmic, object-oriented and applicative programming languages currently in use. The machine can thus be said to be multi-paradigm.

The Napier system is designed so that implementors wishing to use the abstract machine may compile to an intermediate level architecture, consisting of abstract syntax trees. A code generator is available to compile to the abstract machine level. This persistent architecture intermediate language, PAIL[5], supports all of the abstractions listed above, and is sufficiently high-level to ease the burden of compiler writing. Furthermore, it is possible to check at this level that correct (i.e. consistent) PAIL code has been generated, and so output from an untrusted compiler cannot cause a malfunction in the abstract machine. This assures that the persistent store may not be corrupted by the generation of illegal instruction sequences, removing the onus of segmentation protection from the store.

Many features of the design of the Persistent Abstract Machine are directly attributable to the Napier language. The major points fall into two sections:

1. The machine is an integral part of an entire layered architecture; in particular, it interfaces cleanly with the persistent store, and allows an elegant implementation of persistence in a high-level language. As another consequence of persistence, the machine exists in a single heap-based storage architecture. This architecture directly gives a method

of implementing block retention, for almost no extra cost. This allows the implementation of first class procedures and modules in programming languages with the minimum of effort.

2. A primitive two-level type system within the machine contains enough information to allow machine instructions whose behaviour depends on the dynamic type of their operands. It has a fast and efficient integer encoding. In conjunction with the block retention architecture, the type system is used to great effect to provide a fast implementation of polymorphic procedures, abstract data types, and bounded universal quantification.

1.1 A Heap Based Storage Architecture

One of the most notable features of the abstract machine is that it is built entirely upon a heap-based storage architecture. Although the machine was primarily designed to support a block-structured language, for which a stack implementation might be the obvious choice, the heap-based architecture was considered advantageous for the following reasons:

1. Only one storage mechanism is required, easing implementation and system evolution.

2. There is only one possible way of exhausting the store. In a persistent system this is an essential requirement, since applications should only run out of store when the persistent store is exhausted, and not merely when one of the storage mechanisms runs out. Although this could be modelled in an environment with more than one storage mechanism, it would be expensive in terms of implementation and evolution.

3. The Napier language supports first-class procedures with free variables. To achieve the desired semantics, the locations of these variables may have to be preserved after their names are out of scope, which would not happen conveniently in a conventional stack-based system.

Stacks are still used conceptually, and each stack frame is modelled as an individual data object. Stack frames represent the piece of stack required to implement each block or procedure execution of the source language. To aid garbage collection, a stack frame contains two separate stacks, one for pointers and one for non-pointers. The size of each frame can be determined statically, which leads to an efficient use of the available working space.

1.2 A Low-Level Type System

A major design decision of the system is to have non-uniform representation of different types of objects in the machine. Some systems, particularly those which support polymorphism and other type abstractions, have a uniform representation in which every object is wrapped in a pointer to a heap object[12]. This allows type abstraction to be implemented easily, but has a drawback in efficiency. The Napier system has a number of different representations for objects on the machine's stacks, including some which have part of their value on either stack. This causes problems with stack balancing and object addressing, solutions to which are presented below.

The abstract machine has its own type system, albeit a very low-level and unenforced system. It is a two-level system, one level describing object layout and the other including some semantic knowledge of the object. Most of the abstract machine instructions are typed, although no attempt is made to ensure that the operand is of the correct type ; the type acts in effect as a parameter to the instruction.

Many language constructs involve operations where the type of the operands is not known statically. As equality is defined over all types in Napier, but is defined differently according to the type of its operands, it is necessary to perform a dynamic type lookup wherever the operand type is

not known statically. This happens in the case of variants, anys, polymorphic quantified types, and witnesses to abstract data types. A further need occurs when statically unknown types are assigned into or dereferenced from other data objects, when it is necessary to find the dynamic type to calculate the correct size and addressing information.

Both levels of the type system are finite, and contain only a small number of different types. The first level of the type system contains information as to the location and size of the instruction's operand. The machine supports six different types of objects, which are:

Operand shape	Prefix to instruction mnemonic (appendix 1)
one integer word,	w
two integer words,	dw
one pointer	p
two pointers	dp
one integer word and one pointer	wp
two integer words and two pointers	dwdp

The instructions which are typed in this manner are those instructions which need to know only the shape of the object upon which they operate. These are instructions such as stack load, assignment, duplicate, and retract. The machine operations need to know nothing about the semantic nature of the objects in these locations.

The other, slightly higher level, system is required when the operation does depend on the semantics of the operand. These are all the operations which involve comparison of two objects, in which case the shape of the object is not sufficient. The machine supports different types with the same object formats. An example of this is structures and strings: they both consist of a single pointer, but equality is defined by identity on structures and by character equality on strings. The types supported are:

High-level type(s)	Suffix to instruction mnemonic (appendix 1)
integer,boolean, pixel,	.ib
real	.r
string	.s
structure,vector,image,file abstract data type	.p
procedure	.pr
variant	.var
polymorphic	.poly
any	.any

These are the eight different classes of equivalence defined by the abstract machine.

It would be possible to do a certain amount of static checking on abstract machine code to ensure that this type system is not broken, which could perhaps be useful if an untrusted compiler was producing abstract machine code. However, incorrect store instructions, such as addressing off the end of an object, could not be statically checked from abstract machine code. This is the main danger in allowing untrusted compilers to access the abstract machine at this level. As there is only one persistent store for all users, it is essential that untrusted machine code is not used.

1.3 Concurrency, Distribution and User Transactions

To support concurrency the abstract machine provides lightweight threads. Threads may be created using a thread creation operation which is supplied with a void procedure and returns an integer

identifier. The new thread executes the supplied procedure in parallel with the invoking thread. Operations such as who am I, suspend, restart and kill can be performed on any thread provided its thread identifier is known.

Interaction between threads that share data is controlled by the abstract machine. All abstract machine operations are executed as atomic operations. Thus, conflicting operations on shared data appear to be performed one at a time in some arbitrary order. When a Napier level logical operation must be performed without interruption, a per thread lock word is used to mark a thread as executing in a critical section. Until the lock word is reset to 0 the executing thread has exclusive access to the abstract machine. Access to the lock word is controlled by an abstract machine instruction.

Distribution is supported through the provision of a socket interface. Using the socket interface, a Napier system is able to communicate with other Napier systems exchanging a stream of 32-bit integers. Since all persistent objects are defined in terms of 32-bit integer values, Napier systems may communicate with each other without regard to their architecture specific integer representations. Given this simple building block more sophisticated protocols and distribution mechanisms may be constructed.

No explicit support for user transactions is currently provided.

1.4 Errors and Events

The persistent abstract machine supports the automatic execution of user defined procedures when an error condition or asynchronous event occurs. These procedures are held in data structures pointed to by the root object, see Chapter 6.

User defined procedures for error conditions may supply an alternate result for the erroneous operation or report an error and abort the executing thread. With the exception of variant projection errors, an executing thread may continue execution if the error procedure returns.

Asynchronous events handled by the abstract machine include the UNIX signals hangup, interrupt and quit together with two timer interrupts, one for timer related tasks and one for network related tasks. Since the handler procedures are called by the current thread, they are only invoked between abstract machine instructions and then only if the per thread lock word is 0.

2 Abstract Machine Registers

The registers of the persistent abstract machine are:

ROP	abstract machine root object pointer
LFB	local frame base
LMSP	local frame main stack top
LPSP	local frame pointer stack top
CP	code pointer

2.1 ROP

A garbage collection of the persistent heap retains all objects that are reachable, by following object addresses (pointers), from the root object. Since the persistent heap is the only storage mechanism available to the persistent abstract machine, the abstract machine must arrange for all active data objects, including its own housekeeping information, to be reachable from the root object. This is achieved by making the root object of the persistent heap point to a special object created for the abstract machine. The special object, known as the root object for the abstract machine, is pointed to by the ROP register. The object contains all the housekeeping information required by the abstract machine, including the current state of any active programs, and a pointer field that is used as the root of persistence for user data.

2.1.1 Object Formats

All heap objects are laid out in a consistent manner in order that the system utilities may operate on them irrespective of their type. Thus all heap objects have the same format which is as follows: (a word is a 32 bit integer)

word 0	header
word 1	the size in words of the object
word 2..n	the pointer fields
word n+1..	the non pointer fields

2.1.1.1 The Header

Word 0 has the following interpretation:

bits 0-23	the number of pointer fields in the object
bit 24	trace bit for use by procedure return instructions **or** constancy bit for validating updates to vectors
bit 25-31	reserved for implementation experiments

where bit 0 is the least significant bit of the word.

2.1.1.2 The Pointer Fields

The pointer fields within an object are a single word in length. Since the persistent heap uses object level addressing, all pointers must address the start of an object. That is, a pointer may never directly address the contents of an object. Individual fields are addressed by a pointer to an object and an index within the object.

2.1.2 The Abstract Machine Root Object

The root object for the abstract machine may be logically viewed as having the following fields:

word 0,1	object header and size
word 2	the pointer literal **nil**
word 3,4	the closure for the startup procedure
word 5	the logical root of persistence
word 6	the file literal **nilfile**
word 7	the string literal ""
word 8	a pointer to the vector of all 128 single character strings
word 9	the image literal **nilimage**
word 10,11	closure for a procedure that compares the types of two anys
word 12	a pointer to the vector of event handling procedures
word 13	a pointer to the structure of error handling procedures
word 14	a pointer to the vector of open files
word 15,16	the closure for the type checking procedure, **nil**'s if not in use
word 17	a pointer to the types module, **nil** if not in use
word 18	the error number for the last I/O instruction executed
word 19	the integer literal **maxint**
word 20,21	the real literal **maxreal**
word 22,23	the real literal **pi**
word 24,25	the real literal **epsilon**
word 26	the abstract machine magic number: the second last word in the root object
word 27	the compiler magic number: the last word in the root object

In practise, the abstract machine root object may be significantly different and in fact may be distributed among multiple thread context objects. However, all implementations of the abstract machine will provide the logical view presented above by suitably interpreting the root object accessing instructions.

2.2 LFB

The persistent abstract machine implements a stack using a separate heap object for each stack frame, described in section 3.2.6. A stack frame is created whenever a procedure is called or a block is executed. The LFB register is used to point to the stack frame for the currently executing procedure or block (the local frame) and must be updated on every procedure call, procedure return, block entry and block exit. All local data may be accessed by indexing the LFB.

2.3 LMSP and LPSP

In order to conform to the single object format described above, each object representing a stack frame actually contains two distinct stacks. One stack contains pointers (the pointer stack) and the other contains non-pointers (the main stack). Within the local frame, pointed to by the LFB register, the LMSP register points to the top of the main stack and the LPSP register points to the top of the pointer stack. In fact the LMSP and LPSP registers point to the word following the last word on the appropriate stack. It should be noted that the LMSP and LPSP directly address the contents of a heap object. However, these registers are never stored in the persistent heap and are always recalculated whenever the LFB register is updated.

2.4 CP

The next abstract machine instruction to be executed is directly addressed by the CP register. The CP register is similar to the LMSP and LPSP registers in that it is never stored in the persistent heap. Its contents are always recalculated whenever the object containing the abstract machine code is changed or moved.

3 Data Types

The persistent abstract machine supports a range of data types that may be classified as scalar data types, pointer data types, and mixed data types. The scalar data types represented by integer words are: *integer*, *boolean*, *pixel* and *real*. The pointer data types represented by the addresses of data objects (pointers) are: *string, file, vector, image, structure, procedure* and *abstract data type*. The mixed data types represented by a combination of integer words and pointers are: *variant* and *any*.

3.1 Scalar Data Types

3.1.1 Integer

Integers are represented by a single 32 bit word using two's complement, i.e. the range of *integer* values is -2147483648 to +2147483647. The bits within an integer word are numbered from 0 to 31 with bit 0 being the least significant and bit 31 the most significant.

The following operations are permitted on an *integer*, described in chapter 4:

> equals, not equals, less than, less than or equals, greater than, greater than or equals, negate, plus, minus, multiply, quotient on division, remainder on division bitwise and, bitwise or, bitwise not, arithmetic shift right, arithmetic shift left and translate into the floating point representation (*real*).

Any arithmetic operation on *integers* whose result is outwith the supported range of values, or requires division by 0, is treated as an error.

3.1.2 Boolean

The *boolean* data type has two values, **true** and **false**. **true** is represented by the integer value 1 and **false** is represented by the integer value 0.

The following operations are permitted on a *boolean*, described in chapter 4:

Two *booleans* are equal if they have the same integer value.

3.1.3 Pixel

The *pixel* data type supports pixels of up to 24 planes in depth. A *pixel* is represented as an integer value. The depth of the *pixel* is held as an 8 bit integer in the most significant 8 bits of the integer value with each of the remaining 24 bits representing an individual plane. Plane 0 of the *pixel* is represented by the least significant bit of the integer, bit 0, plane 1 of the *pixel* is represented by bit 1, and so on.

The following operations are permitted on a *pixel*, described in chapter 4:

> equals, not equals and not. equals, not equals,
> concatenate *pixels* and
> subpixel selection.

An attempt to create a *pixel* whose depth is not in the range 1 to 24 or an attempt to select non existent planes from a *pixel* are both treated as errors. Two *pixels* are equal if they have the same integer value.

3.1.4 Real

The *real* data type supports floating point numbers with magnitudes in the range 4.94065645841246544e-324 to 1.79769313486231470e+308. A *real* is represented as a pair of integer words that make up a 64 bit floating point number conforming to the IEEE 754 standard[13]. The integer word with the lower address is referred to as word 0, and the other word as word 1. The address of word 0 is used as the address of the real. Bit 31 of word 0 contains the sign bit, with the signed exponent being held in bits 20 to 30 of word 0. The remaining 52 bits form the fraction, the higher numbered bits are more significant than the lowered number bits and the bits of word 0 are more significant than the bits of word 1.

The following operations are permitted on a *real*, described in chapter 4:

equals, not equals, less than, less than or equals, greater than, greater than or equals,
negate, plus, minus, multiply, divide,
translate into the *integer* representation ignoring the fractional part,
sine, cosine, arctangent, square root, log to base e and e raised to a given power.

Any floating point operation that causes a floating point overflow or underflow or whose result is a NaN (not a number) is treated as an error. Similarly, an attempt to translate a *real* into an *integer* whose value cannot be represented by the *integer* data type is also an error. All comparison operations on *reals* conform to the IEEE standard.

3.2 Pointer Data Types

All pointer data types are represented by either one or two object addresses.

3.2.1 Strings

The *string* data type is represented by a single pointer to a heap object with the following object format:

word 0,1	object header and size
word 2	number of characters in the *string*
word 3..	the characters 4 per word, the last word is padded with zero characters.

Within each word of 4 characters, the first character is encoded in the most significant byte, the second character in the next most significant byte, the third character in the next most significant byte and the fourth character is encoded in the least significant byte.

