# ProcessBase Abstract Machine Manual

**Version 2.0.6**

**August 1999**

Ron Morrison[†]
Dharini Balasubramaniam[†]
Mark Greenwood[¥]
Graham Kirby[†]
Ken Mayes[¥]
Dave Munro[*]
Brian Warboys[¥]

[†]School of Mathematical and Computational Sciences,
University of St Andrews

[¥]Department of Computer Science,
University of Manchester

[*]Department of Computer Science,
University of Adelaide

## Contents

# 1 Introduction

ProcessBase is the simplest of a family of languages and support systems designed for process modelling. It consists of the language and its persistent environment. The persistent store is populated and, indeed, the system uses objects within the persistent store to support itself. The implication of orthogonal persistence is that the user need never write code to move or convert data for long or short term storage [ABC+83]. The model of persistence in ProcessBase is that of reachability from a root object. The persistent store is stable, that is, it is transformed atomically from one consistent state to the next. Execution against the persistent store is always restarted from the last stable state.

The ProcessBase language is in the algol tradition as were its predecessors S-algol [Mor79], PS-algol [PS88] and Napier88 [MBC+96]. Following the work of Strachey [Str67] and Tennent [Ten77] the languages obey the principles of correspondence, abstraction and type completeness. This makes for languages with few defining rules allowing no exceptions. It is the belief of the designers that such an approach to language design yields more powerful and less complex languages.

The ProcessBase type system philosophy is that types are sets of values from the value space. The type system is mostly statically checkable, a property we wish to retain wherever possible. However, dynamic projection out of unions for type *any*, allows the dynamic binding required for orthogonal persistence [ABC+83] and system evolution [MCC+93].

The type system contains the base types integer, real, boolean and string. Higher-order procedures allow code to exist in the value space. Aggregates may be formed using the vector and view types. Both of these allow information hiding without encapsulation. Finally there is an explicit constructor to provide locations.

The type equivalence rule in ProcessBase is by structure and both aliasing and recursive types are allowed in the type algebra.

ProcessBase programs are executed in a strict left to right, top to bottom manner except where the flow of control is altered by one of the language clauses.

The ProcessBase persistent programming system was originally planned as part of the Compliant Systems Architecture Project. It is supported by the EPSRC under grant GR/L32699 at the University of St Andrews and GR/L34433 at the University of Manchester.

The ProcessBase programming system provides the following facilities:

- Orthogonal persistence
  - models of data independent of longevity
- Type completeness
  - no restrictions on constructing types
- Higher-order procedures
  - procedures are data objects
- Information hiding without encapsulation
  - views of data that hide detail
- A strongly typed stable store
  - a populated environment of typed data objects that may be updated atomically
- Hyper-code
  - one representation of a value throughout its lifetime [KCC+92]

- Linguistic reflection
  - to allow reflective programming [Kir92]
- Exceptions
  - for recovering from exceptional conditions
- Interrupts and down-calls
  - for communication between the ProcessBase language and the implementation level

The ProcessBase language consists of a core part and an extensible platform-dependent part. The core part of the language must be provided by all implementations whereas the platform-dependent part may be implemented in different ways depending on the host platform. At present I/O, threads, semaphores, persistence, and string and arithmetic functions are considered as platform-dependent. A separate manual, the ProcessBase Standard Library Reference Manual, describes the platform-dependent part of the language. The library mechanism may also be used to extend the language and its implementation to support a particular application.

Three sets of rules are used to define ProcessBase. The context free syntax of the language is captured by using extended BNF. This context free syntax is then constrained by type rules into only allowing context sensitive constructions. The BNF and types are used in the ProcessBase Language Reference Manual. The meaning of every legal ProcessBase language construct is defined in terms of a set of code generation rules that describe the effect of the construct as a sequence of instructions to an abstract machine. The code generation rules are given in this manual.

As mentioned above, ProcessBase is the first in a family of languages. The reflective compiler is defined in terms of ProcessBase and implemented in it. Using that the hyper-code system will be added. The conceptual approach is that any language in the compliant architecture will be implemented by reflecting into ProcessBase itself. Thus a process modelling language or a language allowing polymorphic definition of code may be added as higher layers of the compliant architecture.

This manual describes the execution engine for ProcessBase [MBG+99]. The ProcessBase Abstract Machine (PBAM) is derived from extensive experience in constructing abstract machines for block-retention, persistent languages such as PS-algol [PS85] and Napier88 [CBC+90]. The storage architecture is a cross between the S-algol abstract machine [BMM80] and Napier88 PAM [BCC+88]. The heap layout and dynamic tags are from PAM as are many of the instructions. The two stack architecture derives from S-algol.

For block retention PBAM uses an environment mechanism first described by Davie [DM88]. It uses boxing of variables, in individual heap objects, so that locations may be shared.

## 1.1 The PBAM Storage Architecture

The PBAM uses a heap-based storage architecture to share data, together with two contiguous stacks per thread to facilitate execution. The heap contains data and code, and the stacks contain the values necessary for the execution of the dynamic call chain.

The heap consists of values in a uniform object format. A garbage collector, which traces objects starting with the pointers in the root object, is used to reclaim unused space. The uniform object format allows the garbage collector and persistent object store to operate without knowing the details of the PBAM.

There are two stacks per thread: one which contains the non-pointer values (main stack) and one which contains the pointer values (pointer stack). This arrangement facilitates the identification of root pointers for the marking phase of garbage collection.

A stack frame, for local storage, is placed on each stack on procedure call and removed on procedure exit. The stacks are contiguous and reuse the frame space without recourse to the garbage collector. However, contiguous stacks cannot be used to execute block retention languages, such as ProcessBase, without making special arrangement for values whose extent exceeds their scope.

The PBAM stacks do not have a display mechanism for keeping track of the static environment. Instead each closure has an environment object which contains all the free variables for the procedure, local values being contained in the stacks. The closure is calculated at the point of the procedure literal. The *form closure* instruction constructs the environment from an environment template. The template indicates where the environment values may be found (either in the enclosing environment or the current stack frame). This may be calculated statically by the compiler and executed dynamically. Some consequences of this addressing technique are:

- Addressing within a procedure is either to the current stack frame or the environment.
- Since the environment is copied, the shared *loc* types must be boxed and the pointers copied in the environment.
- Since value retention is through the environments, stack frames are only required for procedure, and not block, entry.
- A contiguous stack may be used since the static environment is not being maintained by the stacks.
- There is no frame space allocation on procedure entry. A procedure closure (CV, ENVIRONMENT) is calculated once at the point of expression. The stack is then used for local values. Where a procedure is called many times this should reduce the work of the heap management system.

## 1.2 Concurrency

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 *suspend*, *resume* and *kill* can be performed on any thread provided its thread identifier is known. A thread can be in any of the states *runnable*, *suspended* or *killed*. Threads are not part of the core language and are therefore implementation dependent; they are described in the ProcessBase Standard Library Reference Manual.

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.

Semaphores are provided as a primitive for user-level concurrency control. They are not part of the core language and are therefore implementation dependent; they are described in the ProcessBase Standard Library Reference Manual.

## 2 Abstract Machine Registers

The registers of the PBAM are:

| | |
|---|---|
| FB | main stack local frame base |
| PFB | pointer stack local frame base |
| SP | main stack top |
| PSP | pointer stack top |
| CP | code pointer |
| E | environment pointer |
| ROP | abstract machine root object pointer |
| EHSP | exception handler stack pointer |
| IHSP | interrupt handler stack pointer |

## 2.1 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 | total size in words of the object |
| word 2..n | the pointer fields |
| word n+1..m | the non pointer fields |
| word m+1 | reserved for hash code |

### 2.1.1 The Header

The header word 0 has the following interpretation:

| | |
|---|---|
| bits 0-23 | the number of pointer fields in the object |
| bit 24-31 | reserved for implementation experiments |

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

### 2.1.2 Hash Codes

The last word in every object is reserved for storing a hash code, to support efficient sorting of objects. The word is initialised to 0 on object creation, and is filled in by the *hashCode* instruction.

### 2.1.3 The Pointer Fields

The pointer fields within an object are each a single word in length and point to objects in the heap. Since the heap uses object level addressing, all pointers must address the start of an object. Thus, 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.

## 2.2 FB and PFB

A stack frame, for local storage, is placed on each stack on procedure call and removed on procedure exit. The FB register is used to point to the main stack local frame base and PFB to the pointer stack local frame base for the currently executing procedure (the local frame).

Both are updated on procedure call and return. Local data is accessed by indexing from either FB or PFB.

## 2.3 SP and PSP

The SP register points to the top of the main stack and the PSP register points to the top of the pointer stack. In fact, the SP and PSP registers point to the word following the last word on the appropriate stack. The values of SP and PSP are never stored in the heap.

