RATIONAL MACHINES INSTRUCTION SET

VERSION 1.0

January 3, 1983

Chapter 1
INTRODUCTION

This document describes the instruction set as defined by the Rational Machines Architecture. In particular, we provide detailed information regarding the composition and functionality of each instruction. Implementation-specific formats may be found in a corresponding processor reference manual. Additionally, the Rational Machines System Concept document includes a rationale for the organization of this instruction set. In each of these documents, we presume that the reader has an understanding of the semantics of the Ada* programming language.

This document is divided into three major sections, namely:

* GENERAL CONCEPTS -- Chapter 2.
* DETAILED DISCUSSION -- Chapters 3 - 9.
* SUMMARY INFORMATION -- Appendices A - C.

Chapter 2 introduces the primitive classes of objects and exceptions that are recognized by the instruction set. Chapters 3 through 9 are organized by groups of functionally related instructions. These seven chapters provide detailed information on the form and effect of every instruction and their variants. The appendices are provided as a convenience to the reader to aid in locating specific instruction set information.

For detailed information regarding the organization of each class of stacks as defined by the architecture, as well as the definition of modules, messages, and class descriptors, consult the Rational Machines Run-time Structure.

---

*Ada is a trademark of the Department of Defense, Ada Joint Program Office

Chapter 2
GENERAL CONCEPTS

The Rational Machines instruction set directly supports and encourages the use of modern software engineering methodologies. In particular, the instruction set is optimized for supporting the application of object-oriented programming in Ada-like languages. The design of the instruction set is heavily influenced by the premise that a well-structured program consists of many small modular components with controlled and well-specified interfaces.

Every program program segment consists of one or more words , where each word contains one or more instructions. A program segment represents either a task or a package , and so the number of words per segment will vary. On the other hand the number of instructions per word is generally a fixed number for each implementation. Each instruction is further divided into an opcode and one or more fields which provide operand information for the instruction. Formally, we may express word and segment as:

```
type WORD    is array(SEGMENTS.INSTRUCTION_INDEX)     of INSTRUCTION;
type SEGMENT is array(SEGMENTS.DISPLACEMENT range <>) of WORD;
```

2.1. CLASSES

The Rational Machines instruction set is strongly typed, which means that there exists a unique and well-defined set of operations associated with every primitive class recognized by the architecture. We use the term "class" to avoid confusion with the term "type" as used in a context relating to the declaration of (keyword) types in a source program. An object is an instance of a particular class and so may be manipulated by various instructions. No other operations on an object of a given class are legal, and furthermore, as in Ada , objects of incompatible classes may not implicitly operate with each other.

This elementary set of classes was designed to directly and efficiently support the semantics of high-order programming languages similar to Ada. Collectively, we call the primitive types that are recognized by the architecture the OPERAND_CLASS. The operations associated with objects of each OPERAND_CLASS are found in Appendix B, the OBJECT/OPERATION CROSS_REFERENCE.

The following classes are recognized by the architecture:

```
type OPERAND_CLASS is
  (ACCESS_CLASS,          ANY_CLASS,        ARRAY_CLASS,
   DISCRETE_CLASS,        ENTRY_CLASS,      EXCEPTION_CLASS,
   FAMILY_CLASS,          FLOAT_CLASS,      MATRIX_CLASS,
   MODULE_CLASS,          PACKAGE_CLASS,    RECORD_CLASS,
   SEGMENT_CLASS,         SELECT_CLASS,     SUBARRAY_CLASS,
   SUBMATRIX_CLASS,       SUBVECTOR_CLASS,  TASK_CLASS,
   VARIANT_RECORD_CLASS,  VECTOR_CLASS);
```

Consult the Rational Machines Run-time Structure for a complete definition regarding the characteristics of each of these classes and their representation on the various machine stacks. Following, we provide a summary description of the primitive classes:

REORDER

ACCESS_CLASS	Denotes a pointer to an object of a specific class.
ANY_CLASS	Denotes an object of an arbitrary class. ANY_CLASS objects are operands only of DECLARE_TYPE, DECLARE_VARIABLE and EXECUTE, and so represent generic declarative and imperative instructions. Characteristics of ANY_CLASS objects are bound at execution time.
ARRAY_CLASS	Denotes a composite object consisting of components of the same component class indexed by n-dimensions.
DISCRETE_CLASS	Denotes an enumeration or integer object.
ENTRY_CLASS	Denotes an entry of a task.
EXCEPTION_CLASS	Denotes an exception.
FAMILY_CLASS	Denotes a family of entries.
FLOAT_CLASS	Denotes a floating point object.
MATRIX_CLASS	Denotes a two dimensional array. MATRIX_CLASS objects are used entirely to support EXECUTE instruction optimizations.
MODULE_CLASS	Denotes a package or a task.
PACKAGE_CLASS	Denotes a package.
RECORD_CLASS	Denotes a composite object consisting of named components, which may be of different classes.
SEGMENT_CLASS	Denotes a program segment. Objects of this class are used to transform a data segment of some form into executable code.
SELECT_CLASS	Denotes an executable object that handles processing of a task select statement.

SUBARRAY_CLASS Denotes an n-1 dimensional array as a substructure of an n-dimensional parent. SUBARRAY_CLASS objects are used entirely to support EXECUTE optimizations.

SUBMATRIX_CLASS Denotes a one dimensional array as a substructure of a two dimensional parent. SUBMATRIX_CLASS objects are used entirely to support EXECUTE optimizations.

SUBVECTOR_CLASS Denotes a slice of a one dimensional parent. SUBVECTOR_CLASS objects are used entirely to support EXECUTE optimizations.

TASK_CLASS Denotes a task.

VARIANT_RECORD_CLASS
Denotes a discriminated union of objects consisting of named components, which may be of different classes. There exists a fixed part of the record common to all variants, and which contains a discriminant field indicating which one of the possible variants is contained in a particular instance.

VECTOR_CLASS Denotes a one-dimensional composite object. VECTOR_CLASS objects are used entirely to support EXECUTE instruction optimization.

A number of instructions operate upon implicit, rather than explicitly referenced objects. We classify such instructions as unclassed.

2.2. EXCEPTIONS

The Rational Machines instruction set defines facilities for dealing with errors that arise during program execution. In particular, the architecture recognizes several different exceptions that cause suspension of normal program execution. These exceptions include:

```
CAPABILITY_ERROR,     CONSTRAINT_ERROR,     ELABORATION_ERROR,
INSTRUCTION_ERROR,    MACHINE_RESTRICTION,  NUMERIC_ERROR,
OPERAND_CLASS_ERROR,  PROGRAM_ERROR,        RESOURCE_ERROR,
STORAGE_ERROR,        TASKING_ERROR,        TYPE_ERROR,
VISIBILITY_ERROR      : exception;
```

Following, we summarize the conditions under which exception may be raised:

CAPABILITY_ERROR
Raised when attempting to access an entity that is private or otherwise out of scope.

CONSTRAINT_ERROR
: Raised in any of the following situations: upon attempt to violate a range constraint, an index constraint, or a discriminant constraint; upon attempt to use a record component that does not exist for the current discriminant values; and upon attempt to use a selected component, an indexed component, a slice, or an attribute, of an object designated by an access value, if the object does not exist because the access value is null.