The following operations are permitted on a *string*, described in chapter 4.

equals, not equals, less than, less than or equals, greater than, greater than or equals,
concatenate *strings* and
substring selection.

An attempt to select a non existent section of a *string*, or a section of negative length is treated as an error. Two *strings* are equal if they are the same length and all the corresponding characters in each *string* are equal. Two characters are equal if they have the same ascii code. A *string*, A, is less than a *string*, B, if all the characters in A with a corresponding character in B are the same as the corresponding character in B and A is shorter or if the first character in A that differs from the corresponding character in B is less than its corresponding character in B.

3.2.2 Files

There are several kinds of *files* that are supported by the persistent abstract machine, disk files, terminals, raster windows and sockets. All file descriptor objects have four pointer fields, the first of which is the *file*'s name, and at least 2 integer words. The first integer word contains an internal *file* number and associated flag bits and the second integer is a datestamp set when the *file* is opened. The datestamp indicates whether the *file* was opened in this invocation of the system.

The internal *file* number and associated flag bits are represented as follows:

if a socket	bit 21
if a terminal	bit 20
if a disk file	bit 19
if a window	bit 18
if writable	bit 17
if readable	bit 16
file number	bits 0-15

The filename may specify a particular file type and attributes. If no recognised prefix is given the prefix "DISK:" is assumed.

3.2.2.1 Disk Files

word 0,1	object header and size
word 2	a pointer to the *file*'s name
word 3	a pointer to the *working directory* when the file was opened
word 4	unused, a **nil** pointer
word 5	unused, a **nil** pointer
word 6	an internal *file* number and associated flag bits
word 7	a datestamp set when the file is opened
word 8	the current position in the disk file (byte offset from the start)

Disk file objects are created whenever a *file* is opened or created in an external file system. The filename prefix for a disk file is "DISK:". The operations permitted on a disk file are:

equals, not equals
create, open, close, reopen,
read bytes, write bytes and
position the next read or write.

3.2.2.2 Terminal Files

word 0,1	object header and size
word 2	a pointer to the *file*'s name
word 3	unused, a **nil** pointer
word 4	unused, a **nil** pointer
word 5	unused, a **nil** pointer
word 6	an internal *file* number and associated flag bits
word 7	a datestamp set when the file is opened
word 8..n	the terminal modes currently selected

The terminal modes are implementation dependent and should only be accessed via the ioctl instruction interface described in 4.13.

Terminal *file* objects are created whenever a terminal device is opened. The filename prefix is "TTY:". If the filename prefixes "STDIN:", "STDOUT:" or "STDERR:" are specified then *file*

objects are created for the Napier system's standard input, output and error. These files are permanently open and are assumed to be terminal devices.

The operations permitted on a terminal *file* are:

equals, not equals
open, close, reopen,
read bytes, write bytes and
get/ set the terminal modes.

3.2.2.3 Socket Files

word 0,1	object header and size
word 2	a pointer to the *file*'s name
word 3	unused, a **nil** pointer
word 4	unused, a **nil** pointer
word 5	unused, a **nil** pointer
word 6	an internal *file* number and associated flag bits
word 7	a datestamp set when the file is opened

A socket *file* object is created whenever an incoming network connection is accepted or a connection to a remote Napier system is successful. The filename prefixes for a socket are "ACCEPT:", "CONNECT:" and "SHELL:".

"ACCEPT:" is used to accept a connection from a remote Napier system.

"CONNECT:" is used to connect to a named Napier system. The connection name is in the form of a host identifier and the directory name of a persistent store separated by a double colon, **::**. The host identifier may be either a symbolic or numeric internet address.

"SHELL:" is used to specify a socket connected to a command line interpreter. The command line interpreter is started when the Napier system is invoked. In a UNIX system the interpreter is a shell.

The operations permitted on a socket are:

equals, not equals,
open, close, reopen,
read bytes, write bytes and
get/ set the socket modes.

3.2.2.4 Raster Window Files

word 0,1	object header and size
word 2	a pointer to the *file*'s name
word 3	an image representing the raster device's screen
word 4	an image representing the screen's cursor
word 5	unused, a **nil** pointer
word 6	an internal *file* number and associated flag bits
word 7	a datestamp set when the file is opened
word 8	the X position of the cursor on the screen
word 9	the Y position of the cursor on the screen
word 10	the X position of the cursor hot spot
word 11	the Y position of the cursor hot spot
word 12	the raster rule used to display the cursor on the screen (see rasterop)
word 13	the millisecond datestamp of the last locator event read

word 14+n state of the nth button, numbered from 0
a value of 0 indicates that the n^{th} button is up,
a value of 1 indicates that the n^{th} button is down.

A window *file* object is created whenever a raster window is opened. The filename prefix for a window is "WINDOW:". If no window name is given a default window is opened in the host environment. For example, a shell variable DISPLAY may have been set to specify an X display to use. Alternatively it may be possible to access the local frame buffer and use that to simulate a window.

A window filename may include specifications of the x,y and z dimensions of the window as well as its initial x and y positions. The specifications are encoded by prefixing a number by either "XDIM:", "YDIM:","ZDIM:", "XPOS:" or "YPOS:" respectively. Each of these attributes is prefixed by a space character to separate them from the rest of the filename. If possible these specifications will be used. If no z dimension is specified a default of 1 is assumed.

The operations permitted on a window *file* are:

equals, not equals
open, close, reopen, read bytes,
get the screen image, get/ set the colour map
read the position of the window's pointer
get/ set the cursor image and get/ set the cursor information.

Data read from a window file is made up of pairs of 32 bit integers. The first integer is a millisecond datestamp for a keyboard event and the second integer is the X11 Key Symbol for the keyboard event. Only multiples of 8 bytes will be supplied by read bytes. If there are no unread keyboard events, read bytes will fail.

3.2.2.5 Errors and Equality

An attempt to perform an invalid operation on a given *file* kind, such as writing to a closed *file* or obtaining a raster image from a socket, is treated as an error. The particular mechanisms for indicating an error are described in chapter 4. A *file* is only equal to itself, this is checked by comparing pointer values.

3.2.3 Vectors

The *vector* data type is used to implement linear arrays of values with the same type. A *vector* value is represented by a pointer to an object with the following format:

word 0,1 object header and size
word 2..n the elements
word n+1 lower bound
word n+2 upper bound

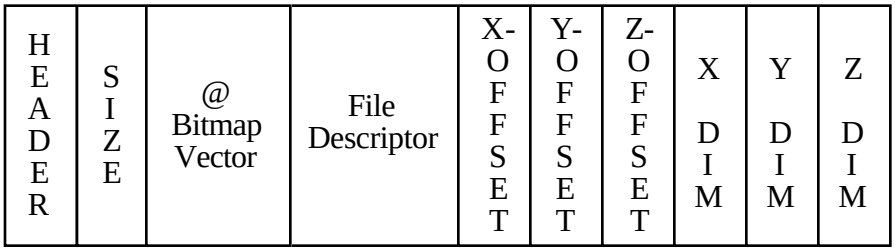
The following operations are permitted on a *vector*, described in chapter 4.

equals, not equals,
create a *vector*, read the lower bound, read the upper bound,
read an element, assign to an element,
mark the *vector* as constant and mark the *vector* as variable.

An attempt to use an index outwith the *vector* bounds, to assign to an element of a constant *vector* or to create a *vector* with an upper bound less than the lower bound, are all treated as errors. Two values of type *vector* are equal if they are pointers to the same *vector*.

3.2.4 Images

The *image* data type is used to represent an aliased area of a rectangular array of *pixels*. The aliased area is itself represented by a pointer to an object with the following format:



word 0,1	object header and size
word 2	pointer to the bitmap *vector*
word 3	pointer to the *file* descriptor (if a cursor or screen of a raster device otherwise **nilfile**)
word 4	X offset into the bitmap *vector*
word 5	Y offset into the bitmap *vector*
word 6	Z offset into the bitmap *vector*
word 7	X dimension of the *image*
word 8	Y dimension of the *image*
word 9	Z dimension of the *image*

There are 3 kinds of raster *image* supported by the Persistent abstract machine, raster displays, cursors of raster displays and memory rasters. Each bitmap *vector* contains a flag word to indicate which kind of raster it represents. This enables the abstract machine to propagate changes to displayed *images* to the corresponding physical devices. The flags have the following values:

if a raster display	bit 0
if a cursor	bit 1
if a memory raster	bit 2

To support colour raster displays a bitmap *vector* for a raster display includes a colour map. The size of the colour map is implementation dependent but would normally include an entry for each *pixel* value that can be displayed. For example, a colour map for an *image* of depth 8 would contain 256 entries in its colour map. A different strategy must be employed for true colour displays which have pixels with 24 or more planes. The bitmap *vector* for an *image* is laid out as follows:

word 0,1	object header and size
word 2	X dimension of the bitmap
word 3	Y dimension of the bitmap
word 4	Z dimension of the bitmap
word 5	length of a scan line in 32 bit words
word 6	type flags for the bitmap
word 7..n-1	bits that represent the *image*'s pixels
word n..m	the colour map if the raster *image* is for a raster display
word m+1	lower bound
word m+2	upper bound

The *pixels* of an image are laid out as follows. The *image* is separated out into its planes. The first plane in the representation is plane 0 of the *image*, the second plane is plane 1 of the *image* and so on. Each plane is separated into scan lines. The first scan line in a plane is the top scan line of the *image* and the last scan line is the bottom scan line of the *image*. Each scanline is separated into 1

bit *pixels* since a plane is only 1 bit deep. A scanline consists of an integral number of 32bit words with the first 32 pixels in the first word, the second 32 pixels in the second word and so on. Within a 32bit word the first *pixel* is represented by bit 31 and the last *pixel* by bit 0.

The operations permitted on a raster *image* are as follows, described in chapter 4.

create an *image*, equals, not equals,
copy *pixels* from another *image* using a raster combination rule,
draw a line of *pixels* using a raster combination rule,
select an area of *pixels* in the X and Y dimensions,
select a range of planes in the Z dimension,
mark the bitmap *vector* as constant and
mark the bitmap *vector* as variable.

An attempt to create an *image* with any non positive dimensions, select an non existent part of an *image*, or update an *image* whose bitmap is constant, are all treated as errors. Two values of type *image* are equal if they have the same pointer value, that is they both point to the same alias object.

3.2.5 Structures

The *structure* data type supports objects containing an arbitrary collection of data types. A *structure* value is represented by a pointer to an object with the following format:

word	0,1	object header and size
word	2..m	the pointer fields
word	m+1..n	the nonpointer fields
word	n+1..	constancy bitmap

Every *structure* is assumed to contain a constancy bitmap of one bit per word. It should be checked whenever a word in a *structure* is to be updated. However updates to the words containing the bitmap are not checked to allow the constancy of fields to be altered. For *structure* fields consisting of more than one word, only the constancy bit for the first word of the field is used. For a *structure* of length L the starting word (S) of the bitmap can be calculated as follows:

S = L - (L + 32) div 33

The word (W) within the bitmap containing the bit for a given field index (I) and the field's bit (B) within that word can be calculated as follows:

W = I div 32
B = I rem 32

To test if a field is constant bit B in word S + W of the *structure* is tested. The field is constant if the bit is set. Note that the bits are numbered in increasing significance from bit 0 to bit 31.

The operations permitted on a *structure* are as follows, described in chapter 4.

equals, not equals,
create a *structure*,
read a *structure* field, assign to a *structure* field and
mark a *structure* field as constant.

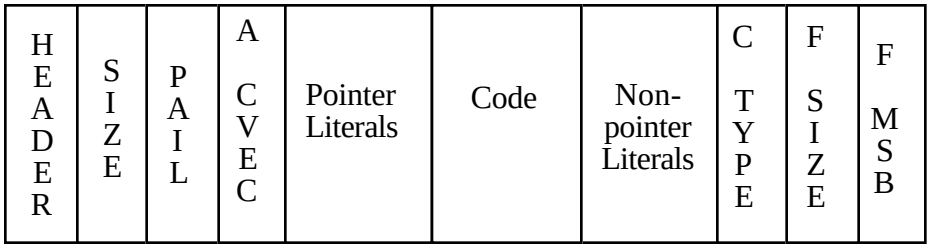
An attempt to assign to a *structure* field that is marked as constant is treated as an error. Two values of the *structure* data type are equal if they have the same pointer value, that is they both point to the same *structure* object.

3.2.6 Procedures

Procedures are the only pointer data type that is represented by two object addresses. The first is a pointer to an object containing executable code (a code vector) and the other is a pointer to the *procedure*'s static environment (a stack frame). Together the two pointers form the closure of the *procedure*. A closure is formed when a *procedure* declaration is executed by taking the pointer to the code vector and combining it with a pointer to the current frame (LFB). The first pointer in a closure is the code vector and the closure is always addressed by addressing the first pointer.

3.2.6.1 Code Vectors

A code vector contains the executable code for a *procedure*, any scalar, pointer or mixed data type literals used by the *procedure*, a PAIL description of the *procedure* and the size of stack frame required when the *procedure* is executing. The format of a code vector is as follows:



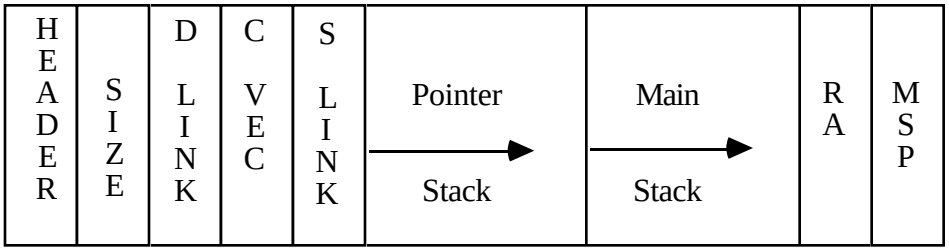
word 0,1	object header and size
word 2	a pointer to the pail tree for the code vector's *procedure* (PAIL)
word 3	a pointer to an alternative code vector (A CVEC), this has the same functionality but contains different code
word 4..l	any pointers to objects that are used by the code vector's *procedure*
word l+1..m	the code to be executed
word m+1..n	any non-pointer literals that are used by the code vector's *procedure*
word n+1	the type of code, 0 if the code is Napier code (C TYPE)
word n+2	the size of the frame (in words) to be created when the code vector's *procedure* is applied (F SIZE)
word n+3	the offset to the main stack (in words) for the frame (F MSB)

The code to be executed is in the form of a byte stream with 4 bytes per word. Within a word the most significant byte is the first byte of code, the next most significant byte is the second byte of code, the next most significant byte is the third byte of code and the least significant byte is the fourth byte of code.