## 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 SP and PSP 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 (by call or return) or moved (by garbage collection).

## 2.5 E

The E register points to the environment object for the current procedure. The environment object contains all the free values necessary for the execution of the procedure. The E register is updated on procedure call and return.

## 2.6 ROP

A garbage collection of the persistent heap retains all objects that are reachable, by following object addresses (pointers), from the root object. The PBAM 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 (stacks), and a pointer field that is used as the root of persistence for user data.

### 2.6.1 PBAM Root Object

| | |
|---|---|
| word 0,1 | object header and size |
| word 2 | the view literal **nil** |
| word 3,4 | the closure for the start-up procedure |
| word 5 | the logical root of persistence |
| word 6,7 | the closure for a procedure to check type equivalence of two **any**s |
| word 8 | pointer to an object used by library-dependent instructions |
| word 9 | the PBAM magic number |
| word 10 | the compiler magic number |
| word 11 | unused (reserved for hash code but never used since root object cannot be manipulated directly by user programs) |

## 2.7 EHSP

The EHSP register, exception handler stack pointer, points to the current exception handler control block on the scalar stack.

## 2.8 IHMSP

The IHSP register, interrupt handler stack pointer, points to the current interrupt handler control block on the scalar stack.

# 3 Data Types

The PBAM supports a range of data types that may be classified as scalar data types and pointer data types. The scalar data types, represented by integer words, are: *integer*, *boolean* and *real*. The pointer data types, represented by the addresses of data objects (pointers), are: *loc*, *string, vector, view*, *procedure* 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, described in Section 4, are permitted on an *integer*:

- *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.*

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

### 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, described in Section 4, are permitted on a *boolean*:

- *equals, not equals, not,*
- *and, or*.

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

### 3.1.3 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. 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, described in Section 4, are permitted on a *real*:

- *equals, not equals, less than, less than or equals, greater than, greater than or equals,*
- *negate, plus, minus, multiply, divide.*

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 exception. 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 Location

The *location* 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..n | the value |
| word n+1 | reserved for hash code |

The following operations, described in Section 4, are permitted on a *location*.

- *equals, not equals,*
- *creation,*
- *dereference* (access the value),
- *assignment.*

### 3.2.2 String

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..n | the characters 1 per byte (4 per word); the last word is padded with zeros up to a 4 byte boundary |
| word n+1 | reserved for hash code |

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

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

An attempt to select a non-existent section of a *string*, or a section of negative length is treated as an exception.

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.

### 3.2.3 Vector

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 a heap 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 |

| | |
|---|---|
| word n+3 | reserved for hash code |

The following operations, described in Section 4, are permitted on a *vector*:

- *equals*, *not equals*,
- *creation*, *access the lower bound*, *access the upper bound*,
- *read an element*.

An attempt to use an index outwith the *vector* bounds or to create a *vector* with an upper bound less than the lower bound, are treated as exceptions.

Two values of type *vector* are equal if they are pointers to the same *vector*.

### 3.2.4 View

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

| | |
|---|---|
| word 0,1 | object header and size |
| word 2..n | the pointer fields |
| word n+1..m | the non-pointer fields |
| word m+1 | reserved for hash code |

The following operations, described in Section 4, are permitted on a *view*:

- *equals*, *not equals*,
- *creation*
- *read a field*.

Two values of type *view* are equal if they are pointers to the same *view*.

### 3.2.5 Procedure

*Procedure* is the only pointer data type that is represented by two 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 object. Together the two pointers form the closure of the *procedure*. A closure is formed when a *procedure* literal is executed. The closure is always addressed by addressing the first pointer.

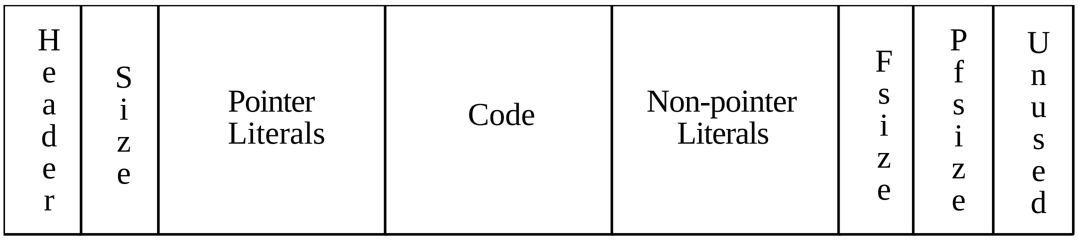
The following operations, described in Section 4, are permitted on a *procedure*:

- *equals*, *not equals*,
- *creation*,
- *apply*.

Two *procedure* values are equal if the code vectors and the environments are the same pointer values respectively.

#### 3.2.5.1 Code Vector

A code vector contains the executable code for a procedure, any scalar or pointer literals used by the procedure and the sizes of the stack frames required when the procedure is executing. The format of a code vector is as follows:



| | |
|---|---|
| word 0,1 | object header and size |
| word 2..m | pointers to objects used by the code vector's procedure |
| word m+1..n | the code to be executed |
| word n+1..p | non-pointer literals used by the code vector's procedure |
| word p+1 | the size of the main frame (in words) required when the code vector's procedure is applied (*Fsize*) |
| word p+2 | the size of the pointer frame (in words) required when the code vector's procedure is applied (*Pfsize*) |
| word p+3 | unused (reserved for hash code but never used since hash codes for procedures are associated with environments) |

### 3.2.5.2 Environment

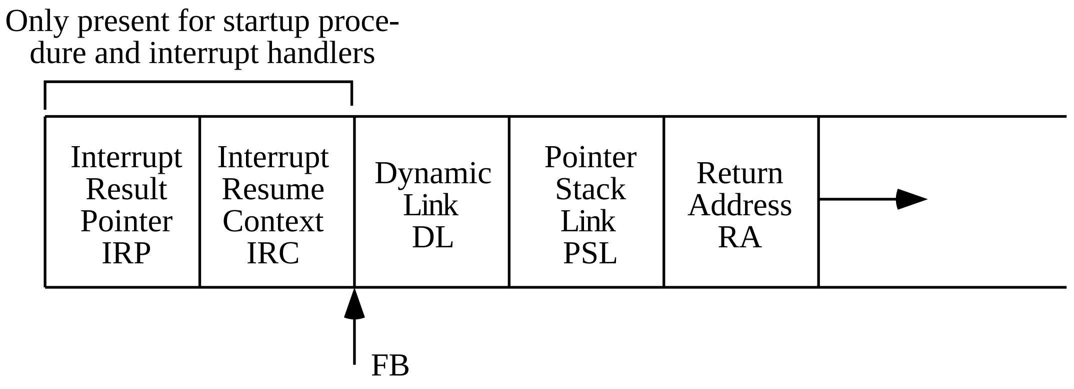
The environment for a procedure contains the free values necessary for the execution of the procedure. An environment value is represented by a pointer to a heap object with the following format:

| | |
|---|---|
| word 0,1 | object header and size |
| word 2..n | free pointer values |
| word n+1..m | free scalar values |
| word m+1 | reserved for hash code |

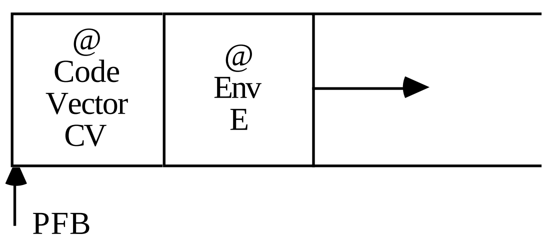
### 3.2.5.3 Frame

The stack frames contain at their bases the mark stack control word (MSCW), which is the housekeeping information necessary to ensure correct procedure entry and exit. The first few elements of the frames that make up the MSCW are laid out as follows:

**Main Stack**


Only present for startup procedure and interrupt handlers
Interrupt Result Pointer IRP
Interrupt Resume Context IRC
Dynamic Link DL
Pointer Stack Link PSL
Return Address RA
FB

**Pointer Stack**



| | |
|---|---|
| IRP | The interrupt result pointer points to the C **struct** that will contain the interrupt handler result |
| IRC | The interrupt result context points to the saved C context for jumping to when the interrupt handler returns |
| DL | The dynamic link is the offset from FB to the main stack local frame base of the calling procedure |
| PSL | The pointer stack link is the offset from PFB to the pointer stack local frame base for the corresponding stack frame on the pointer stack |
| RA | The return address for the frame’s procedure (RA), the saved offset (in bytes) from the start of the procedure’s code vector |
| CV | A pointer to the code vector for the frame’s procedure |
| E | A pointer to the environment for the frame’s procedure |

The respective directions in which the two stacks are laid out in memory are not defined.