ELABORATION_ERROR
: Raised when attempting to access an entity that is not yet elaborated.

INSTRUCTION_ERROR
: Raised when the machine attempts to execute an illegal instruction.

MACHINE_RESTRICTION
: Raised when attempting to create an object that is larger than the machine can allocate or index.

NUMERIC_ERROR
: Raised by the execution of a predefined numeric operation that cannot deliver a mathematical result (within the declared accuracy for real types).

OPERAND_CLASS_ERROR
: Raised when attempting to perform an operation that is illegal for an object of the given OPERAND_CLASS.

PROGRAM_ERROR
: Raised during the execution of a selective wait statement that has not else part, if this execution determines that all alternatives are closed. This exception is also raised upon attempt to execute an action that is erroneous, and for incorrect order dependencies.

RESOURCE_ERROR
: Raised when unable to extend a program stack.

STORAGE_ERROR
: Raised in any of the following situations: when the dynamic storage allocated to a task is exceeded; during the execution of an allocator, if the space available for the collection of allocated objects is exhausted; or during the elaboration of a declarative item, or or during the execution of the subprogram call, if storage is not sufficient.

TASKING_ERROR
: This exception is raised when exceptions arise during intertask communication.

TYPE_ERROR
: Raised when attempting to perform an invalid type derivation or completion.

VISIBILITY_ERROR
Raised when attempting to access an entity that is not currently visible.

2.3. OPCODES

The Rational Machines architecture defines a small set of primitive operations, as defined by the type OP_CODE, which we may formally express as:

```
type OP_CODE is
      (ACTION,          BLOCK_BEGIN,      BLOCK_HANDLER,
       CALL,            COMPLETE_TYPE,    DECLARE_SUBPROGRAM,
       DECLARE_TYPE,    DECLARE_VARIABLE, END_LOCALS,
       EXECUTE,         EXIT_ACCEPT,      EXIT_FUNCTION,
       EXIT_PROCEDURE,  EXIT_UTILITY,     INDIRECT_LITERAL,
       JUMP,            JUMP_CASE,        JUMP_NONZERO,
       JUMP_ZERO,       LITERAL_VALUE,    LOAD,
       LOAD_TOP,        POP_BLOCK,        POP_BLOCK_RESULT,
       REFERENCE,       SEGMENT_HEADER,   SEGMENT_TYPE,
       SEGMENT_VALUE,   SHORT_LITERAL,    STORE);
```

We may further classify each OP_CODE according to its function, namely:

DECLARATIVE INSTRUCTION
Provides facilities for the creation of program types and objects.

IMPERATIVE INSTRUCTION
Invokes an operation upon an object of a given class.

DATA MOVEMENT INSTRUCTION
Provides facilities for setting, using, and referencing values of objects.

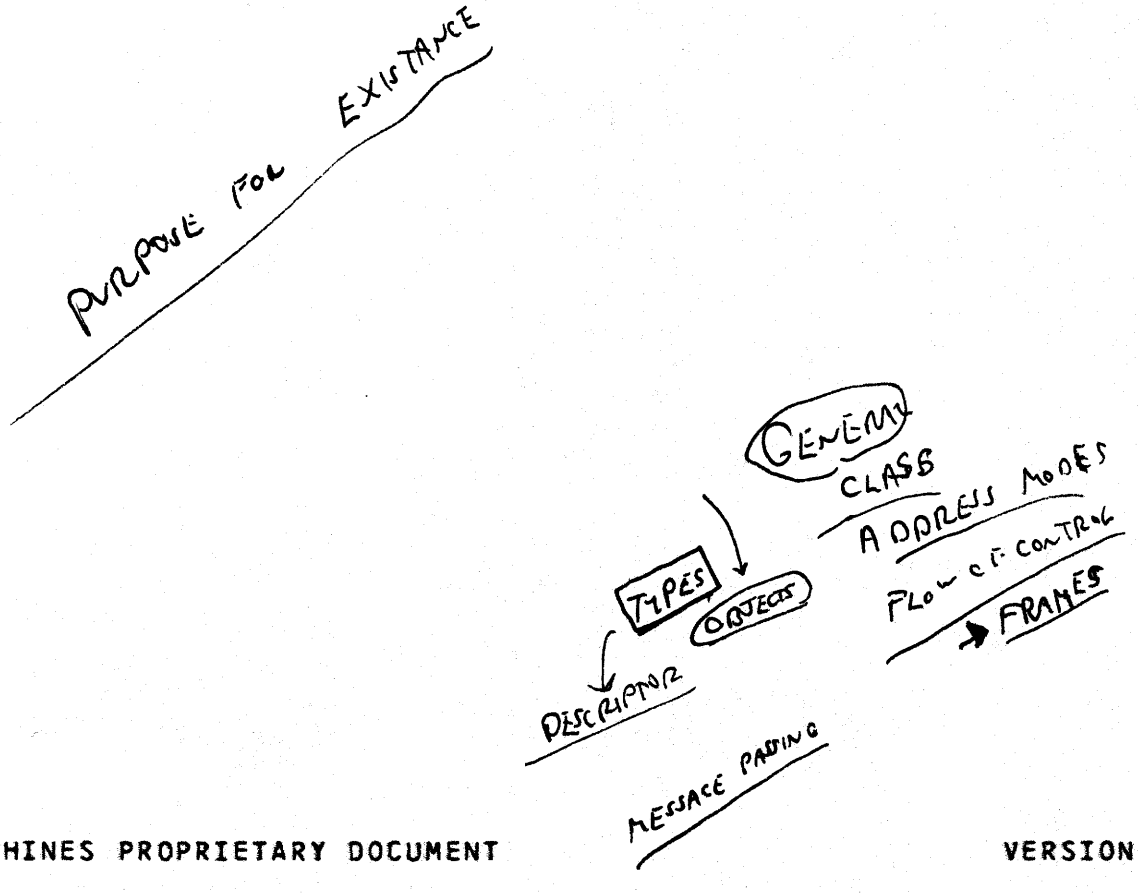
CONTROL TRANSFER INSTRUCTION
Provides facilities for conditional and unconditional changes in the thread of control.

CONTROL RETURN INSTRUCTION
Provides facilities for returning from a subordinate thread of control.

LITERAL DECLARATION
Defines simple and compound literal values.

MODULE LABEL Marks the structure of a given module.

In the following chapters, we provide a detailed definition of each instruction, organized by kind of OP_CODE.



Chapter 3
DECLARATIVE INSTRUCTIONS

A declarative instruction creates the descriptor for a specific type, or an object of a given type or subprogram. Since the definition of a typed entity implies the set of values and operations that are appropriate for a given type, declarative instructions are the first step needed to enforce the architectural principle of strong typing. In addition, this mechanism facilitates the creation of abstract data types and objects, and so permits the architecture to encourage and enforce user-defined abstractions, beyond the relatively small set of primitive class otherwise recognized by the architecture.

Declarative instructions include the following opcodes:

* DECLARE_TYPE -- create the descriptor for a specific type
* COMPLETE_TYPE -- finish the descriptor of an incompletely specified type
* DECLARE_VARIABLE -- create an object of a given type
* DECLARE_SUBPROGRAM -- create a subprogram object

In the following sections, we treat each opcode in detail.