3.2.6.2 Frames

A stack frame contains a pointer stack, a main stack, the relative positions of the stack tops with respect to the start of the frame and the relative position of the next instruction with respect to the

start of the code vector. The relative positions are used to calculate the values of LMSP, LPSP and CP when a frame becomes the local frame. Similarly, the relative positions are recalculated whenever a frame ceases to be the local frame or a store operation is performed that may move the local frame or the code vector. The format of a stack frame is as follows:



word 0,1	object header and size
word 2	the dynamic link (D LINK)
word 3	a pointer to the code vector for the frame's *procedure* (C VEC)
word 4	the static link for the frame's *procedure* (S LINK)
word 5..m	the pointer stack for the frame's *procedure*
word m+1..n	the main stack for the frame's *procedure*
word n+1	the resume address for the frame's *procedure* (RA), the saved offset (in bytes) of CP from the start of the *procedure*'s code vector
word n+2	the saved offset (in words) of the LMSP from the LFB (MSP)

3.2.6.3 Operations and Equality

The operations permitted on a *procedure* are as follows, described in chapter 4.

create a closure,
equals, not equals,
procedure application and *procedure* return.

Two *procedure* values are equal if they describe the same *procedure* closure.

3.2.7 Abstract Data Types

Abstract data types are similar to *structure* types but they may contain special fields of *witness* type whose actual type is not known at compile time. Each of these special fields is represented by two pointers and two non pointers to enable any of the scalar, pointer or mixed data types to be present. In addition, for each *witness* type a pointer field records a type representation for the *witness* type and a non-pointer field is records a dynamic tag for the *witness* type. The format of an *abstract data type* is as follows:

word 0,1	object header and size
word 2..k	the pointer fields
word k+1..l	the type representations for the *witness* types
word l+1..m	the non-pointer fields
word m+1..n	the dynamic tags for the *witness* types
word n+1..	constancy bitmap

The operations permitted on *abstract data types* are the same as for *structures*.

3.3 Mixed Data Types

The mixed data types are used to support union types whose values may be one or more of the data types supported by the abstract machine. The *variant* data type is used to support finite

discriminated unions whose range of data types are known at compile time whereas the *any* data type is used to support the infinite union of all data types.

3.3.1 Variants

A *variant* consists of three parts, a value, a label and a type encoding. The value is represented by a single pointer. However, if the value is not a single pointer then it is wrapped in a heap object and the pointer to the heap object is used. The format of the wrapper object is as follows:

word 0,1	object header and size
word 2..	the value, pointers before non-pointers

The label and type encoding are held in an integer word. Thus a *variant* consists of an integer word and a single pointer each of which is addressed separately. The label is represented by a branch number assigned by the compiler and is held in bits 8-31 of the integer word. A branch number may be any integer in the range 0 to $2^{24} - 1$.

The type encoding, known as a dynamic tag, is used to differentiate each of the data types that are supported by the abstract machine. It describes the size of the value, in integer words and pointers, and includes an additional number to differentiate data types of the same size. The type encoding forms an 8 bit number held in bits 0-7 of the integer word. It is encoded as follows (lower numbered bits are less significant):

bit 0,1	number of integer words
bit 2,5	used to distinguish data types of the same size
bit 6,7	number of pointers

This results in the following encoding for the dynamic tags of Napier objects:

object	bit pattern	integer code
integer	00000001	1
boolean	00000101	5
pixel	00001001	9
real	00000010	2
string	01000100	68
vector, structure, abstract data type, file, image	01000000	64
procedure	10000000	128
variant	01000001	65
any	10000010	130
witness	10000110	134

The operations permitted on a *variant* are as follows, described in chapter 4.

equals, not equals,
inject a value into a *variant* and project a value from a *variant*.

Two *variant* values are equal if their integer words have the same value and if the values represented by their pointers are equal. The dynamic tag contained in the integer words describes how the values should be compared.

3.3.2 Anys

Values of the data type *any* are held as two integer words and two pointers. The address of an *any* value is the address of its first integer word and the address of its first pointer. The actual value injected into an *any* is held in the form of a variant value with a branch number of -1. This variant occupies the first integer word and the first pointer. The second pointer points to the type of the injected value and is supplied by the compiler. The second integer word is padding to permit an *any* to be manipulated as if it were a double word double pointer value.

The operations permitted on an *any* are as follows, described in chapter 4.

> equals, not equals,
> inject a value into an *any* and project a value from an *any*.

Two *any* values are equal if the variant representations of their values are equal and the type representations supplied by the compiler are equivalent. It should be noted that the type checking phase of comparing two *any* values must be performed by the comparison procedure held in the abstract machine's root object.

4 Persistent Abstract Machine Code

The Persistent abstract machine code, PAM-code, is designed to support languages that map into PAIL. The code generated for each PAIL construct may be found in [5]. Here the individual instructions are described. They fall naturally into groups.

Typed instructions have an encoded name with the following convention.

ib	integer,pixel or boolean
r	real
s	string
p	file, vector, image, structure, abstract data type
pr	procedure
var	variant
any	any
poly	polymorphic object

The Persistent abstract machine supports polymorphic operations. These operations consult a word on the main stack with encoded information about the concrete type on which they operate. This information allows polymorphic operations to delay the decision about which actions to perform until runtime. The polymorphic instructions use the type encoding given in section 3.3.1. Note that the encoding is referred to as the *dynamic tag* in the instruction descriptions.

Non type dependant instructions are encoded according to the size of the objects on which they operate and on which stack they reside. These instructions are encoded using the following convention.

w	word on main stack
dw	double word on main stack
p	word on pointer stack
dp	double word on pointer stack
wp	word on main stack and word on pointer stack
dwdp	double word on main stack and double word on pointer stack

Some instructions have special forms that allow for cases which deserve optimisation. These instructions are appended with the letter S.

The length of instruction parameters are in the following units:

byte	8 bits
short	2 bytes

The interpretation of instruction parameters is as the follows:

byte	an 8 bit integer, unsigned unless used with the literal integer instruction
short	an unsigned 16 bit integer, the first byte is most significant

All instruction codes are one byte long.

4.1 Jumps

All the jump offsets are relative to the location following the jump offset. The jump offset is measured in bytes.

fjump(l : short)

Op-Code 1

Description

Jump forwards l bytes.

jumpf(l : short)

Op-Code 2

Description

if the top main stack element is **false**
do Jump forwards l bytes.
Pop the main stack.

bjump(l : short)

Op-Code 3

Description

Jump backwards l bytes.

bjumpt(l : short)

Op-code 4

Description

if the top main stack element is **true**
do Jump backwards l bytes.
Pop the main stack.

jumpff(l : short)

Op-Code 5

Description

if the top stack element is **false**
then Jump forwards l bytes
else Pop the main stack.

jumptt(l : short)

Op-Code 6

Description

if the top element is **true**
then Jump forwards l bytes
else Pop the main stack.

fortest(fs : short, msb : short, l : short)

Op-Code 7

Parameters

fs:	frame size
msb	main stack base offset
l	label

Description

The for loop increment is on top of the main stack,
The for loop limit is below the increment on the main stack and
The control constant is below the limit on the main stack.
if the increment is negative and the control constant is less than the limit **or**
the increment is positive and the control constant is greater than the limit
then Pop the top 3 stack elements and jump forwards l bytes.
else Perform a block enter instruction with parameters fs and msb.
Push a copy of the control constant onto the new frame's main stack.

forstep(l : short)

Op-Code 8

Description

Perform a block.exit.v.
Add the for loop increment to the for loop control constant.
Jump backwards l bytes, (to the fortest instruction).

fortestS(l : short)

Op-Code 9

Description

The for loop increment is on top of the main stack,
The for loop limit is below the increment on the main stack and
The control constant is below the limit on the main stack.
if the increment is negative and the control constant is less than the limit **or**
the increment is positive and the control constant is greater than the limit
do Pop the top 3 stack elements and jump forwards l bytes.

forstepS(l : short)

Op-Code 10

Description

Add the for loop increment to the for loop control constant.
Jump backwards l bytes, (to the fortestS instruction).

cjump.ib,r,s,p,pr(l : short)

Op-Codes

cjump.ib	11	cjump.r	12
cjump.s	13	cjump.p	14
cjump.pr	15		

Description

if the top two values of the relevant stack are equal
then Pop both values from the stack and jump forwards l bytes
else Pop the value at the top of the stack.

The rules for comparing two values are given in section 4.15.

cjump.var,any(l1 : short, l2 : short)

Op-Codes

cjump.var	16	cjump.any	18

Description

if the top two values on the stacks are equal
then Pop both values and jump forwards l2 bytes.
else **if** the top two values on the stacks are not equal
then Pop the value at the top of the stacks and jump forwards l1 bytes.
else **if** equality cannot be determined because one or more pairs of anys must be
type checked
do Pop the value at the top of the stacks
Call the any comparison procedure held in the root object.
The first pair anys to be type checked are passed as parameters.

The rules for comparing two stack elements are described in section 4.15.
The cjump.var and cjump.any instructions should be immediately followed by the following abstract machine instructions:

Comparing	variants	anys	
	jumpf(l1)	jumpf(l1)	jump to l1 if **false**.
	retract(1,1)	retract(2,2)	pop the value on top of stacks.
	fjump(l2)	fjump(l2)	jump to l2 - **true**.

cjump.poly(l1 : short, l2 : short)

Op-Code

cjump.poly 17

Description

The dynamic tag for the values being compared on top of the main stack.
The values to be compared are at the top of both stacks, but below the dynamic tag.
They are both padded to be two integer words and two pointers each.
Pop the dynamic tag from the main stack.
Compare the values using the rules determined by the dynamic tag,
if the values are equal
then Pop both values from the stacks and jump forwards l2 bytes.
else **if** the values are not equal
then Pop the value at the top of the stacks and jump forwards l1 bytes.
else **if** equality cannot be determined because one or more pairs of anys must be
type checked
do Pop the value at the top of the stacks
Call the any comparison procedure held in the root object.
The first pair anys to be type checked are passed as parameters.

The cjump.poly instruction should be immediately followed by the following abstract machine instructions:

jumpf(l1)	jump to l1 if **false**.
retract(2,2)	pop the value on top of stacks.
fjump(l2)	jump to l2 - **true**.

4.2 Stack Load and Assignment

Stack Load

These instructions are used to push the field of an object onto the top of a stack. The object may be the root object, local frame or any object with a pointer on the local frame's pointer stack. A separate instruction exists for each form. Different instructions are also used for the separate stacks. These instructions have a parameter (**d**) which is the displacement (in words) of the field from the base of its object. If the field is in separate pointer and non pointer parts then there are two displacements (**d1** and **d2**) which are the displacements (in words) of each part of the field from the base of its object. The root form of these instructions uses the root object. The local form of these instructions uses the local frame. The load form of the instruction has an additional parameter (**f**) which is the offset (in words) from the local frame base to the pointer to the object.

Stack Assignment

These instructions are used to assign to the field of an object. The value being assigned is always on the top of the appropriate stack and is always popped after the assignment. The object assigned to may be the root object, local frame or any object with a pointer on the local frame's pointer stack. The addressing modes are the same as the stack load instructions described above.

root(d : short)

Op-Codes

wroot	19	dwroot	20
proot	21	dproot	22

Description

Push word d of the root object onto the appropriate stack.
if the instruction is dwroot or dproot
do Push word d + 1 of the root object onto the appropriate stack.

root(d1 : short, d2 : short)

Op-Code

wproot	23	dwdproot	24

Description

Push word d1 of the root object onto the main stack.
Push word d2 of the root object onto the pointer stack.
if the instruction is dwdproot
do Push word d1 + 1 of the root object onto the main stack.
Push word d2 + 1 of the root object onto the pointer stack.

root.ass(d : short)

Op-Code

wroot.ass	25	dwroot.ass	26
proot.ass	27	dproot.ass	28

Description

if the instruction is dwroot.ass or dproot.ass
do Copy the word on top of the main stack to word d + 1 of the root object.
Pop the word from the main stack.
Copy the word on top of the appropriate stack to word d of the root object.
Pop the word from the appropriate stack.

root.ass(d1 : short, d2 : short)

Op-Code

wproot.ass	29	dwdproot.ass	30

Description

if the instruction is dwdproot.ass
do Copy the word on top of the main stack to word d1 + 1 of the root object.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 + 1 of the root object.
Pop the word from the pointer stack.
Copy the word on top of the main stack to word d1 of the root object.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 of the root object.
Pop the word from the pointer stack.

local(d : short)

Op-Codes

wlocal	31	dwlocal	32
plocal	33	dplocal	34

Description

Push word d of the local frame onto the appropriate stack.
if the instruction is dwlocal or dplocal
do Push word d + 1 of the local frame onto the appropriate stack.

local(d1 : short, d2 : short)

Op-Code

wplocal	35	dwdplocal	36

Description

Push word d1 of the local frame onto the main stack.
Push word d2 of the local frame onto the pointer stack.
if the instruction is dwdplocal
do Push word d1 + 1 of the local frame onto the main stack.
Push word d2 + 1 of the local frame onto the pointer stack.

local.ass(d : short)

Op-Code

wlocal.ass	37	dwlocal.ass	38
plocal.ass	39	dplocal.ass	40

Description

if the instruction is dwlocal.ass or dplocal.ass
do Copy the word on top of the main stack to word d + 1 of the local frame.
Pop the word from the main stack.
Copy the word on top of the appropriate stack to word d of the local frame.
Pop the word from the appropriate stack.

local.ass(d1 : short, d2 : short)

Op-Code

wplocal.ass	41	dwdplocal.ass	42

Description

if the instruction is dwdplocal.ass
do Copy the word on top of the main stack to word d1 + 1 of the local frame.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 + 1 of the local frame.
Pop the word from the pointer stack.
Copy the word on top of the main stack to word d1 of the local frame.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 of the local frame.
Pop the word from the pointer stack.

load(f : short, d : short)

Op-Code

wload	43	dwload	44
pload	45	dpload	46

Description

The source object is pointed to by word f of the local frame.
Push word d of the source object onto the appropriate stack.

load(f : short, d1 : short, d2 : short)

Op-Code

wpload	47	dwdpload	48

Description

The source object is pointed to by word f of the local frame.
Push word d1 of the source object onto the main stack.
Push word d2 of the source object onto the pointer stack.
if the instruction is dwdpload
do Push word d1 + 1 of the source object onto the main stack.
Push word d2 + 1 of the source object onto the pointer stack.

assign(f : short, d : short)

Op-Code

wassign	49	dwassign	50
passign	51	dpassign	52

Description

The destination object is pointed to by word f of the local frame.
Copy the word on top of the appropriate stack to word d of the destination object.
Pop the word from the appropriate stack.

assign(f : short, d1 : short, d2 short)

Op-Code

wpassign	53	dwdpassign	54

Description

The destination object is pointed to by word f of the local frame.
if the instruction is dwdpassign
do Copy the word on top of the main stack to word d1 + 1 of the destination object.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 + 1 of the destination object.
Pop the word from the pointer stack.
Copy the word on top of the main stack to word d1 of the destination object.
Pop the word from the main stack.
Copy the word on top of the pointer stack to word d2 of the destination object.
Pop the word from the pointer stack.