### 3.2.6 Any

A value of type *any* is represented by a pointer to a heap object with the following format:

| | |
|---|---|
| word 0,1 | object header and size |
| word 2 | pointer to the type of the injected value |
| word 3..n | the injected value |
| word n+1 | the dynamic tag |
| word n+2 | reserved for hash code |

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

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

Two *any* values are equal if the injected values are equal, as determined by the dynamic tag, 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.

#### 3.2.6.1 Dynamic Tags for Any

The dynamic tag is used to differentiate each of the data types that are supported by the PBAM. 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 dynamic tag. 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 PBAM objects:

| **object** | **bit pattern** | **integer code** |
|---|---|---|
| integer | 00000001 | 1 |
| boolean | 00000101 | 5 |
| real | 00000010 | 2 |
| string | 01000100 | 68 |
| vector, view, loc | 01000000 | 64 |
| procedure | 10000000 | 128 |

# 4 Exceptions, Interrupts and Down-calls

Exception and interrupt handlers are accumulated and discarded dynamically as the program executes [MBG+99] and so may be recorded on the program stack. A single procedure may contain multiple nested handlers and conversely a sequence of nested procedure calls may contain no handlers. Thus the threading of the handlers on the stack is not in direct mapping to the procedure calling sequence but superimposed upon it. For this reason there are two registers, EHSP and IHSP, which record the positions on the stack of the current exception handler and interrupt handler respectively (dynamically). All other handlers may be found via chains of pointers within the main stack.

When execution enters a clause with a handler defined, the new handler comes into dynamic scope and a corresponding handler control block is created and placed on the stack(s). It is removed again on exiting the handled clause. The details differ for exception and interrupt handlers.

## 4.1 Exception Handlers

When execution enters a clause with an associated exception handler, the *enterEHandled* instruction places an exception handler control block (EHCB) on the main stack. On leaving the clause the *exitEHandled* instruction removes the EHCB. The EHSP register points to the most recent EHCB, or **nil** if no exception handlers are in scope. Each EHCB contains the following:

- a link to the next outer EHCB on the main stack (EHL)
- the current value of FB for the frame containing the handler (EHFB)
- the address of the exception handler code (EH@)
- the current value of PSP for the frame containing the handler (EHPSP)

An exception raised explicitly is treated as a jump to the exception handler code. The **raise** may occur at a point statically outside a **handle** clause, either due to the current procedure having been called within a **handle** clause, or due to no handlers having been declared at all. In the first case several frames may have to be discarded, while in the latter case the default exception handling code is called and the thread halted.

The *invokeEHandler* instruction executes the **raise** clause. If EHSP is **nil**, the default hard-coded exception handling code is executed. Otherwise, the instruction uses EHSP to identify the correct EHCB, restores the registers from the EHCB, sets EHSP to point to the next outer EHCB, and jumps to the exception handler code. Should a new **raise** be executed inside the exception handler code then a new *invokeEHandler* instruction will be executed with the correct EHCB.

For implicit exceptions, such as divide-by-zero, an appropriate exception view is created in the heap, a pointer to it is pushed onto the pointer stack, and the *invokeEHandler* instruction is executed.

## 4.2 Interrupt Handlers

When execution enters a clause with an associated interrupt handler, the *enterIHandled* instruction places an interrupt handler control block (IHCB) on the stacks. On leaving the clause the *exitIHandled* instruction removes the IHCB. The IHSP register points to the most recent IHCB, or **nil** if no interrupt handlers are in scope. Each IHCB contains the following:

main stack:

- a link to the next outer IHCB on the main stack (IHL)

- a link to the pointer elements of the IHCB on the pointer stack (IHPL)
- an integer identifying the interrupt handled by this handler (IHI)

pointer stack:

- a pointer to the interrupt handler code vector (IHCV)
- a pointer to the interrupt handler environment (IHE)

An interrupt may be raised to a particular ProcessBase thread by the associated PBAM thread, or by a different PBAM thread. The originating PBAM thread must ensure that it is in a globally consistent state before raising the interrupt. The steps involved in the raise include:

- the originating PBAM thread locates the current corresponding interrupt handler procedure for the ProcessBase thread, if any;
- if an interrupt handler is found, the interrupt parameters if any are pushed onto the stacks of the ProcessBase thread, which then executes the interrupt handler;
- if the originating thread is not the PBAM thread that is executing the ProcessBase thread, the originating thread blocks until the interrupt handler returns;
- any result returned by the interrupt handler procedure is returned to the point at which the interrupt was raised in the originating PBAM thread.

The appropriate interrupt handler is located by traversing the list of IHCBs, starting from IHSP and comparing the interrupt identifier number in each block with the number of the interrupt being raised. If a match is found the closure of the interrupt handler is pushed onto the pointer stack.

Two C data structures are then created and pointers to them are pushed onto the main stack. The first, the interrupt result pointer (IRP) is used to store any result returned by the interrupt handler where it may be accessed by the PBAM code following the interrupt. The second, the interrupt resume context (IRC) records the current execution context at the point that the interrupt is raised, so that control may be resumed there when the interrupt returns.

IRC is an instance of the array type *jmp_buf*, while IRP is an instance of the *struct* type *interrupt_result*:

```
typedef struct {

        psptr *p1;          // first pointer
        psptr *p2;          // second pointer
        psint w1;           // first scalar
        psint w2;           // second scalar

} interrupt_result;
```

The mark stack control word is completed by pushing **nil** for the dynamic link and pointer stack link, and a dummy value for the return address, onto the main stack. Any interrupt parameters are converted to ProcessBase format and pushed on the appropriate stacks. An **apply** instruction is executed. Interpretation of ProcessBase code then continues as normal until the interrupt handler procedure returns and the **nil** dynamic link is encountered. Finally, the **return** instruction copies the procedure result, if any, into the result structure and transfers control back to the stored context.

To enable the same procedure return mechanism to work on thread completion, IRP and IRC pointers are also pushed onto the main stack on thread initialisation. In this case IRP is set to **nil**, since any result is discarded, and IRC is set to the context following the instruction decode loop.

## 4.3 Down-calls

The ProcessBase down-call mechanism allows a user program to invoke a PBAM instruction directly. This does not affect the design of the PBAM, except that the core instructions are categorised into safe and unsafe instructions. A safe instruction is one that may be invoked safely in any context in the PBAM code stream, subject only to the condition that its expected stack parameters have been correctly placed on the appropriate stacks. Safety means that the instruction has no net effect on the stacks other than to pop its stack parameters and to push its result if any. If there is a result value, it must have a fixed ProcessBase type.

# 5 PBAM Code

The PBAM instructions fall naturally into groups.

Typed instructions have an encoded name with the following convention:

| | |
|---|---|
| IB | integer or boolean |
| I | integer |
| R | real |
| S | string |
| P | vector, view, loc |
| Pr | procedure |
| any | any |

Non type-dependent instructions are encoded according to the size of the objects on which they operate and on which stack they operate, 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 |

The length and interpretation of instruction parameters is as follows:

| | | |
|---|---|---|
| byte | 8 bits | an 8 bit integer, unsigned unless used with the literal integer instruction |
| short | 2 bytes | an unsigned 16 bit integer, the first byte most significant |

All instruction codes are one byte long.

## 5.1 Jump

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

| Instruction | Op-Code | Description |
|---|---|---|
| fJump (n:short) | 0 | Jump forwards *n* bytes. |

| Instruction | Op-Code | Description |
|---|---|---|
| bJump (n:short) | 1 | Jump backwards *n* bytes. |

| Instruction | Op-Code | Description |
|---|---|---|
| jumpF (n:short) | 2 | **if** the top main stack element is **false**<br>**do** Jump forwards *n* bytes.<br>Pop the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| jumpFF (n:short) | 4 | **if** the top main stack element is **false**<br>**then** Jump forwards *n* bytes<br>**else** Pop the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| jumpTT (n:short) | 5 | **if** the top main stack element is **true**<br>**then** Jump forwards *n* bytes<br>**else** Pop the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| forTest (n:short) | 6 | 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.<br>**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<br>**do** Pop the top 3 stack elements and jump forwards *n* bytes |

| Instruction | Op-Code | Description |
|---|---|---|
| forStep (n:short) | 7 | 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.<br>Add the for loop increment to the for loop control constant. Jump backwards *n* bytes, (to the *forTest* instruction). |