3.1. DECLARE_TYPE

The DECLARE_TYPE instruction creates the descriptor for a specific type.

Formally, DECLARE_TYPE takes the form:

```
type DECLARE_TYPE_INSTRUCTION is
  record
    PRIVACY      : TYPE_PRIVACY;
    TYPE_CLASS   : OPERAND_CLASS;
    TYPE_KIND    : TYPE_SORT;
    TYPE_OPTIONS : TYPE_OPTION_SET;
  end record;
```

In particular, the TYPE_PRIVACY defines the degree of encapsulation of the type (and hence, implies its visibility). TYPE_PRIVACY is defined as:

```
type TYPE_PRIVACY is (IS_LOCAL, IS_PRIVATE, IS_PUBLIC);
```

Generally, IS_LOCAL implies that the type is not visible outside the entity wherein the type is declared, and IS_PRIVATE and IS_PUBLIC imply that the type is visible (and private or not).

Whereas the TYPE_CLASS (of type OPERAND_CLASS) names the class of the target type that is to be declared, a value of TYPE_SORT further defines the manner in which the type is to be declared. Formally, TYPE_SORT is defined as:

```
type TYPE_SORT is
       (CONSTRAINED, CONSTRAINED_INCOMPLETE,
        DEFINED,     DEFINED_INCOMPLETE,
        DERIVED,     DERIVED_INCOMPLETE,
        INCOMPLETE);
```

Basically, as in Ada, a type may be simply defined, derived from another type, or declared as a constrained subtype of a base type. In each case, the type may be incompletely declared (as with access types).

Finally, a value of TYPE_OPTION_SET further defines the nature of the type to be declared. Formally, we have:

```
type TYPE_OPTION_SET is
       record
         BOUNDS_WITH_OBJECT : BOOLEAN;
         CONSTRAINED        : BOOLEAN;
         DERIVED_PRIVACY    : BOOLEAN;
         UNSIGNED           : BOOLEAN;
         WITH_ENTRIES       : BOOLEAN;
       end record;
```

As we will discuss, fields of TYPE_OPTION_SET are relevant only to specific classes of objects.

DECLARE_TYPE is appropriate only for objects of class:

* ACCESS_CLASS
* ARRAY_CLASS
* DISCRETE_CLASS
* FLOAT_CLASS
* PACKAGE_CLASS
* RECORD_CLASS
* SEGMENT_CLASS
* TASK_CLASS
* VARIANT_RECORD_CLASS

The use of any other TYPE_CLASS value will raise the exception INSTRUCTION_ERROR. In the following sections, we will treat each class in detail.

3.1.1. ACCESS_CLASS

In the declaration of an ACCESS_CLASS type, the field TYPE_OPTIONS is not relevant, and therefore ignored. In addition, the TYPE_SORT of a declared ACCESS_CLASS type may only be DEFINED, CONSTRAINED, DERIVED, or INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

CONSTRAINED

PURPOSE: Declare a constrained ACCESS_CLASS type.

FUNCTION: Pop the CONTROL_STACK to determine the parent object, and create a new type compatible to the parent, with an explicit path to the parent. Pop the CONTROL_STACK again to determine the constraint, and apply that constraint to the new type. Write to the TYPE_STACK to reference the enclosing subprogram type, the class of the new type, and the type information of the new type. Push the new ACCESS_VAR on the CONTROL_STACK, and give it an initial value of null.

STACKS: Preconditions: Top of CONTROL_STACK contains the parent ACCESS_VAR. Top - 1 of CONTROL_STACK contains the constraining ACCESS_INFO.

Postconditions: Top of CONTROL_STACK is reduced by two, and then an ACCESS_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR raised if the parent type is private. TYPE_ERROR is raised if parent cannot be further constrained. CONSTRAINT_ERROR is raised if new constraints are not compatible with the parent.

DEFINED

PURPOSE: Declare an ACCESS_CLASS type.

FUNCTION: Create a null type link of kind ACCESS_VAR, and mark its privacy and visibility as appropriate. Pop the CONTROL_STACK to reference a SUBPROGRAM_VAR (which must be FOR_UTILITY) that encapsulate the new type, and create a path to the subprogram. Pop the CONTROL_STACK again to determine the type of the designated access objects, and determine if the designated object IS_HOMOGENEOUS, IS_CONSTRAINED, and its size. Pop a value from from the CONTROL_STACK to determine a page count, and then create a collection for the designated objects. An explicit path is created from the type definition to the collection. Write

to the TYPE_STACK to reference the enclosing subprogram type, the class of the new type, and the type information of the new type. Push the ACCESS_VAR on the CONTROL_STACK, and give it an initial value of null.

STACKS: Preconditions: Top of CONTROL_STACK contains a SUBPROGRAM_VAR. Top - 1 of CONTROL_STACK contains a classed object identifying the type of the designated access objects. Top - 2 of CONTROL_STACK contains a resource value.

Postconditions: An ACCESS_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if SUBPROGRAM_VAR is invalid.

DERIVED

PURPOSE: Declare a derived ACCESS_CLASS type.

FUNCTION: Pop the CONTROL_STACK to determine the parent object, and create a new type compatible to the parent, with an explicit path to the parent. Write to the TYPE_STACK to reference the enclosing subprogram type, the class of the new type, and the type information of the new type. Push the new ACCESS_VAR on the CONTROL_STACK, and give it an initial value of null.

STACKS: Preconditions: Top of CONTROL_STACK contains the parent ACCESS_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one, and then an ACCESS_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR is raised if parent type is private.

INCOMPLETE

PURPOSE: Declare an incompletely defined ACCESS_CLASS type.

FUNCTION: Create a null type link of kind ACCESS_VAR, and mark its privacy and visibility as appropriate. Create a null utility subprogram that will encapsulate the type, and create a path to the subprogram. Link the type definition to a null designated access object, and mark IS_HOMOGENEOUS, IS_CONSTRAINED, and its size. Write to the

TYPE_STACK to reference the enclosing subprogram type, the class of the new type, and the type information of the new type. Push the ACCESS_VAR on the CONTROL_STACK, and give it an initial value of null.

STACKS: Postconditions: An ACCESS_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now contains descriptor for the new type.

EXCEPTIONS: None.

3.1.2. ARRAY_CLASS

In the declaration of an ARRAY_CLASS type, the only field of TYPE_OPTIONS that is relevant is BOUNDS_WITH_OBJECT, which implies that the array type is constrained or not.In addition, the TYPE_SORT of a declared ACCESS_CLASS type may be any of its possible values.

CONSTRAINED

PURPOSE: Declare a constrained ARRAY_CLASS type.

FUNCTION: Determine the location of the bounds of the parent type (whether the type is constrained or unconstrained). Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the new type will be (with the object or with the type). Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, pop the CONTROL_STACK to determine index bounds (arranged on the stack from last dimension to first), and additionally update the array information along the path of the new type. Pop the CONTROL_STACK past all of the constraint information. Write to the TYPE_STACK to reference the array information for each dimension, the enclosing subprogram, and the type information of the new type. Push the definition of a new ARRAY_CLASS object on the CONTROL_STACK, maintaining the paths created earlier.