4.3 Polymorphic Operations

Two instructions are provided to convert a value on a stack to and from a uniform double-word, double-pointer representation. The uniform representation is used within polymorphic procedures.

contract.poly(ms : short, ps : short)

Op-Code 55

Description

The dynamic tag for the value being contracted is on top of the main stack.
The polymorphic value is at either word ms and ps of the local frame.
Word offset ps is for the pointer stack and word offset ms is for the main stack.
Pop the dynamic tag from the main stack.
From the tag, calculate where the padding words reside on both stacks.
Slide the contents of the stacks above the padding downwards to overwrite the padding.
Decrement the main stack pointer by the size of the main stack padding.
Decrement the pointer stack pointer by the size of the pointer stack padding.

expand.poly(ms : short, ps : short)

Op-Code 56

Description

The dynamic tag for the value being expanded is on top of the main stack.
The value is at either word ms and / or ps of the local frame, according to its type.
Word offset ps is for the pointer stack and word offset ms is for the main stack.
Pop the dynamic tag from the main stack.
From the tag, calculate where the padding words should be in both stacks.
Slide the contents of the stacks above the value to be expanded upwards to make room for the padding.
Insert the padding - padding **nil**s on the pointer stack and 0s on the main stack
Increment the main stack pointer by the size of the main stack padding.
Increment the pointer stack pointer by the size of the pointer stack padding.

4.4 Stack Duplicate Operations

These are used to duplicate the element on top of a stack.

dup

Op-Codes

wdup	57	dwdup	58
pdup	59	dpdup	60
wpdup	61	dwdpdup	62

Description

Push a copy of the value at the top of the appropriate stack onto the same stack. Note that in the wp and dwdp cases this involves copying **both** stack tops.

4.5 Stack Retract Operations

These are used for non-retentive block exits and stack erases.

retract(ms : short, ps : short)

Op-Codes

wretract	63	dwretract	64
pretract	65	dpretract	66
wpretract	67	dwdpretract	68
		retract	70

Description

If non-void copy and then pop the item on top of the appropriate stack.
Pop ms words from the main stack.
Pop ps words from the pointer stack.
If non-void push the copied value onto the appropriate stack.
Note that in the wp and dwdp cases this involves copying **both** stack tops.

4.6 Block Entry and Exit

These instructions are used for each block that may require a stack frame to be retained by virtue of containing a nested procedure declaration. In effect, each block is treated as an in-line procedure call.

block.enter(fs : short, msb : short)

Op-Code 71

Parameters

fs: frame size
msb: main stack base offset

Description

Save the offset (in words) of LMSP from LFB in the current frame.
Set the number of pointers in the current frame to be LPSP - LFB - 2 (in words)
Create an object of size fs (in words), this is the new frame.
Set LMSP to the main stack base of the new frame, msb is the offset to the start of main stack (in words) from the base of the new frame.
Set LPSP to the word following size field.
Push the dynamic link (the current frame LFB) onto the new pointer stack.
Push the pointer to the current code vector onto the pointer stack.
Push the static link (the current frame) onto the new pointer stack.
Set LFB to point to the new frame.

block.exit

Op-Codes

wblock.exit	72	dwblock.exit	73
pblock.exit	74	dpblock.exit	75
wpblock.exit	76	dwdpblock.exit	77
		block.exit	79

Description

Copy and pop the result of the block at the top of the appropriate stack.
Set the number of pointers in the current frame to be LPSP - LFB - 2 (in words).
if the trace bit is set,
then Set the trace bit in the frame pointed to by the dynamic link.
else Free the heap space allocated to the exiting block's frame.
Set LFB to the dynamic link of the current frame, the new current frame.
Set LMSP to be LFB + the saved offset for LMSP held in the current frame.
Set LPSP so that the last pointer in the current frame is at the top of the pointer stack (LFB + #pntrs + 2)
Push the result of the block onto the appropriate stack.
Note that in the wp and dwdp cases this involves copying **both** stack tops.

4.7 Procedure Entry and Exit

The instruction sequence to call a procedure is:

1. load closure
2. evaluate the parameters
3. apply

apply(ms : short, ps : short)

Op-Code 80

Description

Save the offset (in bytes) of CP from the start of the current code vector, in the current frame (the resume address).
The main stack parameters start at word ms in the current frame.
The code vector for the procedure being applied, the new code vector, is at word ps in the current frame.
Above the new code vector on the pointer stack is the static link for the procedure being applied, the new static link.
Above the new static link on the pointer stack are the pointer stack parameters.
Create an object to be the frame for the procedure being applied, its size is held in the new code vector (in words), this is the new frame.
Save the offset (in words) of LMSP from LFB in the current frame as ms,(forces the main stack parameters to be removed).
Set the number of pointers in the current frame to be ps - 2,(forces the procedure closure and pointer stack parameters to be removed).
Set LMSP to the main stack base of the new frame, the offset to the start of the main stack (in words) from the base of the new frame is held in the new code vector.
Set LPSP to word following size field.
Push the dynamic link (the current frame) onto the new pointer stack.
Push the pointer to the new code vector onto the pointer stack.
Push the new static link onto the new pointer stack.
Push the pointer stack parameters onto the new pointer stack.
Push the main stack parameters onto the new main stack.
Set LFB to point to the new frame.
Set CP to the start of the abstract machine code in the new code vector.

return

Op-Codes

wreturn	81	dwreturn	82
preturn	83	dpreturn	84
wpreturn	85	dwdpreturn	86
		return	88

Description

if the instruction is return and the dynamic link is **nil**
then **halt the abstract machine**.
else Copy and pop the result of the procedure at the top of the appropriate stack.
Set the number of pointers in the current frame to be LPSP - LFB - 2 (in words).
if the trace bit is set
then Set the trace bit in the frame pointed to by the dynamic link.
else Free the heap space allocated to the returning procedure's frame.
Set LFB to the dynamic link of the current frame, the new current frame.
Set LMSP to be LFB + the saved offset for LMSP held in the current frame.
Set LPSP so that the last pointer in the current frame is top of the pointer stack.
if the trace bit is set
do Set the dynamic link in the returning procedure's frame to be **nil**.
Push the result of the procedure onto the appropriate stack.
Set CP to be the start of the current code vector + the saved offset for CP held in the current frame.

return.poly

Op-Code 87

Description

Pop the dynamic tag for the value being returned from the top of the main stack.
Contract the value on the top of the stacks from two integer words and two pointers to its actual size as determined by the dynamic tag.
Perform the return operation applicable to the value's actual size.

current.frame

Op-Code 97

Description

Push a pointer to the current frame (LFB) onto the pointer stack.
Set the trace bit in the current frame.

4.8 Image Operations

These instructions manipulate raster images and raster windows.

makepixel(n : byte)

Op-Code 98

Description

Sum the depths of the n pixels on top of the main stack.
Create a new pixel of the combined depth.
Copy the planes of the pixels on the main stack into the new pixel.
The lowest pixel on the main stack represents the first planes of the new pixel.
The pixel on top of the main stack represents the last planes of the new pixel.
Pop the n pixels from the main stack.
if the total depth of the new pixel is greater than 24
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g02PixelOverflow in the error structure.
The pixel formed by the first 24 planes is passed as a parameter.
else Push the new pixel onto the main stack.

subpixel

Op-Code 99

Description

Pop the depth of the new pixel from main stack
Pop the start plane of the new pixel from the main stack.
The subscripted pixel is now on top of the main stack.
Compare the start plane and depth of the new pixel with the subscripted pixel.
if the bounds are illegal
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Pop the subscripted pixel from the main stack.
Call the procedure g03SubPixel in the error structure.
The subscripted pixel, start plane and depth are passed as parameters.
else **if** the depth of the new pixel is less than the depth of the subscripted pixel
do Create the new pixel and set its depth.
Copy the selected planes from the subscripted pixel to the new pixel.
Pop the subscripted pixel from the main stack.
Push the new pixel onto the main stack.

makeimage

Op-Code 100

Description

Pop the initialising pixel for the image from the main stack.
Lookup the depth of the pixel, Z.
Pop the Y dimension of the image from the main stack.
Pop the X dimension of the image from the main stack.
if either X or Y dimension is less than or equal to 0
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g04MakeImage in the error structure.
The X dimension, Y dimension and initialising pixel are passed as parameters.
else Create an image descriptor for an image with dimensions X by Y by Z.
Create a vector of integers to hold the image's pixels.
Initialise the vector of integers by replicating the initialising pixel.
Place a pointer to the vector of integers in the image descriptor.
Push a pointer to the image descriptor onto the pointer stack.

subimage

Op-Code 101

Description

The image descriptor being subscripted is on the pointer stack.
Pop the number of planes from the main stack.
Pop the start plane of the new image (numbered from 0) from the main stack.
Compare the start plane and depth with the bounds of the subscripted image.
if the bounds are illegal
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Pop the subscripted descriptor from the pointer stack.
Call the procedure g05SubImage in the error structure.
The image, start plane and depth are passed as parameters.
else
if the bounds are a subset of the subscripted descriptor
do Create a copy of the subscripted descriptor.
Pop the subscripted descriptor from the pointer stack.
Push the created copy onto the pointer stack.
Increment the copy's depth offset by the start plane.
Set the copy's depth to be the number of planes.

limAt

Op-Code 102

Description

Pop the new X offset from the main stack.
Pop the new Y offset from the main stack.
The subscripted image descriptor is on top of the pointer stack.
Compare the new X and Y offsets with the dimensions of the subscripted image.
if the new X and Y offsets are outwith the dimensions of the subscripted image
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Pop the subscripted image descriptor from the pointer stack.
Call the procedure g06LimitAt in the error structure.
The image, X offset and Y offset are passed as parameters.
else Create a copy of the subscripted image descriptor.
Pop the subscripted image descriptor from the pointer stack.
Push the copy onto the pointer stack.
Decrement the X dimension of the copy by the new X offset
Add the new X offset to the copy's X offset.
Decrement the Y dimension of the copy by the new Y offset
Add the new Y offset to the copy's Y offset.

limAtBy

Op-Code 103

Description

Pop the new X offset from the main stack.
Pop the new Y offset from the main stack.
Pop the new X dimension from the main stack.
Pop the new Y dimension from the main stack.
The subscripted image descriptor is on top of the pointer stack.
Compare the new offsets and dimensions with the dimensions of the subscripted image.
if the new offsets and dimensions are outwith the dimensions of the subscripted image **or** either of the dimensions is less than or equal to 0
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Pop the subscripted image descriptor from the pointer stack.
Call the procedure g07LimitAtBy in the error structure.
The image, X offset, X dimension, Y offset and Y dimension are passed as parameters.
else Create a copy of the subscripted image descriptor.
Pop the subscripted image descriptor from the pointer stack.
Push the copy onto the pointer stack.
Set the X dimension of the copy to be the new X dimension
Add the new X offset to the copy's X offset.
Set the Y dimension of the copy to be the new Y dimension
Add the new Y offset to the copy's Y offset.

rasterOp

Op-Code 104

Description

Pop the destination image descriptor from the top of the pointer stack.
Pop the source image descriptor from the top of the pointer stack.
Pop the rasterop rule to be used from the top of the main stack.
The size of the destination image dictates the clipping area for the source image.
if the destination bitmap is marked as constant
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g08ConstantImage in the error structure.
The destination image is passed as a parameter.
else Perform the raster.op from source onto destination using the specified rule.
if the destination image is part of a cursor or screen
do **if** the image's file descriptor is open
then propagate the changes to the cursor or screen
else **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure x1ClosedWindow in the error structure.
The file descriptor and destination image are passed as parameters.

Notice that the image may be a cursor or screen to which any operations must be propagated. Such an image contains an open file descriptor in its image descriptor.

The interpretation of the raster rules is as follows: the rules are encoded as integers:

0	S **and** ~S	8	S **and** D
1	~(S **or** D)	9	~S **xor** D
2	~S **and** D	10	D
3	~S	11	~S **or** D
4	S **and** ~D	12	S
5	~D	13	S **or** ~D
6	S **xor** D	14	S **or** D
7	~(S **and** D)	15	S **or** ~S

raster.line

Op-Code 105

Description

Pop destination image descriptor from the top of the pointer stack.
Pop rasterop rule to be used from the top of the main stack.
Pop the pixel value to be used to draw the line from the top of the main stack.
Pop the Y coordinate of the last point of the line from the top of the main stack..
Pop the X coordinate of the last point of the line from the top of the main stack..
Pop the Y coordinate of the first point of the line from the top of the main stack..
Pop the X coordinate of the first point of the line from the top of the main stack..
if the destination bitmap is marked as constant
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g08ConstantImage in the error structure.
The destination image is passed as a parameter.
else Raster the supplied pixel onto the pixels forming the specified line using the specified rasterop rule.
if the destination image is part of a cursor or screen
do **if** the image's file descriptor is open
then propagate the changes to the cursor or screen
else **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure x1ClosedWindow in the error structure.
The file descriptor and destination image are passed as parameters.

Notice that the image may be a cursor or screen to which any operations must be propagated. Such an image contains a file descriptor in its image descriptor.

get.screen

Op-Code

get.screen 106

Description

Pop the file descriptor from the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g09GetScreen in the error structure.
The file descriptor is passed as a parameter.
else Push the screen field of the descriptor onto the pointer stack.

locator

Op-Code 107

Description

Pop the pointer to the destination vector from the pointer stack.
Pop the file descriptor from the top of the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g10Locator in the error structure.
The file descriptor and the vector are passed as parameters.
else Copy the locator information for the file into the vector.

The elements of the vector are filled in as follows:
element 1: the X dimension of the window,
element 2: the Y dimension of the window,
element 3: the locator X position,
element 4: the locator Y position,
element 5: the millisecond datestamp of the event being reported,
element 6: the state of button 1,
element n: the state of button n-4,
if the vector has more elements than the information available the extra are ignored,
if it has too few elements only the ones supplied are filled in,
if there are no unread locator events the datestamp is -1 and the previous event is reported.
The X and Y positions are relative to the lower left of the window.

colour.map

Op-Code 108

Description

Pop the colour map entry from the top of the main stack.
Pop the pixel from the top of the main stack.
Pop the file descriptor from the top of the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g11ColourMap in the error structure.
The file descriptor, pixel and colour map entry are passed as parameters.
else Set the colour map entry for the specified pixel to be the specified entry.

If the pixel parameter has more planes than the window, the additional planes are ignored. Alternatively, if the pixel has fewer planes than the window, the missing planes are treated as **off**. If the window has no colourmap then the instruction is a non operation.

colour.of

Op-Code 109

Description

Pop the pixel from the top of the main stack.
Pop the file descriptor from the top of the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g12ColourOf in the error structure.
The file descriptor and pixel are passed as parameters.
else Push the colour map entry for the specified pixel onto the main stack.