## 5.2 Assignment

Assignments in ProcessBase may only be made to locations. The value to be assigned is always found at the top of the appropriate stack. Once the value is popped off the stack, the address of the location to be assigned to is always on the top of the pointer stack. A different form of the instruction is used for each different kind of assignment. The assignment instructions are:

| Instruction | Op-Code | Description |
|---|---|---|
| assign<br>wAssign<br>dwAssign<br>pAssign<br>dpAssign | <br>10<br>11<br>12<br>13 | **atomic** [<br>**if** the instruction is *wAssign* or *dwAssign*<br>**then** the location pointer is on the top of the pointer stack.<br>**else** the location pointer is under the value on the pointer stack.<br>**if** the instruction is *dwAssign* or *dpAssign*<br>**do** Pop a word from the appropriate stack and copy it to word 3 of the location.<br>Pop a word from the appropriate stack and copy it to word 2 of the location.<br>]<br>Pop the location pointer from the pointer stack. |

A special instruction is used to initialise the elements of a vector in the **using** clause.

| Instruction | Op-Code | Description |
|---|---|---|
| Vassign | | **atomic** [ |
| wVassign | 14 | Pop the word *w1* from the appropriate stack. |
| dwVassign | 15 | **if** the instruction is *dwVassign* or *dpVassign* |
| pVassign | 16 | **do** Pop the word *w2* from the appropriate stack. |
| dpVassign | 17 | |
| | | Pop the vector index from the main stack. |
| | | Pop the pointer to the vector from the pointer stack. |
| | | Calculate the word offset *n* of the vector element within the vector. |
| | | **if** the instruction is *dwVassign* or *dpVassign* |
| | | **then** Copy the value of *w2* to word *n* of the vector. |
| | | Copy the value of *w1* to word *n+1* of the vector. |
| | | **else** Copy the value of *w1* to word *n* of the vector. |
| | | ] |
| | | (There is no need to check the bounds of the vector, since the index is generated by the compiler). |

## 5.3 Stack Load

The stack load instructions are used to push a value onto the top of a stack. The value may be contained in the root object, local frame, a location, a vector, a view, the procedure environment, an any or the code vector. Variations of the load instruction exist depending on: whether the object pointer is kept in a register or on the pointer stack; where the displacement (in words) of the field from the base of the object is found; and the type of value being loaded. These are outlined below:

| Source | Object Pointer Location | Displacement (d) |
|---|---|---|
| root object | ROP | Instruction |
| local frame | FB or PFB | Instruction |
| location | Top of PS | Implicit |
| vector | Top of PS | On main stack |
| view | Top of PS | Instruction |
| environment | E | Instruction |
| any | Top of PS | Implicit |
| code vector | PS location pointed at by PFB | Instruction |

A separate instruction exists for each form with different instructions used for the separate stacks and value sizes.

The stack load instructions are as follows:

| Instruction | Op-Code | Description |
|---|---|---|
| root (d:short) | | **atomic** [ |
| wRoot | 20 | Push word *d* of the root object onto the appropriate stack. |
| dwRoot | 21 | **if** the instruction is *dwRoot* or *dpRoot* |
| pRoot | 22 | **do** Push word *d+1* of the root object onto the appropriate stack. |
| dpRoot | 23 | ] |

| Instruction | Op-Code | Description |
|---|---|---|
| local (d:short) | | **atomic** [ |
| wLocal | 24 | Push word *d* of the local frame onto the appropriate stack. |
| dwLocal | 25 | **if** the instruction is *dwLocal* or *dpLocal* |
| pLocal | 26 | **do** Push word *d+1* of the local frame onto the appropriate stack. |
| dpLocal | 27 | ] |

| Instruction | Op-Code | Description |
|---|---|---|
| deref | | **atomic** [ |
| wDeref | 28 | Pop the pointer to the location from the pointer stack. |
| dwDeref | 29 | Push word 2 of the location onto the appropriate stack. |
| pDeref | 30 | **if** the instruction is *dwDeref* or *dpDeref* |
| dpDeref | 31 | **do** Push word 3 of the location onto the appropriate stack. |
| | | ] |

| Instruction | Op-Code | Description |
|---|---|---|
| subVector | | **atomic** [ |
| wSubVector | 32 | Pop the vector index from the main stack. |
| dwSubVector | 33 | Pop the pointer to the vector from the pointer stack. |
| pSubVector | 34 | Compare the index with the lower and upper bounds of the vector. |
| dpSubVector | 35 | **if** index is outwith the bounds |
| | | **then** raise **vector exception** |
| | | **else** Calculate the word offset of the indexed element. |
| | | Push the first word of the indexed element onto the appropriate stack. |
| | | **if** the instruction is *dwSubVector* or *dpSubVector* |
| | | **do** Push the second word of the indexed element onto the appropriate stack. |
| | | ] |

| Instruction | Op-Code | Description |
|---|---|---|
| subView (d:short)<br>wSubView<br>dwSubView<br>pSubView<br>dpSubView | <br>36<br>37<br>38<br>39 | **atomic** [<br>Pop the pointer to the view from the pointer stack.<br>Push word *d* of the view onto the appropriate stack.<br>**if** the instruction is *dwSubView* or *dpSubView*<br>**do** Push word *d+1* of the view onto the appropriate stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| env (d:short)<br>wEnv<br>dwEnv<br>pEnv<br>dpEnv | <br>40<br>41<br>42<br>43 | **atomic** [<br>Push word *d* of the environment onto the appropriate stack.<br>**if** the instruction is *dwEnv* or *dpEnv*<br>**do** Push word *d+1* of the environment onto the appropriate stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| project<br>wProject<br>dwProject<br>pProject<br>dpProject | <br>44<br>45<br>46<br>47 | **atomic** [<br>Pop the pointer to the any from the pointer stack.<br>Push word 3 of the any onto the appropriate stack.<br>**if** the instruction is *dwProject* or *dpProject*<br>**do** Push word 4 of the any onto the appropriate stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| loadAnyType | 48 | **atomic** [<br>Pop the pointer to the any from the pointer stack.<br>Push word 2 of the any onto the pointer stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|

| | | |
|---|---|---|
| literal ( p:short<br>d:short)<br>wLiteral<br>dwLiteral<br>pLiteral<br>dpLiteral | <br><br>50<br>51<br>52<br>53 | **atomic** [<br>*p* is the displacement of the appropriate literal area from the start of the code vector.<br>Push word *d* of the appropriate literal area onto the appropriate stack.<br>**if** the instruction is *dwLiteral* or *dpLiteral*<br>**do** Push word *d+1* of the appropriate literal area onto the appropriate stack.<br>] |

There are two other instructions used to load the value of a literal onto the appropriate stack. They are:

| Instruction | Op-Code | Description |
|---|---|---|
| lLInt (n:byte) | 54 | Push the signed integer value *n* onto the main stack.<br>The byte is an 8 bit twos complement number. |

| Instruction | Op-Code | Description |
|---|---|---|
| lLChar (n:byte) | 55 | **atomic** [<br>Lookup the vector of single character strings in the library dependent root object.<br>Use *n* as an index into the vector.<br>Push the indexed string element onto the pointer stack.<br>] |

## 5.4 Block Exit

| Instruction | Op-Code | Description |
|---|---|---|
| retract ( ms:short, ps:short) | | **atomic** [ |
| | | **if** the instruction is not *retract* |
| wRetract | 60 | **do** Pop the word *w1* from the appropriate stack. |
| dwRetract | 61 | **if** the instruction is *dwRetract* or *dpRetract* |
| pRetract | 62 | **do** Pop the word *w2* from the appropriate stack. |
| dpRetract | 63 | Pop *ms* words from the main stack. |
| retract | 64 | Pop *ps* words from the pointer stack. |
| | | **if** the instruction is *dwRetract* or *dpRetract* |
| | | **do** Push the value of *w2* onto the appropriate stack. |
| | | **if** the instruction is not *retract* |
| | | **do** Push the value of *w1* onto the appropriate stack |
| | | ] |

## 5.5 Procedure Entry and Exit

The instruction sequence to call a procedure is: load the procedure, markStack, evaluate the parameters, and apply.