STACKS: Preconditions: Top of CONTROL_STACK must contain a reference to the parent type. Lower CONTROL_STACK entities contain constraint information for each dimension.

Postconditions:CONTROL_STACK is popped to the start of all the constraining information. New ARRAY_CLASS object is pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR raised if index operand is private.

TYPE_ERROR is raised if a path does not exist for each dimension of the parent, or if the parent may not be further constrained. CONSTRAINT_ERROR is raised if the new type constraints are not compatible with the parent. ELABORATION_ERROR is raised if the parent type information is not complete.

CONSTRAINED_INCOMPLETE

PURPOSE: Declare a constrained incompletely defined ARRAY_CLASS type.

FUNCTION: Determine the location of the bounds of the parent type (whether the type is constrained or unconstrained). Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the new type will be (with the object or with the type). Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, pop the CONTROL_STACK to determine index bounds (arranged on the stack from last dimension to first), and additionally update the array information along the path of the new type. Pop the CONTROL_STACK past all of the constraint information. Write to the TYPE_STACK to reference the array information for each dimension, the enclosing subprogram, and the type information of the new type. Push the definition of a new ARRAY_CLASS object on the CONTROL_STACK, maintaining the paths created earlier.

STACKS: Preconditions: Top of CONTROL_STACK must contain a reference to the parent type. Lower CONTROL_STACK entities contain constraint information for each dimension.

Postconditions:CONTROL_STACK is popped to the start of all the constraining information. New ARRAY_CLASS object is pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR raised if index operand is private. TYPE_ERROR is raised if the parent may not be further constrained. CONSTRAINT_ERROR is raised if the new type constraints are not compatible with the parent.

DEFINED

PURPOSE: Declare an ARRAY_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK to determine the dimension of the new type. Create a type link for the new type, considering a descriptor for each dimension. Pop the CONTROL_STACK to determine the enclosing subprogram. Pop the CONTROL_STACK again to determine the type of the array component. Build the array information for the new type, including information for each dimension index; this operation requires that the CONTROL_STACK be popped twice for each index to obtain the range of the index. The TYPE_STACK is updated to reference the type information for each index descriptor. Pop the CONTROL_STACK down to the end of the index information. The TYPE_STACK is updated to reference the type information for the new type. Push a new ARRAY_VAR object on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a value indicating the dimension of the array. Top - 1 of the CONTROL_STACK contains a SUBPROGRAM_VAR. Top - 2 of the CONTROL_STACK contains a type defining the component type of the array. Pairs of index bounds follow, in order of the highest dimension to the lowest.

Postconditions: The CONTROL_STACK is reduced to the level below all of the index information. An ARRAY_VAR is then pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: INSTRUCTION_ERROR is raised if too many dimension are specified. TYPE_ERROR is raised if the component type of the new array is incomplete or otherwise unconstrained. OPERAND_CLASS_ERROR is raised if component type is not valid, or if the SUBPROGRAM_VAR is not valid.

DEFINED_INCOMPLETE

PURPOSE: Declare an incompletely defined ARRAY_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK to determine the dimension of the new type. Create a type link for the new type, considering a descriptor for each dimension. Pop the CONTROL_STACK to determine the enclosing subprogram. Pop the CONTROL_STACK again to determine the type of the array component. Build the array information for the new type, including information for each dimension index; this operation requires that the CONTROL_STACK be popped twice for each index to obtain the range of the index. The TYPE_STACK is updated to reference the type information for each index descriptor. Pop the CONTROL_STACK down to the end of the index information. The TYPE_STACK is updated to reference the type information for the new type. Push a new ARRAY_VAR object on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a value indicating the dimension of the array. Top - 1 of the CONTROL_STACK contains a SUBPROGRAM_VAR. Top - 2 of the CONTROL_STACK contains a type defining the component type of the array. Pairs of index bounds follow, in order of the highest dimension to the lowest.

Postconditions: The CONTROL_STACK is reduced to the level below all of the index information. An ARRAY_VAR is then pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: INSTRUCTION_ERROR is raised if too many dimension are specified. TYPE_ERROR is raised if the otherwise unconstrained. OPERAND_CLASS_ERROR is raised if component type is not valid.

DERIVED

PURPOSE: Declare a derived ARRAY_CLASS type.

FUNCTION: The bounds of the new type are inherited form the parent. Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the new type will be (with the object or with the type). Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, Trace the type path of the parent type for each dimension, and update the array information along the path of the new type. Pop the CONTROL_STACK past all of the constraint information. Write to the TYPE_STACK to reference the array information for each dimension, the enclosing subprogram, and the type information of the new type. Push the definition of a new ARRAY_CLASS object on the CONTROL_STACK, maintaining the paths created earlier.

STACKS: Preconditions: Top of CONTROL_STACK must contain a reference to the parent type.

Postconditions:CONTROL_STACK is popped below the parent type. New ARRAY_CLASS object is pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR raised if index operand is private. ELABORATION_ERROR is raised if the parent type information is not complete.

DERIVED_INCOMPLETE

PURPOSE: Declare a derived incomplete ARRAY_CLASS type.

FUNCTION: The bounds of the new type are inherited form the parent. Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the new type will be (with the object or with the type). Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, Trace the type path of the parent type for each dimension, and update the array information along the path of the new type. Pop the CONTROL_STACK past all of the constraint information. Write to the TYPE_STACK to reference the array information for each dimension, the enclosing subprogram, and the type information of the new type. Push the definition of a new ARRAY_CLASS object on the CONTROL_STACK, maintaining the paths created earlier.

STACKS: Preconditions: Top of CONTROL_STACK must contain a reference to the parent type.

Postconditions:CONTROL_STACK is popped below the parent type. New ARRAY_CLASS object is pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: CAPABILITY_ERROR raised if index operand is private.

INCOMPLETE

PURPOSE: Declare an incompletely defined ARRAY_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK to determine the dimension of the new type. Create a type link for the new type, considering a descriptor for each dimension. Pop the CONTROL_STACK to determine the enclosing subprogram. Using incomplete type paths, build the array information for the new type, including information for each dimension index. The TYPE_STACK is updated to reference the type information for each index descriptor. The TYPE_STACK is updated to reference the type information for the new type. Push a new ARRAY_VAR object on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a value indicating the dimension of the array. Top - 1 of the CONTROL_STACK contains a SUBPROGRAM_VAR.

Postconditions: The CONTROL_STACK is reduced by one. An ARRAY_VAR is then pushed on the CONTROL_STACK. TYPE_STACK now contains a descriptor for the new type.

EXCEPTIONS: INSTRUCTION_ERROR is raised if too many dimension are specified.

3.1.3. DISCRETE_CLASS

In the declaration of a DISCRETE_CLASS type, the only field of TYPE_OPTIONS that is relevant is UNSIGNED, which implies that the type represents an integer with a natural range or an enumeration type. In addition, the TYPE_SORT of a declared DISCRETE_CLASS type may only be CONSTRAINED, DEFINED, DERIVED, or INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

CONSTRAINED

PURPOSE: Declare a constrained DISCRETE_VAR type.