If the pixel parameter has more planes than the window, the additional planes are ignored. Alternatively, if the pixel has fewer planes than the window, the missing planes are treated as **off**. If the window has no colourmap the result placed on the stack is undefined.

get.cursor

Op-Code 110

Description

Lookup the type of the file descriptor on top of the pointer stack.
Pop the file descriptor from the pointer stack.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g13GetCursor in the error structure.
The file descriptor is passed as a parameter.
else Push the cursor field of the file descriptor onto the pointer stack.

set.cursor

Op-Code 111

Description

Pop the image descriptor from the pointer stack.
Pop the file descriptor from the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g14SetCursor in the error structure.
The file descriptor and image descriptor are passed as parameters.
else Set the cursor field of the file descriptor to be the specified image descriptor.

get.cursor.info

Op-Code 112

Description

Pop the pointer to the destination vector from the pointer stack.
Pop the file descriptor from the top of the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g15GetCursorInfo in the error structure.
The file descriptor and the vector are passed as parameters.
else Copy the cursor information for the device into the vector.

The elements of the vector are filled in as follows:
element 1: the cursor's X position,
element 2: the cursor's Y position,
element 3: the rasterop rule used to display the cursor
if the vector has more than 3 elements the extra are ignored,
if the vector has less than 3 only the ones supplied are filled in.

set.cursor.info

Op-Code 113

Description

Pop the pointer to the source vector from the pointer stack.
Pop the file descriptor from the top of the pointer stack.
Lookup the type of the file descriptor.
if the file is not an open window
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g16SetCursorInfo in the error structure.
The file descriptor and the vector are passed as parameters.
else Copy the cursor information for the device from the vector.

The elements of the vector are used as follows:
element 1: specifies the cursor's X position,
element 2: specifies the cursor's Y position,
element 3: specifies the rasterop rule used to display the cursor.
if the vector has more than 3 elements the extra are ignored,
if the vector has less than 3 only the ones supplied are used.

get.pixel

Op-Code 114

Description

Pop the image descriptor from the pointer stack.
Pop the X position of the pixel being looked up.
Pop the Y position of the pixel being looked up.
Compare the pixel's position with the dimensions of the image descriptor.
if the pixel is outwith the dimensions of the image descriptor
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g00GetPixel in the error structure.
The image, X position and Y position are passed as parameters.
else Push the pixel at position X,Y in the image onto the main stack.

set.pixel

Op-Code 115

Description

Pop the image descriptor from the pointer stack.
Pop the new value for the pixel being set.
Pop the X position of the pixel being set.
Pop the Y position of the pixel being set.
Compare the pixel's position with the dimensions of the image descriptor.
if the pixel is outwith the dimensions of the image descriptor
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure g01SetPixel in the error structure.
The image, X position, Y position and new pixel are passed as parameters.
else Set the pixel value at position X,Y to be the pixel value specified.
if the image is part of a cursor or screen
do **if** the image's file descriptor is open
then propagate the changes to the cursor or screen
else **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure x1ClosedWindow in the error structure.
The file descriptor and image are passed as parameters.

Notice that the image may be a cursor or screen to which any operations must be propagated. Such an image contains an open file descriptor in its image descriptor.

4.9 Vector and Structure Creation Instructions

These instructions take information off the stacks to create and/or initialise heap objects.

subconst

Op-Codes

wsubconst	116	dwsubconst	117
psubconst	118	dpsubconst	119

Description

Pop the word offset, W, to the field to be made constant from the main stack.
Pop the pointer to the structure from the pointer stack.
Set the constancy bit for word W in the structure.

subconst

Op-Codes

wpsubconst	120	dwdpsubconst	121

Description

Pop the word offset, W1, to the pointer field to be made constant from the main stack.
Pop the pointer to the structure from the pointer stack.
Set the constancy bit for word W1 in the structure.
Pop the word offset, W2, to the non pointer field to be made constant from the main stack.
Set the constancy bit for word W2 in the structure.

subconst.poly

Op-Code 122

Description

The dynamic tag for the field to be made constant is at the top of the main stack.
Below the dynamic tag is the offset to use if the field to be made constant is a pointer.
Below this offset is the offset to use if the field to be made constant is a non pointer.
Both offsets are used if the field to be made constant contains pointers and non-pointers.
Pop the dynamic tag from the top of the main stack.
Contract the main stack to eliminate the unnecessary offset, as determined by the dynamic tag.
Perform the appropriate subconst instruction.

makeobject(m : short, n : short)

Op-Code 123

Description

Create an object of size m (in words) with n pointer fields.
Initialise the n pointer fields to the value **nil**.
Initialise the remaining m - n - 2 words with the integer value 0.
Push the pointer to the new object onto the pointer stack.

makeobject.poly

Op-Code 124

Description

Pop the number of pointer fields n from the main stack.
Pop the size m of the object from the main stack.
Create an object of size m (in words) with n pointer fields.
Initialise the n pointer fields to the value **nil**.
Initialise the remaining m - n - 2 words with the integer value 0.
Push the pointer to the new object onto the pointer stack.

makestruct(m : short, n : short)

Op-Code 125

Description

Create an object of size m (in words) with n pointer fields.
Copy n words from the top of the pointer stack to the object, preserving their order.
Pop n words from the pointer stack.
Copy (m - n - 2) words from the top of the main stack to the object, preserving their order.
Pop (m - n - 2) words from the main stack.
Push the pointer to the new object onto the pointer stack.

polystructaddress(nfields : short)

Op-Code 126

Description

This instruction is used to calculate the field addresses and size information for a structure whose specialised type is not known at compile time.
The main stack contains a word for the size of the structure.
Above the size is a word for the number of pointers in the structure.
Above the number of pointers is a pair of words for each field of the structure, the fields are ordered alphabetically with the last field at the top of the main stack.
Each pair of words consists of an offset to the non pointer part of the field and an offset to the pointer part of the field.
The non pointer offset is initialised to 0.
The pointer offset is initialised to the field's dynamic type.
The parameter nfields is the number of fields in the structure to allow the size word to be found.
The algorithm for calculating the correct field offsets is as follows:

1. create a variable to hold the pointer offset for the next field, initially 3 - this is to allow for the 2 word header and the pointer to the type.
2 create a variable to hold the non pointer offset for the next field, initially 0 - the non pointer offsets are patched later since they must allow for all the pointer fields.
3. for each field of the structure in alphabetic order of field name:
    a. lookup the pointer offset, the field's dynamic type
    b. overwrite the pointer offset with the next pointer offset.
    c. increment the next pointer offset by the pointer size in the dynamic type.
    d. overwrite the non pointer offset with the next non pointer offset.
    e. increment the next non pointer offset by the non pointer size in the dynamic type.
4. for each field of the structure increment the non pointer offset by the next pointer offset, the non pointer fields come after the pointer fields in a structure.
5. overwrite the number of pointers in the structure by the next pointer field offset - 2, the 2 allows for the 2 word header on the structure.
6. overwrite the size of the structure with the next pointer offset + the next non pointer offset + the size of the constancy bitmap required.

makev

Op-Codes

wmakev	127	dwmakev	128
pmakev	129	dpmakev	130
wpmakev	131	dwdpmakev	132

Description

Pop the initialising value from the top of the appropriate stack.
Pop the upper bound from the top of the main stack.
Pop the lower bound from the top of the main stack.
if the lower bound is greater than the upper bound
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v4MakeVector in the error structure ******.
The lower bound, upper bound and initialising value are passed as parameters.
else Create a vector of the appropriate size.
Initialise the vector's elements with the initial value.
Push a pointer to the vector onto the pointer stack.

It should be noted that before the error procedure v4MakeVector can be called it must first be specialised. This is also the case with calls to the other vector error procedures v1ConstantVector, v2VectorIndexAssign and v3VectorIndexSubs. In the vector indexing instructions listed below the calls requiring special attention are market with ******. The following steps are required to specialise and call one of these procedures:

Create a code vector that contains the following:
the parameters to the specialised form of the procedure,
literal instructions to load the parameters for the specialised procedure onto the stacks,
an apply instruction to call the specialised procedure and
a return instruction to return the result of the specialised procedure.
Call the procedure formed by the new code vector and a static link of **nil**, do not execute it. Call the unspecialised error procedure parameterised by a dynamic tag describing the size of the vector's elements, the error procedure will appear to have been called by the newly created procedure. When the error procedure returns, its result (the specialised procedure) will be placed on the new procedure's pointer stack and then called with the appropriate parameters.

makev.poly

Op-Code 133

Description

Pop the dynamic tag for the vector's elements from the top of the main stack.
Contract the initialising value to its actual size, as determined by the dynamic tag.
Perform the above makev operation appropriate to the value's actual size.

4.10 Vector and Structure Accessing Instructions

These instructions are generated by the compiler to index a vector or a structure. Note that the index of the vector must be checked against the bounds before the indexing is done.

subs

Op-Codes

wsubs	134	dwsubs	135
psubs	136	dpsubs	137

Description

Pop the word offset, W, to the field being looked up from the main stack.
Pop the pointer to the structure from the pointer stack.
Push word W of the structure onto the appropriate stack.
if the instruction is dwsubs **or** dpsubs
do Push word W + 1 of the structure onto the appropriate stack.

subs

Op-Code

wpsubs	138	dwdpsubs	139

Description

Pop the word offset, W1, to the pointer being looked up from the main stack.
Pop the pointer to the structure from the pointer stack.
Push word W1 of the structure onto the pointer stack.
Pop the word offset, W2, to the non pointer being looked up from the main stack.
Push word W2 of the structure onto the main stack.
if the instruction is dwdpsubs
do Push word W1 + 1 of the structure onto the pointer stack.
Push word W2 + 1 of the structure onto the main stack.

subs.poly

Op-Code 140

Description

The dynamic tag for the value being loaded is on top of the main stack.
Below the dynamic tag is the offset to use if the value being indexed is a pointer value.
Below this offset is the offset to use if the value to be assigned is a non pointer value.
Both offsets are used if the value to be assigned contains pointers and non-pointers.
Pop the dynamic tag from the main stack.
Contract the main stack to eliminate the unnecessary offset, as determined by the dynamic tag.
Perform the subs instruction appropriate to the value's actual size.
Expand the value on the top of the stack, as determined by the dynamic tag.
Remember pointer stack must be padded with **nil**, and the main stack with 0s.

subv

Op-Codes

wsubv	141	dwsubv	142
psubv	143	dpsubv	144

Description

Pop vector index from the main stack.
Pop the pointer to the vector from the pointer stack.
Compare the index with the lower and upper bounds of the vector.
if index is outwith the bounds
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v3VectorIndexSubs in the error structure ******.
The vector and vector index are passed as parameters.
else Modify the index to be the word offset to the indexed element.
Push the first word of the indexed element onto the appropriate stack.
if the instruction is dwsubv **or** dpsubv
do Push the second word of the indexed element onto the appropriate stack.

subv

Op-Code

wpsubv	145	dwdpsubv	146

Description

Pop vector index from the main stack.
Pop the pointer to the vector from the pointer stack.
Compare the index with the lower and upper bounds of the vector.
if index is outwith the bounds
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v3VectorIndexSubs in the error structure ******.
The vector and vector index are passed as parameters.
else Modify the index to be the word offset to the indexed pointer element.
Push the first word of the indexed pointer element onto the pointer stack.
if the instruction is dwdpsubv
do Push the second word of the indexed pointer element onto the pointer stack.
Modify the index to be the word offset to the indexed non pointer element.
Push the first word of the indexed non pointer element onto the pointer stack.
if the instruction is dwdpsubv
do Push the second word of the indexed non-pointer element onto the main stack.

subv.poly

Op-Code 147

Description

Pop the dynamic tag for the value being loaded from the top of the main stack.
Perform the appropriate subv instruction, as determined by the dynamic tag.
Expand the value on the top of the stack, as determined by the dynamic tag.
Remember the pointer stack must be padded with **nil**, the main stack with 0s.

subsass

Op-Codes

wsubsass	148	dwsubsass	149
psubsass	150	dpsubsass	151

Description

Pop the value to be assigned from the appropriate stack.
Pop the word offset, W, to the field being assigned from the main stack.
Pop the pointer to the structure from the pointer stack.
Copy the first word of the value to be assigned to word W of the structure.
if the instruction is dwsubs **or** dpsubs
do Copy the second word of the value to be assigned to word W + 1 of the structure.

subsass

Op-Codes

wpsubsass	152	dwdpsubsass	153

Description

Pop the value to be assigned from the stacks.
Pop the word offset, W1, to the pointer field being assigned to, from the main stack.
Pop the word offset, W2, to the non pointer field being assigned to, from the main stack.
Pop the pointer to the structure from the pointer stack.
Copy the first word of the pointer value to word W1 of the structure.
Copy the first word of the non pointer value to word W2 of the structure.
if the instruction is dwdpsubs
do Copy the second word of the pointer value to word W1 + 1 of the structure.
Copy the second word of the non pointer value to word W2 + 1 of the structure.

subsass.poly

Op-Code 154

Description

The dynamic tag for the value being assigned is on top of the main stack.
Below the dynamic tag are the two integer words of the polymorphic value.
Below the two words on top of the main stack there are two field offsets.
The top offset is the offset to use if the value to be assigned is a pointer value.
The bottom offset is the offset to use if the value to be assigned is a non pointer value.
Both offsets are used if the value to be assigned contains pointers and non-pointers.
Pop the dynamic tag from the main stack.
Contract the main stack to eliminate the unnecessary offset, as determined by the dynamic tag.
Contract the value on the top of the stack, as determined by the dynamic tag.
Perform the appropriate subsass instruction.

subvass

Op-Codes

wsubvass	155	dwsubvass	156
psubvass	157	dpsubvass	158

Description

Pop the value to be assigned from the appropriate stack.
Pop the vector index from the main stack.
Pop the pointer to the vector from the pointer stack.
Compare the index with the lower and upper bounds of the vector.
if index is outwith the bounds
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v2VectorIndexAssign in the error structure ******.
The vector, vector index and the value to be assigned are passed as parameters.
else **if** the vector's constancy bit is set
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v1ConstantVector in the error structure ******.
The vector, vector index and the value to be assigned are passed as parameters.
else Modify the index to be the word offset to the indexed element.
Copy the first word of the value being assigned to the first word of the indexed element.
if the instruction is dwsubv **or** dpsubv
do Copy the second word of the value being assigned to the second word of the indexed element.

subvass

Op-Codes

wpsubvass	159	dwdpsubvass	160

Description

Pop the value to be assigned from the appropriate stack.
Pop the vector index from the main stack.
Pop the pointer to the vector from the pointer stack.
Compare the index with the lower and upper bounds of the vector.
if index is outwith the bounds
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v2VectorIndexAssign in the error structure ******.
The vector, vector index and the value to be assigned are passed as parameters.
else **if** the vector's constancy bit is set
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v1ConstantVector in the error structure ******.
The vector, vector index and the value to be assigned are passed as parameters.
else Modify the index to be the word offset to the indexed pointer element.
Copy the pointer value being assigned to the indexed pointer element
Modify the index to be the word offset to the indexed non pointer element.
Copy the non pointer value being assigned to the indexed non pointer element.

subvass.poly

Op-Code 161

Description

Pop the dynamic tag for the value being assigned from the top of the main stack.
Contract the value on the top of the stack, as determined by the dynamic tag.
Perform the appropriate subvass instruction, as determined by the dynamic tag.

makeconst

Op-Code 162

Description

Set the constancy bit in the vector whose pointer is on top of the pointer stack.

makevar

Op-Code 163

Description

Clear the constancy bit in the vector whose pointer is on top of the pointer stack.