| Instruction | Op-Code | Description |
|---|---|---|
| markStack | 70 | Place the values of FB (DL) and PSP - 2 (PSL) on the main stack.<br>Leave space for the return address (RA) on the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| apply ( ms:short,<br>ps:short) | 71 | **atomic** [<br>The main stack parameters start at word *ms* in the current frame.<br>The code vector for the procedure being applied, the new code vector, is at (*ps* - *2*) in the current frame.<br>The environment pointer for the procedure being applied, the new environment, is above the new code vector on the pointer stack.<br>Check that there is sufficient stack space for the procedure being applied; the sizes are held in the new code vector (in words).<br>Increment the pointer count in the header of the stack object by the size of the new pointer frame.<br>Save the offset (in bytes) of CP from the start of the current code vector, in the main frame (the return address (*ms* - *1*)).<br>Set E to the new environment.<br>Set FB to the new frame base (*ms* - *3*).<br>Set PFB to the pointer frame base (*ps* - *2*).<br>Set CP to the start of the abstract machine code in the new code vector.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| return<br>wReturn<br>dwReturn<br>pReturn<br>dpReturn<br>return | <br>74<br>75<br>76<br>77<br>78 | **atomic** [<br>Decrement the pointer count in the header of the stack object by the size of the current pointer frame.<br>**if** the dynamic link is **nil**<br>**then** The IRP at (FB-2) points to a C *struct* containing 2 pointers followed by 2 scalars.<br>The IRC at (FB-1) points to a C *jmp_buf* array containing the resume context.<br>Copy the result of the procedure, if any, from the top(s) of the appropriate stack(s) into the appropriate words of the C *struct*.<br>Copy the *jmp_buf* array into temporary memory.<br>De-allocate the *jmp_buf* array.<br>Perform a *longjmp* to the copied resume context.<br>**else** Copy and pop the result of the procedure at the top of the appropriate stack.<br>Copy RA from the main stack (FB+2).<br>Set PSP to PSL (FB+1).<br>Set SP to FB.<br>Set FB to the dynamic link of the current frame.<br>Set PFB from the PSL in the resumed frame.<br>Set E from ENV in the resumed frame.<br>Push the result of the procedure onto the appropriate stack.<br>Set CP to the start of the resumed code vector + the RA saved earlier.<br>] |

## 5.6 Heap Object Creation

These instructions create heap objects. The objects are then initialised from the stacks and a pointer to them left on the top of the pointer stack. Heap objects are created for locations, strings, vectors, views, procedure environments and anys.

### 5.6.1 Location

| Instruction | Op-Code | Description |
|---|---|---|
| makeLoc<br>wMakeLoc<br>dwMakeLoc<br>pMakeLoc<br>dpMakeLoc | <br>80<br>81<br>82<br>83 | **atomic** [<br>Create a heap location of the appropriate size.<br>Set the last word of the location to 0.<br>**if** the instruction is *dwMakeLoc* or *dpMakeLoc*<br>**do** Pop the word from the appropriate stack and copy it to word 3 of the location.<br>Pop the word from the appropriate stack and copy it to word 2 of the location.<br>Push the pointer to the new object onto the pointer stack.<br>] |

### 5.6.2 String

| Instruction | Op-Code | Description |
|---|---|---|
| concatenate | 84 | **atomic** [<br>Pop the second string from the pointer stack.<br>Pop the first string from the pointer stack.<br>**if** the total length of the two strings is greater than the longest possible string<br>**then** raise **string exception**<br>**else** Create a new string whose length is the sum of the lengths of the two strings.<br>Copy the characters of the first string into the new string, followed by the characters of the second string.<br>Set the last word of the new string to 0.<br>Push the new string onto the pointer stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| subString | 85 | **atomic** [<br>Pop the new length from the main stack.<br>Pop the starting position of the new string from the main stack.<br>Pop the string from the pointer stack.<br>Compare the new string’s start and length with the length of the sub-scripted string.<br>**if** the new string is not a substring of the sub-scripted string or has a negative length<br>**then** raise **string exception**<br>**else** **if** the new string is shorter than the subscripted string<br>    **then** Create the new string and copy its characters from the sub-scripted string, starting at the start position.<br>        Set the last word of the new string to 0.<br>        Push the new string onto the pointer stack.<br>    **else** Push the original string onto the pointer stack.<br>] |

### 5.6.3 Vector

| Instruction | Op-Code | Description |
|---|---|---|
| makeVector (n:short) | | There are *n* initialising elements for the vector on the appropriate stack. |
| wMakeVector | 88 | Calculate the size of the vector in words. |
| dwMakeVector | 89 | **atomic** [ |
| pMakeVector | 90 | Create a vector object of the calculated size. |
| dpMakeVector | 91 | **for** *i* = *n* **to** 1 **by** -1 **do** |
| | | **begin** |
| | | **if** the instruction is *dwMakeVector* or *dpMakeVector* |
| | | **then** Pop the word from the appropriate stack and copy it to word *2i+1* of the vector. |
| | | Pop the word from the appropriate stack and copy it to word *2i* of the vector. |
| | | **else** Pop the word from the appropriate stack and copy it to word *i+1* of the vector. |
| | | **end** |
| | | Pop the lower bound from the main stack and place it in the vector. |
| | | Calculate the upper bound and place it in the vector. |
| | | Set the last word of the vector to 0. |
| | | Push the pointer to the new object onto the pointer stack. |
| | | ] |

| Instruction | Op-Code | Description |
|---|---|---|
| makeEvec<br>wMakeEvec<br>dwMakeEvec<br>pMakeEvec<br>dpMakeEvec | <br>92<br>93<br>94<br>95 | Pop the upper bound for the vector from the main stack.<br>Pop the lower bound for the vector from the main stack.<br>**if** the lower bound is greater than the upper bound<br>**then** raise **vector exception**<br>**else** Calculate the size of the vector in words.<br>**atomic** [<br>Create a vector object of the calculated size.<br>Initialise the elements of the vector to 0 or **nil** as appropriate.<br>Place the lower and upper bounds in the vector.<br>Set the last word of the vector to 0.<br>Push the pointer to the new object onto the pointer stack.<br>] |

## 5.6.4 View

| Instruction | Op-Code | Description |
|---|---|---|
| makeView (m:short, n:short ) | 96 | **atomic** [<br>Create an object of size *m+3* words with *n* pointer fields.<br>Set the last word of the view to 0.<br>Pop an address map vector containing *m* integer elements from the pointer stack.<br>Each element of the address map indicates the offset in the view object to which the corresponding stack word should be copied.<br>Element 1 maps the pointer stack value below the address map, element *n* maps the bottom pointer stack value, element *n+1* maps the top main stack value and element *m* maps the bottom main stack value.<br>Pop *n* words from the pointer stack and *m-n* words from the main stack, placing them in the view object at the offsets indicated by the corresponding elements of the address map.<br>Push the pointer to the new object onto the pointer stack.<br>] |

### 5.6.5 Environment

| Instruction | Op-Code | Description |
|---|---|---|
| formClosure (m:short, n:short) | 98 | **atomic** [<br>Create an object of size *m+3* words with *n* pointer fields.<br>Set the last word of the environment to 0.<br>Pop an address map vector containing *m* integer elements from the pointer stack.<br>The first *n* elements of the vector indicate where the environment pointer values are to be found (on the pointer stack or in the current environment). The addresses are offsets from PFB if positive and E if negative.<br>Elements *n+1* to *m* of the vector indicate where the non-pointer values are to be found (on the main stack or in the current environment). The addresses are offsets from FB if positive and E if negative.<br>Use the map to copy the values from the stacks or the current environment, placing them in the new environment object.<br>Push the pointer to the new object onto the pointer stack.<br>] |

### 5.6.6 Any

| Instruction | Op-Code | Description |
|---|---|---|
| makeAny (m:short)<br>wMakeAny<br>dwMakeAny<br>pMakeAny<br>dpMakeAny | <br>100<br>101<br>102<br>103 | **atomic** [<br>Create a heap location of the appropriate size.<br>**if** the instruction is *dwMakeAny* or *dpMakeAny*<br>**do** Pop a word from the appropriate stack and copy it into word 4 of the any.<br>Pop a word from the appropriate stack and copy it into word 3 of the any.<br>Copy *m* into the second last word of the any.<br>Pop a type representation from the pointer stack and copy it into word 2 of the any.<br>Set the last word of the any to 0.<br>Push the pointer to the new object onto the pointer stack.<br>] |

## 5.7 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.

| Instruction | Op-Code | Description |
|---|---|---|
| equals | | **atomic** for *eqS*, *eqP*, *eqPr*, *eqAny* [ |
| eqIB | 110 | Pop two elements from the appropriate stack. |
| eqR | 111 | **if** the two elements are equal |
| eqS | 112 | **then** Push the boolean value **true** onto the main stack. |
| eqP | 113 | |
| eqPr | 114 | **else** Push the boolean value **false** onto the main stack. |
| eqAny | 115 | |
| | | ] |

Equality of the stack elements is defined as follows:

| | |
|---|---|
| eqIB | The elements are single words on the main stack; they must have the same integer value. |
| eqR | The elements are pairs of words on the main stack; they must be compared by the floating point implementation. |
| eqS | The elements are pointers to strings on the pointer stack; they must be the same pointer **or** they must have exactly the same characters. |
| eqP | The elements are single words on the pointer stack; they must have the same integer value. |
| eqPr | 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 environments. The code vectors and the environments must have the same integer value. |
| eqAny | The dynamic tags of the injected values must be equal. The types of two anys must be equal as checked by the *eqAny* procedure in the root object. Finally the values must be equal as defined above. |

| Instruction | Op-Code | Description |
|---|---|---|
| not.equals | | **atomic** for *neqS*, *neqP*, *neqPr* [ |
| neqIB | 116 | Pop two elements from the appropriate stack. |
| neqR | 117 | **if** the two elements are equal |
| neqS | 118 | **then** Push the boolean value **false** onto the main stack. |
| neqP | 119 | |
| neqPr | 120 | **else** Push the boolean value **true** onto the main stack. |
| | | ] |