FUNCTION: Pop a DISCRETE_VAR from the CONTROL_STACK as the base type. Get the class information from the parent, and pop the CONTROL_STACK twice to obtain the maximum and minimum bounds. Create a new type just like the parent, using the given privacy and visibility information. Determine the size of the type, and create a type descriptor the the new type. Push the new DISCRETE_VAR on the CONTROL_STACK, and copy the type information from the parent. Write the same type information into the TYPE_STACK, plus an entry for the bounds information.

STACKS: Preconditions: Top of CONTROL_STACK must contain the base DISCRETE_VAR. Top - 1 and top - 2 of the CONTROL_STACK must contain the new bounds (upper then lower bounds).

Postconditions: Top of CONTROL_STACK is reduced by three. A DISCRETE_VAR is then pushed on top of the CONTROL_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if the base type cannot be constrained. OPERAND_CLASS_ERROR is raised if the bounds on the CONTROL_STACK are not valid. CONSTRAINT_ERROR is raised if the new constraints are not compatible with the parent type.

DEFINED

PURPOSE: Declare a DISCRETE_CLASS type.

FUNCTION: Construct a type link with privacy and visibility set as appropriate. Pop the CONTROL_STACK to reference the enclosing subprogram. Pop the CONTROL_STACK twice again to reference the maximum then minimum value of the discrete

range. In each case, the class of the bound must be valid and not private. Compute the necessary size of the discrete type, and push a DISCRETE_VAR on the CONTROL_STACK. Write to the TYPE_STACK the type information for the new type, and include a path to the bounds of the discrete type, which are also written to the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a SUBPROGRAM_VAR. Top - 1 of CONTROL_STACK must contain the maximum bounds information, followed by the minimum bounds information at top - 2.

Postconditions: Top of CONTROL_STACK is reduced by three, then a DISCRETE_VAR is pushed on the CONTROL_STACK. The TYPE_STACK is updated to include type information and bounds information.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not valid, or if a bound is not of the appropriate class. CAPABILITY_ERROR is raised if a bound is private.

DERIVED

PURPOSE: Declare a derived DISCRETE_CLASS type.

FUNCTION: Pop a DISCRETE_VAR from the CONTROL_STACK to reference the parent type. Get the class information from the parent type on the TYPE_STACK, and then create a new type derived from the parent. Push a DISCRETE_VAR on the CONTROL_STACK. Copy the type information from the parent, and write entries into the TYPE_STACK to provide type information and class information for the new type.

STACKS: Preconditions: Top of CONTROL_STACK must contain the parent DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one, and then a DISCRETE_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if a the parent type is not on the CONTROL_STACK. TYPE_ERROR is raised if the parent type is not derivable.

INCOMPLETE

PURPOSE: Declare an incompletely defined DISCRETE_VAR type.

FUNCTION: In this instance, the instruction uses the UNSIGNED field of TYPE_OPTIONS_SET. First, create a type link to a DISCRETE_VAR, using the stated privacy and visibility. Push the new DISCRETE_VAR on the CONTROL_STACK, and create a path to the enclosing subprogram. Write entries into the TYPE_STACK to provide incomplete type and bounds information.

STACKS: Postconditions: A DISCRETE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

3.1.4. FLOAT_CLASS

In the declaration of a FLOAT_CLASS type, the field TYPE_OPTIONS is not relevant, and therefore ignored. In addition, the TYPE_SORT of a declared FLOAT_CLASS may only be CONSTRAINED, DEFINED, DERIVED, or INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

CONSTRAINED

PURPOSE: Declare a constrained FLOAT_VAR type.

FUNCTION: Pop a FLOAT_VAR from the CONTROL_STACK as the base type. Get the class information from the parent, and pop the CONTROL_STACK twice to obtain the maximum and minimum bounds. Create a new type just like the parent, using the given privacy and visibility information.Push the new FLOAT_VAR on the CONTROL_STACK, and copy the type information from the parent. Write the same type information into the TYPE_STACK, plus an entry for the bounds information.

STACKS: Preconditions: Top of CONTROL_STACK must contain the base FLOAT_VAR. Top - 1 and top - 2 of the CONTROL_STACK must contain the new bounds (upper then lower bounds).

Postconditions: Top of CONTROL_STACK is reduced by three. A FLOAT_VAR is then pushed on top of the CONTROL_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if the base type cannot be constrained. OPERAND_CLASS_ERROR is raised if the bounds on the CONTROL_STACK are not valid. CONSTRAINT_ERROR is raised if the new constraints are not compatible with the parent type.

DEFINED

PURPOSE: Declare a FLOAT_CLASS type.

FUNCTION: Construct a type link with privacy and visibility set as appropriate. Pop the CONTROL_STACK to reference the enclosing subprogram. Pop the CONTROL_STACK twice again to reference the maximum then minimum value of the discrete range. In each case, the class of the bound must be valid and not private.i Compute the necessary size of the discrete type, and push a FLOAT_VAR on the CONTROL_STACK. Write to the TYPE_STACK the type information for the new type, and include a path to the bounds of the discrete type, which are also written to the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a SUBPROGRAM_VAR. Top - 1 of CONTROL_STACK must contain the maximum bounds information, followed by the minimum bounds information at top - 2.

Postconditions: Top of CONTROL_STACK is reduced by three, then a FLOAT_VAR is pushed on the CONTROL_STACK. The TYPE_STACK is updated to include type information and bounds information.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not valid, or if a bound is not of the appropriate class. CAPABILITY_ERROR is raised if a bound is private.

DERIVED

PURPOSE: Declare a derived FLOAT_CLASS type.

FUNCTION: Pop a FLOAT_VAR from the CONTROL_STACK to reference the parent type. Get the class information from the parent type on the TYPE_STACK, and then create a new type derived from the parent. Push a FLOAT_VAR on the CONTROL_STACK. Copy the type information from the parent, and write entries into the TYPE_STACK to provide type information and class information for the new type.

STACKS: Preconditions: Top of CONTROL_STACK must contain the parent FLOAT_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one, and then a FLOAT_VAR is pushed on top of the CONTROL_STACK. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if a the parent type is not on the CONTROL_STACK. TYPE_ERROR is raised if the parent type is not derivable.

INCOMPLETE

PURPOSE: Declare an incompletely defined FLOAT_VAR type.

FUNCTION: In this instance, the instruction uses the UNSIGNED field of TYPE_OPTIONS_SET. First, create a type link to a FLOAT_VAR, using the stated privacy and visibility. Push the new FLOAT_VAR on the CONTROL_STACK, and create a path to the enclosing subprogram. Write entries into the TYPE_STACK to provide incomplete type and bounds information.

STACKS: Postconditions: A FLOAT_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

3.1.5. PACKAGE_CLASS

In the declaration of a PACKAGE_CLASS type, the field TYPE_OPTIONS is not relevant, and therefore ignored. In addition, the TYPE_SORT of a declared PACKAGE_CLASS type may only be DEFINED, DERIVED, or INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

DEFINED

PURPOSE: Declare a PACKAGE_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop a segment position off the CONTROL_STACK and reference the corresponding segment name and module start. Pop a value off the CONTROL_STACK as the generic count, and then another value as the import count. If there are any imports, then allocate an import space and transfer the imports from the parent module. Otherwise, create a null import link. Write to the TYPE_STACK the type information for the new type, as well as the class information. Push the new PACKAGE_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent SUBPROGRAM_VAR, followed by a segment position value on top - 1. Top - 2 of the CONTROL_STACK contains the generic count, and top - 3 contains the import count.