4.11 String Operations

These instructions create a new string object either by copying two strings or copying a contiguous selection of characters from a string.

concat.op

Op-Code 164

Description

Pop the second string from the pointer stack.
Pop the first string from the pointer stack.
if the total length of the two strings is greater than the longest possible string
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure s0Concatenate in the error structure.
The two strings are passed as parameters.
else Push a new string which is the characters of the first string immediately followed by the characters of the second string.

substr.op

Op-Code 165

Description

Pop the length of the new string from the main stack.
Pop the starting position of the new string from the main sack.
The subscripted string is on the top of the pointer stack.
Compare the new string's start and length with the length of the subscripted string.
if the new string is not a substring of the subscripted string **or** has a negative length
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure s1SubString in the error structure.
The string, start position and length are passed as parameters.
else **if** the new string is shorter than the subscripted string
do Create the new string.
Copy the new string's characters from the subscripted string, starting at the start position.

4.12 Load Literal Instructions

These are used to load the value of a literal onto the appropriate stack.

ll.int(n : byte)

Op-Code 166

Description

Push the signed integer value n onto the main stack.
The byte is an 8 bit twos complement number.

ll.char(n : byte)

Op-Code 168

Description

Lookup the vector of single character strings in the root object.
Use n as an index into the vector.
Push the indexed string element onto the pointer stack.

4.13 Primitive I/O Interface

These instructions provide a primitive interface to the host operating system's I/O facilities.

create.file

Op-Code 170

Description

Pop the file name from the pointer stack.
Pop the file's protection mask from the main stack.
Create a file in the underlying system with the specified name and protection mask.
if the file was created
do Discover the type of file that was created, a disk file, terminal file, mouse file, tablet file or raster file.
if the file is of the expected type
then Create a file descriptor for the type of file opened.
Perform any initialisation necessary to use the created file.
Push the file descriptor onto the pointer stack.
Set the I/O error number in the root object to 0.
else Push the value **nilfile** onto the pointer stack.
Set the I/O error number in the root object to indicate why the create failed.

Note that in the first UNIX implementation of the persistent abstract machine, the I/O error numbers used to indicate failure conditions correspond exactly to the UNIX error numbers.

open

Op-Code 171

Description

Pop the file name from the pointer stack.
Pop the file's access mode from the main stack, the mode can be 0 for read only, 1 for write only or 2 for read and write.
Open a file in the underlying system with the specified name and access mode.
if the file was opened
do Discover the type of file that was opened, a disk file, terminal file, mouse file,
if the file is of the expected type
then Create a file descriptor for the type of file opened.
Perform any initialisation necessary to use the opened file.
Push the file descriptor onto the pointer stack.
Set the I/O error number in the root object to 0.
else Push the value **nilfile** onto the pointer stack.
Set the I/O error number in the root object to indicate why the open failed.

close

Op-Code 172

Description

Pop the file descriptor from the pointer stack.
if the file descriptor is for a closed file
then Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate an attempt to close a closed file.
else Close the open file.
if the file was closed
then Push the integer value 0 onto the main stack.
Set the I/O error number in the root object to 0.
else Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate why the close failed.

Note that the files for standard input, output and error and the socket connection to the shell are never closed.

seek

Op-Code 173

Description

Pop the seek key from the main stack, 0 seek from the start of file, 1 seek from the current file position or 2 seek from the end of the file.
Pop the byte offset that the file position should be modified by.
Pop the file descriptor from the pointer stack.
if the file descriptor is for a closed file
then Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate an attempt to seek within a closed file.
else if the file is not a disk file
then Push the integer value -1 onto the main stack.
Set the I/O error number to indicate an attempt to seek on a non disk file.
else Set the current file position as indicated by the byte offset and seek key.
if the file position was set
then Push the integer value 0 onto the main stack.
Set the I/O error number in the root object to 0.
else Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate why the seek failed.

ioctl

Op-Code 174

Description

Pop the ioctl command number to be performed from the main stack.
Pop the vector of integers holding the command's data from the pointer stack.
Pop the file descriptor from the pointer stack.
if the file descriptor is for a closed file
then Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate ioctl was passed a closed file.
else **if** the file is not a terminal file, a disk file, a window file or a socket file
then Push the integer value -1 onto the main stack.
Set the I/O error number to indicate ioctl was not passed a terminal file.
else execute the specified ioctl command using the data vector.
if the command was successful
then Push the integer value 0 onto the main stack.
Set the I/O error number in the root object to 0.
else Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate why the requested ioctl command failed.

The ioctl commands for the UNIX implementation correspond to a subset of those supported by the UNIX ioctl system call and use the same command numbers. An implementation independent command number may also be used. The ioctl instruction will not execute the specified command unless it is applicable to the file type and the vector of integers contains sufficient integer elements to hold the parameters or results of the specified command. The supported commands (described in section 4 of the 4.2BSD manual set) and their alternatives in brackets are:

TIOCSETP (1), TIOCSETN (2), TIOCSETC (3), TIOCSLTC (4), TIOCSETD (5), TIOCFLUSH (6), TIOCSTI (7), TIOCSPGRP (8), TIOCLBIS (9), TIOCLBIC (10), TIOCEXCL (11), TIOCNXCL (12), TIOCHPCL (13), TIOCSBRK (14), TIOCCBRK (15), TIOCSDTR (16), TIOCCDTR (17), TIOCSTOP (18), TIOCSTART (19), TIOCGETP (20), TIOCGETC (21), TIOCGLTC (22), TIOCGETD (23), TIOCGPRG (24), TIOCOUTQ (25), FIONREAD (26) and FIONBIO (27).

In order to preserve the state of a terminal over a checkpoint, the terminal file descriptor records the 4 state structures used by 4.2BSD. These are the sgttyb structure, the tchars structure, the ltchars structure and a word of local flags. Whenever a specified command updates one of these state structures, the change is also recorded in the file descriptor.

One abstract machine specific ioctl command is provided to reopen a closed file. Command number 28 reopens a closed file using the name held in the file descriptor. For disk files, the open is performed from the same working directory as the original open. All other reopens are performed in the new working directory and host machine context.

read.bytes

Op-Code 175

Description

Pop the number of bytes to be read from the main stack.
Pop the byte offset into the vector of integers from the main stack.
Pop the vector of integers into which the bytes will be read from the pointer stack.
Pop the file descriptor from the pointer stack.
if the file descriptor is for a closed file
then Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate read.bytes was passed a closed file.
else **if** the file is not a disk file, a terminal file, a window file or a socket file
then Push the integer value -1 onto the main stack.
Set the I/O error number to indicate read.bytes was not passed a disk file, a terminal file, a window file or a socket file.
else Read at most the number of bytes specified into the vector of integers, starting at the specified byte offset, from the file.
if no error occurred
then Push the number of bytes read onto the main stack.
Set the I/O error number in the root object to 0.
else Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate why read.bytes failed.

Note that the bytes within each integer of the vector are temporarily converted to big-endian order during the I/O. That is, the most significant byte in an integer is the first byte read and the least significant byte is the last byte read.

write.bytes

Op-Code 176

Description

Pop the number of bytes to be written from the main stack.
Pop the byte offset into the vector of integers from the main stack.
Pop the vector of integers from which the bytes will be written from the pointer stack.
Pop the file descriptor from the pointer stack.
if the file descriptor is for a closed file
then Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate write.bytes was passed a closed file.
else if the file is not a disk file, a terminal file or a socket file
then Push the integer value -1 onto the main stack.
Set the I/O error number to indicate write.bytes was not passed a disk file or a terminal file.
else Write at most the number of bytes specified from the vector of integers, starting from the specified byte offset, to the file.
if no error occurred
then Push the number of bytes written onto the main stack.
Set the I/O error number in the root object to 0.
else Push the integer value -1 onto the main stack.
Set the I/O error number in the root object to indicate why write.bytes failed.

Note that the bytes within each integer of the vector are temporarily converted to big-endian order during the I/O. That is, the most significant byte in an integer is the first byte written and the least significant byte is the last byte written.

get.byte

Op-Code 177

Description

Pop the byte offset to the desired byte in the word to be indexed.
Pop the word to be indexed from the main stack.
if the byte offset is less than 0 or greater than 3
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a4GetByte in the error structure.
The word to be indexed and the byte offset are passed as parameters.
else Push the unsigned integer value of the indexed byte onto the main stack.

The byte index of 0 reads the most significant byte in the integer and a byte index of 3 reads the least significant byte in the integer.

set.byte

Op-Code 178

Description

Pop the integer value of the byte to be assigned to.
Pop the byte offset to the desired byte in the word to be indexed.
Pop the word to be modified from the main stack.
if the byte offset is less than 0 or greater than 3
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a5SetByte in the error structure.
The word to be modified, the byte offset and byte to be assigned are passed as parameters.
else Set the value of the indexed byte to be the least significant byte of the specified value.
Push the modified word value onto the main stack.

The byte index of 0 sets the most significant byte in the integer and a byte index of 3 sets the least significant byte in the integer.

4.14 Comparison Operations

The comparison operations act on the top two elements of the appropriate stack. They are compared and removed. The boolean result **true** or **false** is left on the main stack.

eq.ib,r,s,p,pr

Op-Codes

eq.ib	179	eq.r	180
eq.s	181	eq.p	182
eq.pr	183	eq.var	184
		eq.any	186

Description

Compare the two elements at the top of the appropriate stack.
Pop the two elements off the appropriate stack.
if the two elements are equal
then Push the boolean value **true** onto the main stack.
else **if** the two elements are not equal
then Push the boolean value **false** onto the main stack.
else **if** equality cannot be determined because one or more pairs of anys must be type checked
do Call the any comparison procedure held in the root object. The first pair anys to be type checked are passed as parameters.

Equality of the stack elements is defined as follows:

eq.ib: the elements are single words on the main stack, they must have the same integer value.

eq.r: the elements are pairs of words on the main stack, they must be compared by the floating point implementation.

eq.s: the elements are pointers to strings on the pointer stack, they must be the same pointer **or** they must have exactly the same characters.

eq.p: the elements are single words on the pointer stack, they must have the same integer value.

eq.pr: the elements are pairs of words on the pointer stack, their first words are the code vectors for the procedures being compared and their second words are the corresponding static links. The code vectors must be identical and the static links must be identical. If the alternate code vector field of a code vector is not **nil** it is used to test code vector identity. If an alternate code vector field points to its own code vector the static link field of the corresponding static link is used to test static link identity.

eq.var: the elements are a single label word on the main stack and a single pointer on the pointer stack, they must have the same labels, if the labels are equal the least significant byte of the label is the dynamic tag of the injected values, if the values to be compared are single pointers compare the two pointers otherwise the two pointers point to objects containing the values to be compared, in that case the values start immediately after the size field.

eq.any: the elements are both two integer words and two pointers, the first integer word and the first pointer of each element form a variant, the two variants are compared as described above, if the variants are not equal the anys are not equal, if the variants are equal then the types of two anys must be checked by the compiler's type checker.

eq.poly

Op-Code 185

Description

Pop the dynamic tag for the values being compared from the top of the main stack.
The values being compared are both padded to form two integer words and two pointers and are now at the top of both stacks.
Perform the comparison appropriate for the data type indicated by the dynamic tag.
Pop four words from each stack and push the result onto the main stack.
It should be noted that this instruction may be required to call the any comparison procedure.

neq.ib,r,s,p,pr

Op-Codes

neq.ib	187	neq.r	188
neq.s	189	neq.p	190
neq.pr	191		

Description

Compare the two elements at the top of the appropriate stack.
Pop the two elements off the appropriate stack.
if the two elements were equal
then Push the boolean value **false** onto the main stack.
else Push the boolean value **true** onto the main stack.

It should be noted that there are no neq.var, neq.any or neq.poly instructions. These comparisons are implemented by performing the appropriate eq.var, eq.any or eq.poly instruction followed by a not instruction.

lt.i,r,s

Op-Codes

lt.i	195	lt.r	196
lt.s	197		

Description

Compare the two elements at the top of the appropriate stack.
Pop the element, B, off the appropriate stack.
Pop the element, A, off the appropriate stack.
if the element A was less than the element B
then Push the boolean value **true** onto the main stack.
else Push the boolean value **false** onto the main stack.

Less than between two stack elements A and B is defined as follows:

lt.ib: the elements A and B are single words on the main stack, element A must have a smaller integer value than element B

lt.r: the elements A and B are pairs of words on the main stack, element A must have a smaller floating point value than element B.

lt.s: the elements A and B are pointers to strings on the pointer stack, the characters in A's string are compared with the characters at the same position in B's string until either all the characters in one string have been compared or two characters being compared differ, if all of a string's characters have been compared A's string must be shorter than B's string, if two characters differ the character from A's string must have a smaller ascii code than the character from B's string.

le.i,r,s

Op-Codes

le.i	198	le.r	199
le.s	200		

Description

Compare the two elements at the top of the appropriate stack.
Pop the element, B, off the appropriate stack.
Pop the element, A, off the appropriate stack.
if the element A was less than or equal to the element B
then Push the boolean value **true** onto the main stack.
else Push the boolean value **false** onto the main stack.

gt.i,r,s

Op-Codes

gt.i	201	gt.r	202
gt.s	203		

Description

Compare the two elements at the top of the appropriate stack.
Pop the element, B, off the appropriate stack.
Pop the element, A, off the appropriate stack.
if the element A was less than or equal to the element B
then Push the boolean value **false** onto the main stack.
else Push the boolean value **true** onto the main stack.

ge.i,r,s

Op-Codes

ge.i	204	ge.r	205
ge.s	206		

Description

Compare the two elements at the top of the appropriate stack.
Pop the element, B, off the appropriate stack.
Pop the element, A, off the appropriate stack.
if the element A was less than the element B
then Push the boolean value **false** onto the main stack.
else Push the boolean value **true** onto the main stack.