It should be noted that there is no *neqAny* instruction. The comparison is implemented by performing an *eqAny* followed by a *not* instruction.

| Instruction | Op-Code | Description |
|---|---|---|
| lessThan<br>ltI<br>ltR<br>ltS | <br>122<br>123<br>124 | **atomic** for *ltS* [<br>Pop element *B* from the appropriate stack.<br>Pop element *A* from the appropriate stack.<br>**if** element *A* is less than element *B*<br>**then** Push the boolean value **true** onto the main stack.<br>**else** Push the boolean value **false** onto the main stack.<br>] |

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

| | |
|---|---|
| ltI: | The elements *A* and *B* are single words on the main stack; element *A* must have a smaller integer value than element *B*. |
| ltR: | 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. |
| ltS: | The elements *A* and *B* are pointers to strings on the pointer stack.<br>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.<br>If all of a string's characters have been compared *A*'s string must be shorter than *B*'s string.<br>If two characters differ the character from *A*'s string must have a smaller ASCII code than the character from *B*'s string. |

| Instruction | Op-Code | Description |
|---|---|---|
| lessThanOrEqual<br>leI<br>leR<br>leS | <br>126<br>127<br>128 | **atomic** for *leS* [<br>Pop element *B* from the appropriate stack.<br>Pop element *A* from the appropriate stack.<br>**if** element *A* is less than or equal to element *B*<br>**then** Push the boolean value **true** onto the main stack.<br>**else** Push the boolean value **false** onto the main stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| greaterThan<br>gtI<br>gtR<br>gtS | <br>130<br>131<br>132 | **atomic** for *gtS* [<br>Pop element *B* from the appropriate stack.<br>Pop element *A* from the appropriate stack.<br>**if** element *A* is less than or equal to element *B*<br>**then** Push the boolean value **false** onto the main stack.<br>**else** Push the boolean value **true** onto the main stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| greaterThanOrEqual<br>geI<br>geR<br>geS | <br>134<br>135<br>136 | **atomic** for *geS* [<br>Pop element *B* from the appropriate stack.<br>Pop element *A* from the appropriate stack.<br>**if** element *A* is less than element *B*<br>**then** Push the boolean value **false** onto the main stack.<br>**else** Push the boolean value **true** onto the main stack.<br>] |

## 5.8 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*. Note that each real number is two stack words long.

| Instruction | Op-Code | Description |
|---|---|---|
| plus<br>fPlus | 140<br>150 | Pop values *A* and *B* from the main stack.<br>Add *A* and *B*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the result onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| times<br>fTimes | 141<br>151 | Pop values *A* and *B* from the main stack.<br>Multiply *A* and *B*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the result onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| minus<br>fMinus | 142<br>152 | Pop value *B* from the main stack.<br>Pop value *A* from the main stack.<br>Subtract *B* from *A*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the result onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| div | 143 | Pop integer value *B* from the main stack.<br>Pop integer value *A* from the main stack.<br>Divide *A* by *B*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the quotient of *A* divided by *B* onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| fDivide | 153 | Pop floating point value *B* from the main stack.<br>Pop floating point value *A* from the main stack.<br>Divide *A* by *B*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the floating point value of *A* divided by *B* onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| neg<br>fNeg | 144<br>154 | Pop value *A* from the main stack.<br>Negate *A*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the result onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| rem | 145 | Pop integer value *B* from the main stack.<br>Pop integer value *A* from the main stack.<br>Divide *A* by *B*.<br>**if** an arithmetic error occurs<br>**then** raise **arithmetical exception**<br>**else** Push the remainder of *A* divided by *B* onto the main stack. |

| Instruction | Op-Code | Description |
|---|---|---|
| not | 146 | Pop boolean value *A* from the main stack.<br>**if** *A* is **true**<br>**then** Push the boolean value **false** onto the main stack.<br>**else** Push the boolean value **true** onto the main stack. |

## 5.9 Exception and Interrupt Handling

| Instruction | Op-Code | Description |
|---|---|---|
| enterEHandled (n:short) | 160 | Push the value of EHSP onto the main stack.<br>Push the value of FB onto the main stack.<br>Push *n* onto the main stack. *n* is the offset of the handler clause from the start of the current code vector (in bytes).<br>Push the value of PSP onto the main stack.<br>Put the value SP – 4 in EHSP.<br>The main stack now contains the EHCB for this clause and EHSP points to it. |

| Instruction | Op-Code | Description |
|---|---|---|
| exitEHandled | 161 | Place the value on the main stack pointed at by EHSP (EHL) in EHSP. |

| Instruction | Op-Code | Description |
|---|---|---|
| invokeEHandler | 162 | **atomic** [<br>Pop a pointer to an exception view from the pointer stack.<br>**if** EHSP is **nil**<br>**then** Print the exception name and description to the standard output.<br>Exit the current thread.<br>**else** Place the value pointed at by EHSP + 3 (EHPSP) in PSP.<br>Push the exception view pointer onto the pointer stack.<br>Place the value pointed at by EHSP + 1 (EHFB) in FB.<br>Place the value pointed at by FB + 1 (PSL) in PFB.<br>Place the value pointed at by EHSP + 2 (EH@) in CP and add the code vector base into CP. This can be found at PFB.<br>Place the environment pointer at PFB + 1 in E.<br>Set SP to EHSP.<br>Place the value pointed at by EHSP (EHL) in EHSP.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| enterIHandled (n:short) | 163 | The closure for the interrupt handler procedure is on the pointer stack.<br>Push *n*, the identifier for the handled interrupt, onto the main stack.<br>Push the value of IHSP onto the main stack.<br>Push the value pointed at by IHSP + 1 (IHPL) onto the main stack.<br>Place the value SP – 3 in IHSP.<br>The stacks now contain the IHCB for this clause and IHSP points to it. |

| Instruction | Op-Code | Description |
|---|---|---|
| exitIHandled | 164 | Place the value pointed at by IHSP (IHL) in IHSP. |

## 5.10 Miscellaneous

| Instruction | Op-Code | Description |
|---|---|---|
| lwb | 170 | **atomic** [<br>Pop the pointer to a vector from the pointer stack.<br>Push the lower bound of the vector onto the main stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| upb | 171 | **atomic** [<br>Pop the pointer to a vector from the pointer stack.<br>Push the upper bound of the vector onto the main stack.<br>] |

| Instruction | Op-Code | Description |
|---|---|---|
| hashCode | 172 | **atomic** [<br>Pop a pointer to an any from the pointer stack.<br>Read the dynamic tag from the last word of the any and calculate the number of pointers in the injected value.<br>**if** the number of pointers is 0<br>**then** Push 0 onto the main stack.<br>**else** **if** the number of pointers is 1<br>    **then** Read the object at word 3 of the any.<br>    **else** Read the object at word 4 of the any (a procedure environment).<br>    **if** the last word of the object is 0<br>    **do** Generate a non-zero pseudo-random integer and assign it to the last word of the object.<br>    Push the last word of the object onto the main stack.<br>] |

## 6 References

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1983.html#approach.persistence*

[BCC+88] Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. Universities of Glasgow and St Andrews Technical Report PPRR-59-88 (1988).

[BMM80] Bailey, P.J., Maritz, P. & Morrison, R. “The S-algol Abstract Machine”. University of St Andrews Technical Report CS/80/2 (1979).

[CBC+90] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (eds), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1990) pp 353-366. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1990.html#pam*

[DM88] Davie, A.J.T. & McNally, D.J. “CASE - A Lazy Version of an SECD Machine in a Flat Environment”. University of St Andrews Technical Report Staple/StA/88/2 (1988).

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (eds), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed), ISBN 3-540-19800-8 (1992) pp 86-106. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1992.html#persistent.hyperprograms*

[Kir92] Kirby, G.N.C. “Persistent Programming with Strongly Typed Linguistic Reflection”. In Proc. 25th International Conference on Systems Sciences, Hawaii, Morrison, R. & Atkinson, M.P. (eds) (1992) pp 820-831, Technical Report ESPRIT BRA Project 3070 FIDE FIDE/91/32. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1992.html#programming.reflection*

[MBC+96] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “Napier88 Reference Manual (Release 2.2.1)”. University of St Andrews (1996). URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1996.html#napier.ref.man.221*

[MBG+99] Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G.N.C., Mayes, K., Munro, D.S. & Warboys, B.C. “ProcessBase Reference Manual (Version 1.0.4)”. Universities of St Andrews and Manchester (1999). URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1999.html#ProcessBase.manual*

[MCC+93] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Stemple, D. “Mechanisms for Controlling Evolution in Persistent Object Systems”. Journal of Microprocessors and Microprogramming 17, 3 (1993) pp 173-181. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1993.html#evolution.mechanisms*

[Mor79] Morrison, R. “On the Development of Algol”. Ph.D. Thesis, University of St Andrews (1979). URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1979.html#thesis.rm*

[PS85] “PS-algol Abstract Machine Manual”. Universities of Glasgow and St Andrews Technical Report PPRR-11-85 (1985).