Postconditions: Top of CONTROL_STACK is reduced by 4.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not

legal, or if the imports cannot be copied from an unelaborated parent. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not found.

DERIVED

PURPOSE: Declare a derived PACKAGE_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Pop a module object off the CONTROL_STACK. Get the class, type, and subprogram information from the parent. Create an entry in the TYPE_STACK indicating the same class, type, and subprogram information for a new type. Push the new PACKAGE_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent module variable.

Postconditions: Top of CONTROL_STACK is reduced by one, and then a PACKAGE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal. OPERAND_CLASS_ERROR is raised if the parent object is not found. CAPABILITY_ERROR is raised if the parent object is private. TYPE_ERROR is raised if the parent cannot be derived.

INCOMPLETE

PURPOSE: Declare an incompletely defined PACKAGE_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Create a null package utility, and write the appropriate type, class, and subprogram information to the TYPE_STACK. Push the new PACKAGE_VAR on the CONTROL_STACK.

STACKS: Postconditions: A PACKAGE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal.

3.1.6. RECORD_CLASS

In the declaration of a RECORD_CLASS type, the field TYPE_OPTIONS is not relevant, and therefore ignored. In addition, the TYPE_SORT of a declared RECORD_CLASS type may only be DEFINED, DEFINED_INCOMPLETE, DERIVED, DERIVED_INCOMPLETE, and INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

DEFINED

PURPOSE: Declare a RECORD_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK indicating the number of fields in the record. Create a type link for the record type. Pop a SUBPROGRAM_VAR off the CONTROL_STACK referencing the enclosing subprogram. For each record component, record the type information of each field into the TYPE_STACK, and then pop the CONTROL_STACK to the end of the field information. Write an entry to the TYPE_STACK indicating the type information for the new type, and then push a corresponding RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a value indicating the number of fields in the record. A parent SUBPROGRAM_VAR is next, followed N entities on the CONTROL_STACK with the type information for each field.

Postconditions: The CONTROL_STACK is popped to the end of the field information. A RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: INSTRUCTION_ERROR raised if the field count is not valid. TYPE_ERROR is raised if there exists and invalid component type.

DEFINED_INCOMPLETE

PURPOSE: Declare an incompletely defined RECORD_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK indicating the number of fields in the record. Create a type link for the record type. Pop a SUBPROGRAM_VAR off the CONTROL_STACK referencing the enclosing subprogram. For each record component that is complete, record the type information of each field into the TYPE_STACK, and then pop the CONTROL_STACK to the end of the field information. Write an entry to the TYPE_STACK indicating the type information for the new type, and then push a corresponding RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a value indicating the number of fields in the record. A parent SUBPROGRAM_VAR is next, followed entities on the CONTROL_STACK with the type information for each field that is complete.

Postconditions: The CONTROL_STACK is popped to the end of the field information. A RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: INSTRUCTION_ERROR raised if the field count is not valid. TYPE_ERROR is raised if there exists and invalid component type.

DERIVED

PURPOSE: Declare a derived RECORD_CLASS type.

FUNCTION: Pop a RECORD_VAR of the CONTROL_STACK as the parent type. Create a new type identical to the parent and mark a path from the parent to the child on the TYPE_STACK. Pop the CONTROL_STACK to reference the enclosing SUBPROGRAM_VAR. Write entries into the TYPE_STACK indicating the type information of the new type. Copy the type information of the fields for the parent to the child. Push a new RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain the parent RECORD_VAR, followed by the enclosing SUBPROGRAM_VAR.

Postconditions: Top of CONTROL_STACK is reduced by two. a RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR is raised if the parent cannot be derived.

DERIVED_INCOMPLETE

PURPOSE: Declare a derived incomplete RECORD_CLASS type.

FUNCTION: Pop a RECORD_VAR of the CONTROL_STACK as the parent type. Create a new type identical to the parent and mark a path from the parent to the child on the TYPE_STACK. Pop the CONTROL_STACK to reference the enclosing SUBPROGRAM_VAR. Write entries into the TYPE_STACK indicating the type information of the new type. Copy the type information of

the complete fields for the parent to the child. Push a new RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain the parent RECORD_VAR, followed by the enclosing SUBPROGRAM_VAR.

Postconditions: Top of CONTROL_STACK is reduced by two. a RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR is raised if the parent cannot be derived.

INCOMPLETE

PURPOSE: Declare an incompletely defined RECORD_CLASS type.

FUNCTION: Pop a value from the CONTROL_STACK indicating the number of fields in the record. Create a type link for the record type. Pop the CONTROL_STACK to reference the enclosing subprogram. Create an empty descriptor for the new type, and write the appropriate type information for each field. Push a RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains a value indicating the number of fields. Top - 1 of the CONTROL_STACK contains a SUBPROGRAM_VAR.

Postconditions: Top of CONTROL_STACK is reduced by 2. A RECORD_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

3.1.7. SEGMENT_CLASS

In the declaration of a SEGMENT_CLASS type, the fields TYPE_OPTIONS and TYPE_SORT are not relevant, and therefore ignored.

PURPOSE: Declare a SEGMENT_CLASS type.

FUNCTION: Create a site on the TYPE_STACK for the type information of an empty code segment descriptor. Push a SEGMENT_VAR referencing this type information on top of the CONTROL_STACK.

STACKS: A SEGMENT_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

3.1.8. TASK_CLASS

In the declaration of a TASK_CLASS type, the only field of TYPE_OPTIONS that has meaning is HAS_ENTRIES, which indicates if the task is an actor task or not. In addition, the TYPE_SORT of a declared TASK_CLASS type may only be DEFINED, DERIVED, or INCOMPLETE. Any other combination raises the exception INSTRUCTION_ERROR.

DEFINED

PURPOSE: Declare a TASK_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop a segment position off the CONTROL_STACK and reference the corresponding segment name and module start. Pop a value off the CONTROL_STACK as the generic count, and then another value as the import count. If there are any imports, then allocate an import space and transfer the imports from the parent module. Otherwise, create a null import link. Write to the TYPE_STACK the type information for the new type, as well as the class information. Push the new TASK_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent SUBPROGRAM_VAR, followed by a segment position value on top - 1. Top - 2 of the CONTROL_STACK contains the generic count, and top - 3 contains the import count.

Postconditions: Top of CONTROL_STACK is reduced by 4.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal, or if the imports cannot be copied from an unelaborated parent. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not found.

DERIVED

PURPOSE: Declare a derived TASK_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Pop a module object off the CONTROL_STACK. Get the class, type, and subprogram information from the parent. Create an entry in the TYPE_STACK indicating the same class, type, and subprogram information for a new type. Push the new TASK_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent module variable.