4.15 Arithmetic and Boolean Operators

These instructions operate on the data types real and integer. The top two elements of the stack are replaced by the result. The real (floating-point) operations are preceded with the letter f. Remember that each real number is two stack elements long.

plus,fplus

Op-Codes

plus	207	fplus	219

Description

Pop the value B from the top of the main stack.
Pop the value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**.
else Call the procedure a1Int or a3Real in the error structure, as appropriate.
The string "+", the value A and the value B are passed as parameters.
else Push the value of A added to B onto the main stack.

times,ftimes

Op-Codes

times	208	ftimes	220

Description

Pop the value B from the top of the main stack.
Pop the value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine.**
else Call the procedure a1Int or a3Real in the error structure, as appropriate.
The string "*", the value A and the value B are passed as parameters.
else Push the value of A times to B onto the main stack.

minus,fminus

Op-Codes

minus	209	fminus	221

Description

Pop the value B from the top of the main stack.
Pop the value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine.**
else Call the procedure a1Int or a3Real in the error structure, as appropriate.
The string "-", the value A and the value B are passed as parameters.
else Push the value of A minus B onto the main stack.

div

Op-Code 210

Description

Pop the integer value B from the top of the main stack.
Pop the integer value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a1Int in the error structure.
The string "div", the value A and the value B are passed as parameters.
else Push the quotient of A divided by B onto the main stack.

fdivide

Op-Code 222

Description

Pop the floating point value B from the top of the main stack.
Pop the floating point value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a3Real in the error structure.
The string "/", the value A and the value B are passed as parameters.
else Push the floating point value of A divided B onto the main stack.

neg,fneg

Op-Codes

neg	211	fneg	223

Description

Pop the value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a0UnaryInt or a2UnaryReal in the error structure, as appropriate.
The string "-" and the value A are passed as parameters.
else Push the negated value of A onto the main stack.

rem

Op-Code 212

Description

Pop the integer value B from the top of the main stack.
Pop the integer value A from the top of the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a1Int in the error structure.
The string "rem", the value A and the value B are passed as parameters.
else Push the remainder of A divided by B onto the main stack.

shift.r

Op-Code 213

Description

Pop the number of bits, S, to shift from the main stack
Pop the integer value to be shifted from the main stack.
if S is positive **do**
Shift the bits of integer value so that bit at position B is at position B - S, the least significant S bit positions are ignored, the most significant S bit positions are cleared.
Push the shifted integer value onto the main stack.

shift.l

Op-Code 214

Description

Pop the number of bits, S, to shift from the main stack.
Pop the integer value to be shifted from the main stack.
if S is positive **do**
Shift the bits of integer value so that bit at position B is at position B + S, the most significant S bit positions are ignored, the least significant S bit positions are cleared.
Push the shifted integer value onto the main stack.

b.and

Op-Code 215

Description

Pop the integer value, B, from the main stack.
Pop the integer value, A, from the main stack.
Construct a new integer value whose bits are set only if the corresponding bits in A and B are both set.
Push the new integer value onto the main stack.

b.or

Op-Code 216

Description

Pop the integer value, B, from the main stack.
Pop the integer value, A, from the main stack.
Construct a new integer value whose bits are set only if either of the corresponding bits in A and B are set.
Push the new integer value onto the main stack.

b.not

Op-Code 217

Description

Pop the integer value from the main stack.
Set all the bits in the integer value that are clear and clear all the bits that are set.
Push the not'd integer value onto the main stack.

not

Op-Code 218

Description

Pop the boolean value A from the top of the main stack.
if A is **true**
then Push the boolean value **false** onto the main stack.
else Push the boolean value **true** onto the main stack.

sin

Op-Code 224

Description

Pop the floating point value R from the main stack, R is an angle in radians.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "sin" and the floating point value R are passed as parameters.
else Push the value of the sine of R onto the main stack.

cos

Op-Code 225

Description

Pop the floating point value R from the main stack, R is an angle in radians.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "cos" and the floating point value R are passed as parameters.
else Push the value of the cosine of R onto the main stack.

exp

Op-Code 226

Description

Pop the floating point value R from the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "exp" and the floating point value R are passed as parameters.
else Push the value of e raised to the power of R onto the main stack.

ln

Op-Code 227

Description

Pop the floating point value R from the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "ln" and the floating point value R are passed as parameters.
else Push the value of the natural logarithm of R onto the main stack.

sqrt

Op-Code 228

Description

Pop the floating point value R from the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "sqrt" and the floating point value R are passed as parameters.
else Push the value of the square root of R onto the main stack.

atan

Op-Code 229

Description

Pop the floating point value R from the main stack, R is an angle in radians.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure a2UnaryReal in the error structure.
The string "atan" and the floating point value R are passed as parameters.
else Push the value of the arctangent of R onto the main stack.

truncate

Op-Code 230

Description

Pop the floating point value R from the main stack.
if an arithmetic error occurs
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure x0Truncate in the error structure.
The floating point value R is passed as a parameter.
else Push the value of the integer part of R onto the main stack.

float

Op-Code 231

Description

Pop the integer value I from the main stack.
Push the floating point number with the same value as I onto the main stack.

4.16 Miscellaneous

These instructions provide a date/time interface and the ability to explicitly invoke a stabilise and a persistent store garbage collection.

no.op

Op-Code 0

Description

Do nothing.

date

Op-Code 232

Description

Push a pointer to a string containing the current date and time in the form :
"Thu Apr 23 17:16:11 1987"
onto the pointer stack.

time

Op-Code 233

Description

Push the number of one sixtieth second ticks since the Napier system started onto the main stack.

stabilise

Op-Code 234

Description

Perform the checkpoint_heap operation provided by the stable store interface.

diskgc

Op-Code 235

Description

Perform the garbage_collect operation provided by the stable store interface.

4.17 Variants

These instructions manipulate the uniform representation of values as variants.

inject.op

Op-Code 236

Description

The label word for the injected value is on top of the main stack.
The value to be injected is below the label word on the appropriate stack.
Pop the label word on top of the main stack and inspect its dynamic tag.
if the dynamic tag does not represent a single pointer value
do Create an object just large enough to contain the injected value.
Pop the injected value from the appropriate stack and copy it into the object.
Push a pointer to the object onto the pointer stack.
Push the label word onto the main stack.

project.op

Op-Code 237

Description

A variant value is on top of the two stacks.
Pop the variant's label word from the main stack and inspect its dynamic tag.
if the dynamic tag does not represent a single pointer value
do Pop the pointer to the object containing the variant's value from the pointer stack.
Push the variant's value onto the appropriate stack.

index.jump(n : short, offsets : n shorts)

Op-Code 238

Description

Pop the index on top of the main stack (I).
Compare the index with the number of supplied offsets (n).
if i < 0 **or** i >= n
then Read the n - 1^{th} offset parameter (O).
else Read the i^{th} offset parameter (O).
Jump forwards O bytes from the location following the chosen offset.

4.18 Structure Constancy

These instructions test structure fields for constancy violations immediately prior to the execution of a structure assignment. These instructions must be immediately followed by the corresponding structure assign. They do not alter the stacks unless an error is detected. On an error the stacks are cleared of the parameters to the assign instruction and an error handler is called. If the handler returns, execution is resumed at the instruction following the structure assign.

subtest

Op-Codes

wsubtest	239	dwsubtest	240
psubtest	241	dpsubtest	242

Description

The stacks are set up to perform the corresponding subsass instruction.
if the next instruction is not the corresponding subsass instruction
do **Halt the abstract machine**
Read the word offset, W, to the field being assigned from the main stack.
Read the pointer to the structure from the pointer stack.
Test the constancy bit for word W in the structure.
if the instruction is dwsubs **or** dpsubs
do Test the constancy bit for word W+1 in the structure.
if either of the constancy bits is set
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v0ConstantField in the error structure.
There are no parameters.
The resume address is after the next instruction.

subtest

Op-Codes

wpsubtest 243 dwdpsubtest 244

Description

The stacks are set up to perform the corresponding subsass instruction.
if the next instruction is not the corresponding subsass instruction
do **Halt the abstract machine**
Read the word offset, W1, to the pointer field being assigned to, from the main stack.
Read the word offset, W2, to the non pointer field being assigned to, from the main stack.
Read the pointer to the structure from the pointer stack.
Test the constancy bit for word W1 in the structure.
Test the constancy bit for word W2 in the structure.
if the instruction is dwsubs **or** dpsubs
do Test the constancy bit for word W1+1 in the structure.
Test the constancy bit for word W2+1 in the structure.
if any of the constancy bits is set
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v0ConstantField in the error structure.
There are no parameters.
The resume address is after the next instruction.

subtest.poly

Op-Code 245

Description

The stacks are set up to perform the subsass.poly instruction.
if the next instruction is not subsass.poly
do **Halt the abstract machine**
Read the dynamic tag from the main stack.
Read the word offset, W1, to the pointer field being assigned to, from the main stack.
Read the word offset, W2, to the non pointer field being assigned to, from the main stack.
Read the pointer to the structure from the pointer stack.
if the value to be assigned contains pointers
then Test the constancy bit for word W1 in the structure.
if two pointers are to be assigned
Test the constancy bit for word W1+1 in the structure.
if the value to be assigned contains non pointers
then Test the constancy bit for word W2 in the structure.
if two non-pointers are to be assigned
Test the constancy bit for word W2+1 in the structure.
if any of the constancy bits is set
then **if** the error structure in the root object is **nil**
then **Halt the abstract machine**
else Call the procedure v0ConstantField in the error structure.
There are no parameters.
The resume address is after the next instruction.

4.19 Host Operating System

These instructions provide access to the host operating system environment and details of the persistent abstract machine execution.

host.environment

Op-Code 167

Description

Create a vector of strings for the command line arguments passed to the Napier system.
Create a vector of strings for the shell environment passed to the Napier system.
Create a two field structure with the command line arguments as the first field and the shell environment as the second field.
Push the pointer to the two field structure onto the pointer stack.

The precise contents of the vectors is implementation dependent. If the host operating system is not UNIX based then the contents of these vectors is undefined.

statistics

Op-Code 169

Description

A vector of vector of integers is on top of the pointer stack.
The first vector of integers should be overwritten with PAM statistics.
The second vector of integers should be overwritten with stable heap statistics.
The third vector of integers should be overwritten with stable storage statistics.
The fourth vector of integers should be overwritten with operating system statistics.

The operation of this instruction is implementation dependent. It is intended to supply performance statistics to a system implementor attempting to measure the system. Only the system implementor will be aware of the significance of all of the data written to the vectors.

The elements of the PAM statistics vector are as follows:
element 1: total user time for this execution of the Napier system in seconds.
element 2: the number of microseconds to be added to element 1.
element 3: total system time for this execution of the Napier system in seconds.
element 4: the number of microseconds to be added to element 3.
element 5: time of day expressed in seconds since 00:00, Jan 1, 1970.
element 6: the number of microseconds to be added to element 5.
if the vector has more than 6 elements the extra elements are implementation dependent,
if the vector has less than 6 elements only the ones supplied are used.

4.20 Thread Operations

Concurrency is supported by lightweight threads. Each thread has some private state information including a pointer to the stack frame for the procedure it is executing, the status of its last I/O operation and a lock word. Access to the I/O status is made through root instructions which automatically access the thread context object rather than the abstract machine's root object. Access to the lock word is made via the modlock instruction, a positive value in the lock word indicates a Napier level logical operation is in progress. Executing threads are only subject to context switching between abstract machine instructions, at the end of Napier level logical operations or by executing an appropriate thread operation. Napier level logical operations are executed as atomic operations as are abstract machine instructions that operate on shared data.

thread.op

Op-Code 248

Description

Pop a procedure closure from the pointer stack.
Pop a pointer to a vector from the pointer stack.
Perform the specified thread operation.

The first element of the vector contains the thread operation code.
The second element of the vector contains the thread identifier.
If there is only one element in the vector, thread operations requiring a thread identifier are treated as no-ops.

The thread operation code has the following interpretation:

0 Start Thread:
Create a new thread to execute the specified procedure.
Assign the new thread's identifier to the first element of the vector.

1 Get Thread Identifier:
Assign the executing thread's identifier to the first element of the vector.

2 Get All Threads:
Assign the current number of threads to the first element of the vector.
Assign the identifiers of the existing threads to successive elements of the vector. If the vector has too few elements only the ones supplied are filled in.

3 Kill Thread:
Mark the specified thread as killed.
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.
if the executing thread is marked as killed
do Clear the thread lock word.
Perform a context switch.

4 Restart Thread:
Mark the specified thread as runnable.
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.

5 Suspend Thread:
Mark the specified thread as suspended.
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.
if the executing thread is marked as suspended
do Clear the thread lock word.
Perform a context switch.

6 Kill All Threads:
Mark all threads in the system as killed.
Assign 1 to the first element of the vector.
Clear the thread lock word.
Perform a context switch.

7 Suspend and Unlock Thread:
Mark the specified thread as suspended.
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.
Clear the thread lock word.
if the executing thread is marked as suspended
do Perform a context switch.

8 Restart and Unlock Thread:
Mark the specified thread as runnable.
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.
Clear the thread lock word.

9 Live Thread?:
if the specified thread does not exist or has already been killed
then Assign 0 to the first element of the vector.
else Assign 1 to the first element of the vector.

modlock

Op-Code 249

Description

Pop an integer value from the main stack.
Increment the thread lock word with the popped integer.
if the result of the increment is negative
do Set the lock word to be 0.
Push the final value of the lock word onto the main stack.

5 Persistence

5.1 The Interface to the Persistent Store

A program module within the abstract machine's implementation provides a main memory heap. The heap is in fact an optional layer of the persistent store that is used to cache objects held in the persistent heap. The main memory heap, hereafter referred to as the local heap, is accessed via 12 interface functions that will now be described. A full description of the layers within the stable store can be found in [14]. It is assumed that the local heap is implemented in a byte addressed RAM.

5.2 Interface Functions to the Local Heap

Initialise_heap:

This function will cause the local to be initialised and the persistent heap to be opened. As a part of the open, the disk store to be used will be locked to prevent interference. The function is called each time the persistent abstract machine is invoked.

Shutdown_heap:

This function causes the local and persistent heaps to be shutdown and so release any system resources they may be using. This is the converse of initialise_heap.

Create_object:

This function is used to create a new object in the local heap. It is parameterised by the size of the new object in words. When the object is created it is initialised as an object of the required size but with no pointer fields. The act of inserting pointers into the object and setting the count of the number of pointers is left to the abstract machine. Once the object has been created the function will return its address. The address will be a byte address in RAM. If for some reason the object cannot be created the address returned will be 0.

Destroy_object:

This function is used to release the store allocated to an object. It is parameterised by the RAM byte address of the object. The function will not release the storage if the particular organisation of the local heap does not support this operation.

Illegal_address:

This function is used to move objects from the persistent heap into the local heap. It is parameterised by the address of the object in the persistent heap and returns the byte address it was copied to in RAM. When illegal address is called the persistent heap address is looked up in a mapping table. The mapping table records all persistent heap objects that currently have a copy in RAM. If the persistent heap address is present then the corresponding RAM address is returned. However if the persistent heap address is not present the object being addressed must first be copied into RAM and then its RAM address returned. An object is copied into RAM in four steps:

a. create_object is called to create an object in the local heap the same size as the addressed object.
b. the addressed object is then copied from the persistent heap overwriting the newly created heap object.
c. all the pointers held within the copied object are negated since they are persistent heap addresses and must be distinguished from RAM addresses.

d . finally the address of the newly created object is entered in the mapping table together with the persistent heap address.