[PS88] “PS-algol Reference Manual, 4th edition”. Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[Str67] Strachey, C. “**Fundamental Concepts in Programming Languages**”. Oxford University Press, Oxford (1967).

[Ten77] Tennent, R.D. “Language Design Methods Based on Semantic Principles”. Acta Informatica 8 (1977) pp 97-112.

# Appendix I: PBAM Operation Codes

## Jumps

| | | | |
|---|---|---|---|
| fJump (short) | 0 | bJump (short) | 1 |
| jumpF (short) | 2 | | |
| jumpFF (short) | 4 | jumpTT (short) | 5 |
| forTest (short) | 6 | forStep (short) | 7 |

## Assignment

| | | | |
|---|---|---|---|
| wAssign | 10 | dwAssign | 11 |
| pAssign | 12 | dpAssign | 13 |
| wVassign | 14 | dwVassign | 15 |
| pVassign | 16 | dpVassign | 17 |

## Stack Load and Assignment

| | | | |
|---|---|---|---|
| wRoot (short) | 20 | dwRoot (short) | 21 |
| pRoot (short) | 22 | dpRoot (short) | 23 |
| wLocal (short) | 24 | dwLocal (short) | 25 |
| pLocal (short) | 26 | dpLocal (short) | 27 |
| wDeref | 28 | dwDeref | 29 |
| pDeref | 30 | dpDeref | 31 |
| wSubVector | 32 | dwSubVector | 33 |
| pSubVector | 34 | dpSubVector | 35 |
| wSubView (short) | 36 | dwSubView (short) | 37 |
| pSubView (short) | 38 | dpSubView (short) | 39 |
| wEnv (short) | 40 | dwEnv (short) | 41 |
| pEnv (short) | 42 | dpEnv (short) | 43 |
| wProject | 44 | dwProject | 45 |
| pProject | 46 | dpProject | 47 |
| loadAnyType | 48 | | |
| wLiteral (short,short) | 50 | dwLiteral (short,short) | 51 |
| pLiteral (short,short) | 52 | dpLiteral (short,short) | 53 |
| lLInt (byte) | 54 | lLChar (byte) | 55 |

## Stack Retract

| | | | |
|---|---|---|---|
| wRetract (short,short) | 60 | dwRetract (short,short) | 61 |
| pRetract (short,short) | 62 | dpRetract (short,short) | 63 |
| retract (short,short) | 64 | | |

## Procedure Entry and Exit

| | | | |
|---|---|---|---|
| markStack | 70 | | |
| apply (short,short) | 71 | | |
| wReturn | 74 | dwReturn | 75 |
| pReturn | 76 | dpReturn | 77 |
| return | 78 | | |

## Locations

| | | | |
|---|---|---|---|
| wMakeLoc | 80 | dwMakeLoc | 81 |
| pMakeLoc | 82 | dpMakeLoc | 83 |

## Strings

| | | | |
|---|---|---|---|
| concatenate | 84 | subString | 85 |

## Vectors

| | | | |
|---|---|---|---|
| wMakeVector (short) | 88 | dwMakeVector (short) | 89 |
| pMakeVector (short) | 90 | dpMakeVector (short) | 91 |
| wMakeEvec | 92 | dwMakeEvec | 93 |
| pMakeEvec | 94 | dpMakeEvec | 95 |

## Views

| | |
|---|---|
| makeView (short, short) | 96 |

## Procedures

| | |
|---|---|
| formClosure (short, short) | 98 |

## Anys

| | | | |
|---|---|---|---|
| wMakeAny (short) | 100 | dwMakeAny (short) | 101 |
| pMakeAny (short) | 102 | dpMakeAny (short) | 103 |

## Comparison Operations

| | | | |
|---|---|---|---|
| eqIB | 110 | eqR | 111 |
| eqS | 112 | eqP | 113 |
| eqPr | 114 | eqAny | 115 |
| neqIB | 116 | neqR | 117 |
| neqS | 118 | neqP | 119 |
| neqPr | 120 | | |
| ltI | 122 | ltR | 123 |
| ltS | 124 | | |
| leI | 126 | leR | 127 |
| leS | 128 | | |
| gtI | 130 | gtR | 131 |

| | | | |
|---|---|---|---|
| gtS | 132 | | |
| geI | 134 | geR | 135 |
| geS | 136 | | |

## Arithmetic and Boolean Operators

| | | | |
|---|---|---|---|
| plus | 140 | times | 141 |
| minus | 142 | div | 143 |
| neg | 144 | rem | 145 |
| not | 146 | | |
| fPlus | 150 | fTimes | 151 |
| fMinus | 152 | fDivide | 153 |
| fNeg | 154 | | |

## Exceptions and Interrupts

| | | | |
|---|---|---|---|
| enterEHandled (short) | 160 | exitEHandled | 161 |
| invokeEHandler | 162 | | |
| enterIHandled (short) | 163 | exitIHandled | 164 |

## Miscellaneous

| | | | |
|---|---|---|---|
| lwb | 170 | upb | 171 |
| hashCode | 172 | | |

## **Standard Libraries** (see ProcessBase Standard Library Manual)

## Maths

| | | | |
|---|---|---|---|
| sin | 200 | cos | 201 |
| exp | 202 | ln | 203 |
| sqrt | 204 | atan | 205 |
| truncate | 206 | float | 207 |
| abs | 208 | rabs | 209 |

## String

| | | | |
|---|---|---|---|
| code | 210 | decode | 211 |
| length | 212 | | |

## I/O

| | | | |
|---|---|---|---|
| readOp (short) | 214 | writeOp (short) | 215 |
| openOp | 216 | createOp | 217 |
| closeOp | 218 | | |

## Threads

| | | | |
|---|---|---|---|
| threadOp (short) | 221 | | |

## Appendix II: Code File Format

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

**1.** PBAM 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 PBAM objects, each of which is 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 PBAM magic number in hexadecimal is 0xFC510000, the least significant 16 bits of which are the PBAM version number.

The code file uses big endian addressing and 32 bit words.

The compiler magic number is the same as in the root object. It is used to compare the versions of PBAM code in the stable store and in the code file. The two sets of PBAM code must have the same compiler magic number in order for the code file to be loaded successfully.

# Appendix III: Code Generation Rules

The code generation rules define the code generated for every legal ProcessBase program. In the code generation rules, ProcessBase syntactic constructs are written in *italics* and the code generated for a construct by using the brackets ⟦⟧. Abstract machine instructions and labels are written in outline. For example, the rule for the **or** expression is

| Source | PBAM Code |
|---|---|
| *E1* **or** *E2* | ⟦*E1*⟧ jumpTT(L)<br>⟦*E2*⟧<br>L: |

Thus an **or** expression generates the code for the expression *E1*, a PBAM jumpTT(L) to label L instruction, followed by the code for the expression *E2*. The value of the label L is indicated by its position in the code stream but appears as a value in the jumpTT instruction.

## Generated Code

The abstract machine code is generated in segments with one segment for every procedure literal and one for the main program. The order of the generated segments is top to bottom with the innermost procedure segments first. The main program segment is generated last.