Postconditions: Top of CONTROL_STACK is reduced by one, and then a TASK_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal. OPERAND_CLASS_ERROR is raised if the parent object is not found. CAPABILITY_ERROR is raised if the parent object is private. TYPE_ERROR is raised if the parent cannot be derived.

INCOMPLETE

PURPOSE: Declare an incompletely defined TASK_CLASS type.

FUNCTION: Determine if the declaration is legal, then create an empty module type and type path. Create a null package utility, and write the appropriate type, class, and subprogram information to the TYPE_STACK. Push the new TASK_VAR on the CONTROL_STACK.

STACKS: Postconditions: A TASK_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal.

3.1.9. VARIANT_RECORD_CLASS

In the case of a VARIANT_RECORD_CLASS type, the only fields of TYPE_OPTIONS that are relevant are DERIVED_PRIVATE and CONSTRAINED. In addition, all values of TYPE_SORT are meaningful when applied to the declaration of a VARIANT_RECORD_CLASS_TYPE.

CONSTRAINED

PURPOSE: Declare a constrained VARIANT_RECORD_CLASS type.

FUNCTION: Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. Get the class information from the parent. Create a new type identical to the base type. For each discriminant, pop a value from the CONTROL_STACK, and constrain each field in turn. In the process, type information for each constrained discriminant and field is written into the TYPE_STACK. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the new type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains a base VARIANT_RECORD_VAR, followed by a constraint value for each field in turn.

Postconditions: Top of the CONTROL_STACK is reduced to below the constraint information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR raised if the base type cannot be constrained. ELABORATION_ERROR is raised if the base type is not complete. INSTRUCTION_ERROR is raised if the number of constraints is not sufficient. CONSTRAINT_ERROR is raised if a constraint violated any base field. CAPABILITY_ERROR is raised if an attempt is made to constrain a private field.

CONSTRAINED_INCOMPLETE

PURPOSE: Declare a constrained incompletely defined VARIANT_RECORD_CLASS type.

FUNCTION: Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. Get the class information from the parent. Create a new type identical to the base type. For each discriminant, pop a value from the CONTROL_STACK, and constrain each complete field in turn. In the process, type information for each constrained discriminant and field is written into the TYPE_STACK. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the new type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains a base VARIANT_RECORD_VAR, followed by a constraint value for each complete field in turn.

Postconditions: Top of the CONTROL_STACK is reduced to below the constraint information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR raised if the base type cannot be constrained. ELABORATION_ERROR is raised if the base type is not complete. INSTRUCTION_ERROR is raised if the number of constraints is not sufficient. CONSTRAINT_ERROR is raised if a constraint violated any base field. CAPABILITY_ERROR is raised if an attempt is made to constrain a private field.

DEFINED

PURPOSE: Declare a VARIANT_RECORD_CLASS type.

FUNCTION: Pop a value off the CONTROL_STACK indicating the number of fields. Pop a value off the CONTROL_STACK indicating the number of discriminant fields. Allocate space on the TYPE_STACK for the definition of the type, and create a type link. Pop the SUBPROGRAM_VAR indicating the enclosing subprogram, and then pop a value indicating the number of variant parts. Write the type information for all of the fixed fields, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicate the new field type information. Next, all of the variant fields are written, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicated the new field type information. The TYPE_STACK is written to include the class information and the type information for the basic type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK as the new type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a value denoting the number of fields. Top - 1 contains a value indicating the number of discriminant fields. Top - 2 is a SUBPROGRAM_VAR indicating the enclosing subprogram. Top - 3 is a value indicating the number of variant parts. The next set of objects defined the types and initialization of the fixed fields, followed by a set of objects for the variant fields.

Postconditions: Top of the CONTROL_STACK is reduced to the end of all the field information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the number of discriminates or fields in not valid.

DEFINED_INCOMPLETE

PURPOSE: Declare a defined incomplete VARIANT_RECORD_CLASS type.

FUNCTION: Pop a value off the CONTROL_STACK indicating the number of fields. Pop a value off the CONTROL_STACK indicating the number of discriminant fields. Allocate space on the TYPE_STACK for the definition of the type, and create a type link. Pop the SUBPROGRAM_VAR indicating the enclosing subprogram, and then pop a value indicating the number of

variant parts. Write the type information for all of the complete fixed fields, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicate the new field type information. Next, all of the complete variant fields are written, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicated the new field type information. The TYPE_STACK is written to include the class information and the type information for the basic type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK as the new type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a value denoting the number of fields. Top - 1 contains a value indicating the number of discriminant fields. Top - 2 is a SUBPROGRAM_VAR indicating the enclosing subprogram. Top - 3 is a value indicating the number of variant parts. The next set of objects defined the types and initialization of the complete fixed fields, followed by a set of objects for the complete variant fields.

Postconditions: Top of the CONTROL_STACK is reduced to the end of all the field information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the number of discriminates or fields in not valid.

DERIVED

PURPOSE: Declare a DERIVED VARIANT_RECORD_CLASS type.

FUNCTION: Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. Get the class information from the parent. Create a new type identical to the base type. The type information for each field and discriminant from the parent is copied to the new type. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the new type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains a base VARIANT_RECORD_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by one. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR raised if the base type cannot be derived. ELABORATION_ERROR is raised if the base type is not complete. CAPABILITY_ERROR is raised if an attempt is made to derive a private field.

DERIVED_INCOMPLETE

PURPOSE: Declare a derived incompletely defined VARIANT_RECORD_CLASS type.

FUNCTION: Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. Get the class information from the parent. Create a new type identical to the base type. For each complete discriminant, copy the corresponding type information from the parent. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the new type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains a base VARIANT_RECORD_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by one. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK is updated to include a descriptor for the new type.

EXCEPTIONS: TYPE_ERROR raised if the base type cannot be derived. ELABORATION_ERROR is raised if the base type is not complete. CAPABILITY_ERROR is raised if an attempt is made to derive a private field.

INCOMPLETE

PURPOSE: Declare a VARIANT_RECORD_CLASS type.

FUNCTION: Pop a value off the CONTROL_STACK indicating the number of fields. Pop a value off the CONTROL_STACK indicating the number of discriminant fields. Allocate space on the TYPE_STACK for the definition of the type, and create a type link. Create an empty type descriptor for the new type, including the types of the discriminant which are found in turn on the CONTROL_STACK. The TYPE_STACK is written to include the class information and the type information for the basic type. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK as the new type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a

value denoting the number of fields. Top - 1 contains a value indicating the number of discriminant fields.

Postconditions: Top of the CONTROL_STACK is reduced to the end of all the discriminant information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the number of discriminates or fields in not valid.

3.2. COMPLETE_TYPE

The COMPLETE_TYPE instruction finishes the descriptor of an incompletely specified type.