If at step **a**, an object cannot be created in the local heap, the result of illegal address will return the address 0. Until a successful garbage collection of the local heap has been performed the illegal address may not be translated.

All persistent heap addresses are distinguished from RAM addresses by being negative values and hence invalid RAM addresses. Therefore it is necessary to translate these addresses before using them. To minimise the number of translations the following rules should be applied:

a . Once a persistent heap address has been translated it is overwritten by its RAM address. Note this overwriting is most effective if it is done at the source of the address, that is in the field of a structure rather than on the pointer stack.

b . All the pointers in the following objects must be RAM addresses:
the current frame,
the root object,
the vector of single character strings,
the vector of event handling procedures,
the structure of error handling procedures,
the vector of open files and
the open files.

c . Whenever a pointer value is found in an object other than those listed in **b** it must be checked and if necessary translated into a RAM address.

d . Whenever control is transferred, the new current frame must have all its pointers checked and if necessary translated into RAM addresses.

e . When the local heap is initialised its root object must be copied from the persistent heap and the requirements of rule **b** enforced.

To support these rules every object is tagged, with a single bit,when it is first copied into the heap and subsequently when a RAM address it contains is overwritten by the corresponding persistent heap address. The tag bit is reset if an object has become the current frame and had all its pointers translated into RAM addresses.

These rules are sufficient to guarantee no persistent heap addresses are encountered by instructions that operate solely on the current frame. Hence they need never deal with persistent heap addresses. In addition, a current frame can be tested to see if it conforms to the above rules by simply testing a tag bit. Thus a procedure return need only test one bit when transferring control to an existing frame object.

First_object:

This function returns the RAM address of the first object in the local heap. The first object in the local heap is the root object of the persistent heap.

Checkpoint_heap:

This function causes all the new or changed objects within the local heap to be copied to the persistent heap. The persistent heap then performs its own checkpoint operation so that the persistent heap changes to a new stable state.

The act of copying any local heap object to the persistent heap is done in two steps. First any RAM addresses it contains must be translated into persistent heap addresses. A RAM address is given a persistent heap address by creating a new object of the same size in the persistent heap and then adding a mapping between the RAM address and the new persistent object to the mapping table. The object is then copied to the persistent heap translating any RAM addresses as it is copied.

Garbage_collect:

This function performs a garbage collection on the local heap destroying any unreachable objects that were not copied from the persistent heap.

Garbage_collect_the_persistent_heap:

This function performs a checkpoint operation followed by a garbage collection of the persistent heap. On completion of the garbage collection the local heap must be reinitialised since the addresses of the persistent heap objects may have changed The checkpoint is necessary to ensure that the persistent heap is in a self consistent state prior to the garbage collection and to ensure that no information is lost when the local heap is reinitialised.

Can_create:

This function indicates whether or not an object of a given size can be created by the persistent heap and that the creation will not cause a checkpoint of the persistent heap to fail. The function must always be called before attempting to create a new object. If the object cannot be created, the persistent abstract machine must initiate a checkpoint_heap operation, possibly followed by a garbage_collect_the_persistent_heap operation, and restart the current instruction. **NB**: this requires any abstract machine instruction that may create an object to be restartable.

Can_modify:

This function indicates whether a particular object, identified by its key, can be modified without causing a checkpoint of the persistent heap to fail. The function must always be called before modifying an object. If the object cannot be modified, the persistent abstract machine must initiate a checkpoint_heap operation and restart the current instruction. **NB**: this requires any abstract machine instruction that may modify an object to be restartable.

Uncreate:

This function indicates to the persistent heap that an object created in the local heap, following a successful call to can_create, has been deleted. The deleted object is then disregarded for the purposes of calculating the result of calls to can_create and can_modify.

5.3 Implementation Consequences

As a direct consequence of the can_modify, can_create and uncreate interface functions, an implementation of the Persistent Abstract Machine must support restartable instructions. In addition, it is also necessary to mark those housekeeping objects that may be changed, such as the current frame, open files vector and root object, as modified before executing any PAM code. In this way, any operation involving an update to a housekeeping object is guaranteed to succeed.

6 Errors and Events

The persistent abstract machine supports the automatic execution of user defined procedures when an error condition or asynchronous event occurs. The procedures associated with error conditions are held in a structure whereas the procedures associated with asynchronous events are held in a vector. Both data structures are pointed to by the root object.

6.1 Errors

The structure of error procedures has the following type:

```
type errorStructure is structure
(
        ! arithmetic errors
        a0UnaryInt          : proc( string,int -> int ) ;
        a1Int               : proc( string,int,int -> int ) ;
        a2UnaryReal         : proc( string,real -> real ) ;
        a3Real              : proc( string,real,real -> real ) ;
        a4GetByte           : proc( int,int -> int ) ;
        a5SetByte           : proc( int,int,int -> int ) ;
        x0Truncate          : proc( real -> int ) ;

        ! graphics errors
        g00GetPixel         : proc( image,int,int -> pixel ) ;
        g01SetPixel         : proc( image,int,int,pixel ) ;
        g02PixelOverflow    : proc( pixel -> pixel ) ;
        g03SubPixel         : proc( pixel,int,int -> pixel ) ;
        g04MakeImage  : proc( int,int,pixel -> pixel ) ;
        g05SubImage         : proc( image,int,int -> image ) ;
        g06LimitAt          : proc( image,int,int -> image ) ;
        g07LimitAtBy        : proc( image,int,int,int,int -> image ) ;
        g08ConstantImage    : proc( image ) ;
        g09GetScreen        : proc( file -> image ) ;
        g10Locator          : proc( file,*int ) ;
        g11ColourMap        : proc( file,pixel,int ) ;
        g12ColourOf         : proc( file,pixel -> int ) ;
        g13GetCursor        : proc( file -> image ) ;
        g14SetCursor        : proc( file,image ) ;
        g15GetCursorInfo    : proc( file,*int ) ;
        g16SetCursorInfo    : proc( file,*int ) ;
        x1ClosedWindow      : proc( file,image ) ;

        ! string errors
        s0Concatenate       : proc( string,string -> string ) ;
        s1SubString         : proc( string,int,int -> string ) ;

        ! structures and vector errors
        v0ConstantField     : proc()
        v1ConstantVector    : proc[ t ]( *t,int,t ) ;
        v2VectorIndexAssign : proc[ t ]( *t,int,t ) ;
        v3VectorIndexSubs   : proc[ t ]( *t,int -> t ) ;
        v4MakeVector        : proc[ t ]( int,int,t -> *t ) ;

        ! variant errors
        v5VarProject        : proc( typeRep,int,int )
)
```

It should be noted that the two character prefix to each procedure's name is to ensure that the procedures are stored in the correct order (the compiler assumes alphabetic order), a0UnaryInt first to x1ClosedWindow last. All of the above error procedures with the exception of v5VarProject are called when an abstract machine instruction detects an error condition. In each case they can perform some desired action, programmed by the user, and then return allowing the running program to continue. Alternatively, the procedures can display a suitable error message and halt the running program or invoke an exception processing mechanism.

The procedure v5VarProject is called by the code planted by the code generator to check that a variant projection is valid. The planted code forces the running program to halt if the error procedure returns because the program is unable to continue after the failed projection.

When a persistent store is first used it does not contain a structure of error procedures. The root object has **nil** in the field that points to the structure. Until an error structure is placed in the persistent store any errors cause the abstract machine to halt, giving a suitable error message.

6.2 Events

The abstract machine recognises a small number of asynchronous events. The procedures associated with each of these events are held in a vector pointed to by the root object. The type of the vector is ***proc**(). When an event occurs the persistent abstract machine sets two flags, one to indicate that an event has occurred and one to indicate which event. The current instruction is then resumed. Between executions of abstract machine instructions the flag indicating an event has occurred is checked. If the flag is set, the flags associated with each recognised event are also checked. For each event that has occurred a procedure call is set up. This is performed in the following steps:

- **a.** The pointer to the vector of event procedures is looked up in the root object. When a persistent store is first used the pointer is set to **nil**.
- **b.** If the pointer is **nil** the event is ignored.
- **c.** If the pointer is not **nil** the procedure entry for the event is looked up in the vector.
- **d.** If the procedure entry is beyond the end of the vector the event is ignored.
- **e.** A call of the event procedure is then made: a frame is created for the procedure, the registers LFB, LPSP and LMSP are set to point to the new frame and the register CP is set to point to the procedure's first instruction.

The net effect of the above steps is to stack up a procedure call for each event that has occurred. On completion of a particular event procedure, the procedure will return to the previously set up call. If there are no previously set up calls the procedure will return to the running program which then continues as normal.

In certain cases it is desirable to delay handling an asynchronous event. For example, if an environment is being updated then no other code should be run which could interfere with the update. To ensure this cannot happen the modlock instruction can be used to indicate that a Napier level logical operation is being executed. The abstract machine will not process an asynchronous event during the execution of a Napier level logical operation.

The events recognised by the first UNIX implementation of the persistent abstract machine are:

event 1:	UNIX hangup signal,
event 2:	UNIX interrupt signal (^C),
event 3:	UNIX quit signal (^\ )
event 4:	Interval timer (30Hz) and
event 5:	Poll for network connection request (30Hz).

The format of the vector of event procedures is:

word 0,1	object header and size,
word 2,3	procedure closure for the UNIX hangup signal,
word 4,5	procedure closure for the UNIX interrupt signal,
word 6,7	procedure closure for the UNIX quit signal,
word 8,9	procedure closure for the 30Hz interval timer,
word 10,11	procedure closure for the network connection poll,
word 12	lower bound for the vector,
word 13	upper bound for the vector.

If the vector contains more than five procedures the additional procedures are never used. Similarly, if the vector contains less than five procedures no attempt will be made to call the missing procedures.

7 References

1. A. Albano, L. Cardelli and R. Orsini
Galileo : a strongly typed interactive conceptual language.
ACM Transactions on Database Systems 10(2), 230-260 (1985).

2. PS-algol Reference Manual, 4th Edition.
Universities of Glasgow and St Andrews PPRR-12-87, 1987

3. R. Morrison, A. Brown, R. Carrick, R. Connor & A. Dearle
The Napier Language Reference Manual
University of St Andrews, 1988

4. A. Dearle
Constructing Compilers in a Persistent Environment
2nd International Workshop on Persistent Object Stores, Appin, August 1987

5. A. Dearle
A Persistent Architecture Intermediate Language
Universities of Glasgow & St Andrews PPRR-35-87, 1987

6. A. Brown
A Distributed Stable Store
2nd International Workshop on Persistent Object Stores, Appin, August 1987

7. PS-algol Abstract Machine Manual
Universities of Glasgow & St Andrews PPRR-11-85, 1985

8. P. Bailey, P. Maritz & R. Morrison
The S-algol Abstract Machine
University of ST Andrews CS-80-2, 1980

9. A. Goldberg & D. Robson
Smalltalk-80. The Language and its Implementation
Addison-Wesley, 1983

10. M. Atkinson & R. Morrison
Procedures as Persistent Data Objects
ACM TOPLAS 7(4) October 1985 539-559

11. D. McNally, A. Davie & A. Dearle
A Scheme for Compiling Lazy Functional Languages
University of St Andrews, Staple/StA/88/4, 1988

12. L. Cardelli
Compiling a Functional Language
Proc. 1984 LISP and Functional Programming Conference
Austin, Texas August 1984

13. A Proposed Standard for Binary Floating Point Arithmetic, Draft 8.0 of IEEE Task P754
IEEE Computer, March 1981, pp51-62.

14. A.L. Brown
Persistent Object Stores (Ph.D. Thesis)
University of St Andrews, 1989.

Appendix I: Persistent Abstract Machine Operation Codes

Jumps

fjump(short)	1	jumpf(short)	2
bjump(short)	3	bjumpt(short)	4
jumpff(short)	5	jumptt(short)	6
fortest(short,short,short)	7	forstep(short)	8
fortestS(short)	9	forstepS(short)	10
cjump.ib(short)	11	cjump.r(short)	12
cjump.s(short)	13	cjump.p(short)	14
cjump.pr(short)	15	cjump.var(short,short)	16
cjump.poly(short,short)	17	cjump.any(short,short)	18

Stack Load and Assignment

wroot(short)	19	dwroot(short)	20
proot(short)	21	dproot(short)	22
wproot(short,short)	23	dwdproot(short,short)	24
wroot.ass(short)	25	dwroot.ass(short)	26
proot.ass(short)	27	dproot.ass(short)	28
wproot.ass(short,short)	29	dwdproot.ass(short,short)	30
wlocal(short)	31	dwlocal(short)	32
plocal(short)	33	dplocal(short)	34
wplocal(short,short)	35	dwdplocal(short,short)	36
wlocal.ass(short)	37	dwlocal.ass(short)	38
plocal.ass(short)	39	dplocal.ass(short)	40
wplocal.ass(short,short)	41	dwdplocal.ass(short,short)	42
wload(short,short)	43	dwload(short,short)	44
pload(short,short)	45	dpload(short,short)	46
wpload(short,short,short)	47	dwdpload(short,short,short)	48
wassign(short,short)	49	dwassign(short,short)	50
passign(short,short)	51	dpassign(short,short)	52
wpassign(short,short,short)	53	dwdpassign(short,short,short)	54
contract.poly(short,short)	55	expand.poly(short,short)	56

Stack Duplicate

wdup	57	dwdup	58
pdup	59	dpdup	60
wpdup	61	dwdpdup	62

Stack Retract

Block Entry and Exit

Procedure Entry and Exit

Image Operations

Vector and Structure Creation Instructions

Vector and Structure Accessing Instructions

Constancy Instructions

String Operations

Load Literal Instructions

Host Operating System

Primitive I/O Instructions

Comparison Operations

Arithmetic and Boolean Operators

Miscellaneous

Variants

Thread Implementation

Appendix II: Code File Format

PAM Code files consist entirely of valid PAM objects except for the file header. This contains the following pieces of information necessary to bootstrap a PAM system.

1. PAM Magic Number
2. Size of the File (bytes)
3. Number of Objects in the File
4. Address of the Root Object
5. Compiler Magic Number

The size of the file is relative to the end of the header information.

The header information is followed by PAM objects each of which are prefixed by a single word containing 0. This word is used during execution by the heap manager.

All addresses in code files are byte offsets from the end of the header information.

The PAM magic number in hexadecimal is 0xFC51000A, the least significant 16 bits of which are the PAM version number, 10.

The PAM and Compiler magic numbers are the same as in the root object. They are used to compare the versions of PAM code in the stable store and in the code file. The two sets of PAM code must have the same compiler and PAM magic numbers.