Session:

| Source | PBAM Code |
|---|---|
| *sequence* | ⟦*declaration*⟧ ⟦*sequence*⟧<br>or<br>⟦*clause*⟧ ⟦*sequence*⟧ |

Type declarations:

| Source | PBAM Code |
|---|---|
| *type_decl* | no code generated |

Object declarations:

| Source | PBAM Code |
|---|---|
| **let** *identifier* ← *clause*<br>**rec let** *identifier*$_1$ ← *literal*$_1$ &<br>*identifier*$_2$ ← *literal*$_2$<br>...<br>*identifier*$_n$ ← *literal*$_n$ | ⟦*clause*⟧<br>⟦*literal*$_1$⟧<br>⟦*literal*$_2$⟧<br>...<br>⟦*literal*$_n$⟧ |

Clauses:

| Source | PBAM Code |
|---|---|
| **if** *clause*$_1$ **do** *clause*$_2$ | [*clause*$_1$] jumpF(L)<br>[*clause*$_2$]<br>L: |

| Source | PBAM Code |
|---|---|
| **if** *clause*$_1$<br>**then** *clause*$_2$<br>**else** *clause*$_3$ | [*clause*$_1$] jumpF(L)<br>[*clause*$_2$] fJump(M)<br>L: [*clause*$_3$]<br>M: |

| Source | PBAM Code |
|---|---|
| **while** *clause*$_1$ **do** *clause*$_2$ | L: [*clause*$_1$] jumpF(M)<br>[*clause*$_2$] bJump(L)<br>M: |

| Source | PBAM Code |
|---|---|
| **for** *identifier* ← *clause*$_1$ **to** *clause*$_2$ **by** *clause*$_3$ **do** *clause*$_4$ | [*clause*$_1$] [*clause*$_2$] [*clause*$_3$]<br>L: forTest(M)<br>[*clause*$_4$] forStep(L)<br>M: |

| Source | PBAM Code |
|---|---|
| **project** *clause* **as** *identifier*<br>**onto** *type_id*$_1$ : *clause*$_1$<br>*type_id*$_2$ : *clause*$_2$<br>...<br>*type_id*$_n$ : *clause*$_n$<br>**default** : *clause*$_m$ | [*clause*]<br>**for** each type_id$_i$ **do**:<br>markStack<br>dpRoot( *offset of type equivalence proc in root object*)<br>pLocal ! any type location<br>loadAnyType ! type representation<br>pLiteral ! *type_id*$_i$<br>apply<br>jumpF(L$_i$)<br>*project* ! particular **project** instruction dictated by *type_id*$_i$<br>[*clause*$_i$]<br>fJump(M)<br>L$_i$:<br>[*clause*$_m$]<br>M: |

| Source | PBAM Code |
|---|---|
| **handle exception** *identifier* **using** *clause*$_1$ **in** *clause*$_2$ | enterEHandled(L)<br>[*clause*$_2$]<br>exitEHandled<br>*retract* ! particular **retract** instruction dictated by type of *clause*$_2$<br>fJump(M)<br>L: [*clause*$_1$]<br>M: |

| Source | PBAM Code |
|---|---|
| **raise** *clause* | [*clause*]<br>invokeEHandler |

| Source | PBAM Code |
|---|---|
| **handle interrupt** *interrupt_identifier* **using** *proc_literal* **in** *clause* | [*proc_literal*]<br>enterIHandled(n) ! int identifier for interrupt<br>[*clause*]<br>exitIHandled<br>*retract* ! particular **retract** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| **downcall** *opcode_identifier* [*int_literal*$_1$, *int_literal*$_2$, ..., *int_literal*$_n$] (*clause*$_1$, *clause*$_2$, ..., *clause*$_n$) | **for** each *clause*$_i$ **do**:<br>[*clause*$_i$]<br>*opcode* (*int_literal*$_1$, *int_literal*$_2$, ..., *int_literal*$_n$)<br>! particular instruction defined by *opcode_identifier* |

| Source | PBAM Code |
|---|---|
| *name* := *clause* | [*name*]<br>[*clause*]<br>*assign* ! particular **assign** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| *identifier* | *local* (local value)<br>*env* (free identifier)<br>! particular instructions dictated by type of *identifier* |

Expressions:

| Source | PBAM Code |
|---|---|
| $E_1$ **or** $E_2$ | ⟦$E_1$⟧ jumpTT(L)<br>⟦$E_2$⟧<br>L: |

| Source | PBAM Code |
|---|---|
| $E_1$ **and** $E_2$ | ⟦$E_1$⟧ jumpFF(L)<br>⟦$E_2$⟧<br>L: |

| Source | PBAM Code |
|---|---|
| ~*E* | ⟦*E* ⟧ not |

| Source | PBAM Code |
|---|---|
| $E_1$ *rel_op* $E_2$ | ⟦$E_1$⟧ ⟦$E_2$⟧ ⟦*rel_op*⟧ |

| Source | PBAM Code |
|---|---|
| $E_1$ *add_op* $E_2$ | ⟦$E_1$⟧ ⟦$E_2$⟧ ⟦*add_op*⟧ |

| Source | PBAM Code |
|---|---|
| $E_1$ *mult_op* $E_2$ | ⟦$E_1$⟧ ⟦$E_2$⟧ ⟦*mult_op*⟧ |

| Source | PBAM Code |
|---|---|
| +*E*<br>-*E* | ⟦*E* ⟧<br>⟦*E* ⟧ neg or<br>⟦*E* ⟧ fNeg |

| Source | PBAM Code |
|---|---|
| *literal* | |
| *int_literal* | wLiteral or lLInt |
| *real_literal* | dwLiteral |
| *bool_literal* | lLInt(0 or 1) |
| *string_literal* | pLiteral (literal) or<br>pRoot( *offset of library dependent root object in root object*)<br>pSubView( *offset of nilstring in library dependent root object*) (nilstring) or<br>lLChar(character ASCII code ) (single character) |
| *view_literal* | pRoot(*offset of nil pointer in root object*) |
| *proc_literal* | dpLiteral formClosure |

| Source | PBAM Code |
|---|---|
| **fun** (…) → *type*; *clause* | ⟦*clause* ⟧ *return* ! particular **return** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| **vector @***clause* **of** [*clause*$_1$, *clause*$_2$, ... *clause*$_n$] | ⟦*clause*⟧ ⟦*clause*$_1$⟧ ⟦*clause*$_2$⟧, ..., ⟦*clause*$_n$⟧<br>*makeVector(n)* ! particular **makeVector** instruction dictated by type of *clause*$_1$ |

| Source | PBAM Code |
|---|---|
| **vector** *clause*$_1$ **to** *clause*$_2$ **using** *clause*$_3$ | ⟦*clause*$_1$⟧ ⟦*clause*$_2$⟧ ⟦*clause*$_3$⟧<br>*makeEvec* ! particular **makeEvec** instruction dictated by return type of procedure<br>wLocal ! init control constant with *clause*$_1$<br>wLocal ! limit *clause*$_2$<br>lLInt(1) ! increment<br>L: forTest(M)<br>dpLocal ! procedure *clause*$_3$<br>wLocal ! control constant<br>apply<br>pLocal ! new vector<br>wLocal ! control constant<br>*Vassign* ! particular **Vassign** instruction dictated by return type of procedure<br>forStep(L)<br>M: pRetract ! remove original ! procedure and bounds |

| Source | PBAM Code |
|---|---|
| (*clause*) | ⟦*clause*⟧ |

| Source | PBAM Code |
|---|---|
| **begin** *sequence* **end**<br>{*sequence*} | ⟦*sequence*⟧<br>*retract* ! particular **retract** instruction dictated by type of *sequence* |

| Source | PBAM Code |
|---|---|
| *clause*$_1$(*clause*$_2$ \| *clause*$_3$) | ⟦*clause*$_1$⟧ ⟦*clause*$_2$⟧ ⟦*clause*$_3$⟧<br>subString |

| Source | PBAM Code |
|---|---|
| *clause*(*clause*$_1$, *clause*$_2$, ..., *clause*$_n$) | ! procedure call<br>⟦*clause*⟧<br>markStack<br>for each *clause*$_i$ do: ⟦*clause*$_i$⟧<br>apply |

| Source | PBAM Code |
|---|---|
| *clause*(*clause*$_1$, *clause*$_2$, ..., *clause*$_n$) | ! vector dereference<br>⟦*clause*⟧<br>**for** i ← 1 **to** n-1 **do**:<br>    ⟦*clause*$_i$⟧<br>    pSubVector<br>⟦*clause*$_n$⟧<br>*subVector*    ! particular **subVector** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| ***view*** ( *identifier*$_1$ ← *clause*$_1$,<br>*identifier*$_2$ ← *clause*$_2$, ...,<br>*identifier*$_n$ ← *clause*$_n$) | ! view creation<br>for each *clause*$_i$ do:<br>    ⟦*clause*$_i$⟧<br>*construct address map*<br>lLInt(*lower bound of address map*)<br>for each element of the address map do:<br>    lLInt(*address map element*)<br>wMakeVector(n)<br>makeView |

| Source | PBAM Code |
|---|---|
| *clause.identifier* | ⟦*clause*⟧<br>*subView*    ! particular **subView** instruction dictated by type of field |

| Source | PBAM Code |
|---|---|
| '*clause* | ⟦*clause*⟧<br>*deref*    ! particular **deref** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| **loc** (*clause*) | ⟦*clause*⟧<br>*makeLoc*    ! particular **makeLoc** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| **any** (*clause*) | ⟦*clause*⟧<br>*makeAny*    ! particular **makeAny** instruction dictated by type of *clause* |

| Source | PBAM Code |
|---|---|
| upb (*clause*) | ⟦*clause*⟧ upb |

| Source | PBAM Code |
|---|---|
| lwb (*clause*) | ⟦*clause*⟧ lwb |

# Index