COMPLETE_TYPE takes the form:

```
type COMPLETE_TYPE_INSTRUCTION is
  record
    COMPLETION_CLASS : OPERAND_CLASS;
    COMPLETION_MODE  : TYPE_COMPLETION_MODE;
  end record;
```

While the COMPLETION_CLASS (of type OPERAND_CLASS) names the class of the target type that is to be completed, a value of TYPE_COMPLETION_MODE further defines the means through which the completion is to be achieved. formally, the TYPE_COMPLETION_MODE is defined as:

```
type TYPE_COMPLETION_MODE is
       (BY_COMPONENT_COMPLETION, BY_CONSTRAINING,
        BY_DEFINING,             BY_DERIVING);
```

COMPLETE_TYPE is appropriate only for objects of class:

* ACCESS_CLASS
* ARRAY_CLASS
* DISCRETE_CLASS
* FLOAT_CLASS
* PACKAGE_CLASS
* RECORD_CLASS
* TASK_CLASS
* VARIANT_RECORD_CLASS

The use of any other COMPLETION_CLASS value will raise the exception INSTRUCTION_ERROR. In the following sections, we will treat each class in detail.

3.2.1. ACCESS_CLASS

In the completion of an ACCESS_CLASS type, the COMPLETION_MODE may only be BY_CONSTRAINING, BY_DEFINING, or BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_CONSTRAINING

PURPOSE: Complete an ACCESS_CLASS type by constraining.

FUNCTION: Pop an incomplete ACCESS_VAR from the top of the CONTROL_STACK. Pop the parent ACCESS_VAR off the CONTROL_STACK. Reference the class information of the parent. Finish the derivation of the child type based on the type information of the parent. Pop the CONTROL_STACK for the constraining ACCESS_INFO. Pop a SUBPROGRAM_VAR from the CONTROL_STACK and set the path to the enclosing subprogram. Update the TYPE_STACK to include the new class and type information for the newly completed type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incompletely specified ACCESS_VAR. Top - 1 contains the parent ACCESS_VAR. Top - 2 of the CONTROL_STACK contains the constraining ACCESS_INFO. Top - 3 contains the parent SUBPROGRAM_VAR.

Postconditions: CONTROL_STACK is popped to below the SUBPROGRAM_VAR. The TYPE_STACK is updated to include the completed type information for the ACCESS_VAR.

EXCEPTIONS: CAPABILITY_ERROR raised if the parent type is private. OPERAND_CLASS_ERROR is raised if SUBPROGRAM_VAR is invalid. TYPE_ERROR is raised if the incomplete type is not valid.

BY_DEFINING

PURPOSE: Complete an ACCESS_CLASS type by defining.

FUNCTION: Pop an incomplete ACCESS_VAR off the top of the CONTROL_STACK. Pop the CONTROL_STACK to reference a SUBPROGRAM_VAR (which must be FOR_UTILITY) that encapsulates the new type, and create a path to the subprogram. Pop the CONTROL_STACK again to determine the type of the designated access objects, and determine if the designated object IS_HOMOGENEOUS, IS_CONSTRAINED, and its size. Pop a value from the CONTROL_STACK to determine a page count, and then create a collection for the designated objects. An explicit path is created from the type definition to the collection. Write to the TYPE_STACK to

reference the enclosing subprogram type, the class of the new type, and the type information of the new type.

STACKS: Preconditions: Top of CONTROL_STACK contains a SUBPROGRAM_VAR. Top - 1 of CONTROL_STACK contains a classed object identifying the type of the designated access objects. Top - 2 of CONTROL_STACK contains a resource value.

Postconditions: The CONTROL_STACK is popped to below the resource value. TYPE_STACK now includes a descriptor for the new type.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if SUBPROGRAM_VAR is invalid. TYPE_ERROR is raised if the incomplete type is not valid.

BY_DERIVING

PURPOSE: Complete an ACCESS_CLASS type by deriving.

FUNCTION: Pop an incomplete ACCESS_VAR from the top of the CONTROL_STACK. Pop the parent ACCESS_VAR off the CONTROL_STACK. Reference the class information of the parent. Finish the derivation of the child type based on the type information of the parent. Pop a SUBPROGRAM_VAR from the CONTROL_STACK and set the path to the enclosing subprogram. Update the TYPE_STACK to include the new class and type information for the newly completed type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incompletely specified ACCESS_VAR. Top - 1 contains the parent ACCESS_VAR. Top - 2 contains the parent SUBPROGRAM_VAR.

Postconditions: CONTROL_STACK is popped to below the SUBPROGRAM_VAR. The TYPE_STACK is updated to include the completed type information for the ACCESS_VAR.

EXCEPTIONS: CAPABILITY_ERROR raised if the parent type is private. OPERAND_CLASS_ERROR is raised if SUBPROGRAM_VAR is invalid. TYPE_ERROR is raised if the incomplete type is not valid.

3.2.2. ARRAY_CLASS

In the completion of an ACCESS_CLASS type, all values of COMPLETION_MODE are appropriate.

BY_COMPONENT_COMPLETION

PURPOSE: Complete an ARRAY_CLASS type by component completion.

FUNCTION: Pop the incomplete type information off the top of the CONTROL_STACK. Get the array and type information from the TYPE_STACK for the incomplete type, and determine the size of the array items after tracing the TYPE_STACK for the completed component information. For each dimension of the array, trace the index descriptor and update the array information on the TYPE_STACK. Set the type information for the array on the TYPE_STACK also.

STACKS: Preconditions: Top of the CONTROL_STACK contains the incompletely specified ARRAY_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by one.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the component information is out of scope.

BY_CONSTRAINING

PURPOSE: Complete an ARRAY_CLASS type by constraining.

FUNCTION: Pop the incomplete type information off the top of the CONTROL_STACK. Determine if the type is already constrained or not. Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the completed type will be. Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, trace the type path of the parent for each dimension, and update the array information along the path of the new type. Pop the CONTROL_STACK past all of the constraint information. Write to the TYPE_STACK to reference the array information for each dimension, the enclosing subprogram, and the type information of the new type.

STACKS: Preconditions: Top of CONTROL_STACK must contain the incomplete type definition. Top - 1 of the CONTROL_STACK must contain a reference to the parent type. Constraint information for each dimension follows on the CONTROL_STACK.

Postconditions: CONTROL_STACK is popped below the constraint information.

EXCEPTIONS: CAPABILITY_ERROR is raised if the index operand is private. ELABORATION_ERROR is raised if the parent type information is not complete.

BY_DEFINING

PURPOSE: Complete an ARRAY_CLASS type by defining.

FUNCTION: Pop the CONTROL_STACK to reference the incomplete type information. Pop the CONTROL_STACK again to determine the enclosing subprogram. Pop the CONTROL_STACK again to determine the type of the array component. Build the array information for the new type, including information for each dimension index; this operation requires that the CONTROL_STACK be popped twice for each index to obtain the range of the index. The TYPE_STACK is updated to reference the type information for each index descriptor. Pop the CONTROL_STACK down to the end of the index information. The TYPE_STACK is updated to reference the type information for he new type.

STACKS: Preconditions: Top of CONTROL_STACK contains the incompletely specified type information. Top - 1 of CONTROL_STACK contains a SUBPROGRAM_VAR. Top - 2 of the CONTROL_STACK contains a type defining the component type of the array. Pairs of index bounds follow, in order of the highest dimension to the lowest.

Postconditions: The CONTROL_STACK is reduced to the level below all of the index information. TYPE_STACK now contains a descriptor for the completed type.

EXCEPTIONS: INSTRUCTION_ERROR is raised if too many dimensions are specified. TYPE_ERROR is raised if the component class of the new array is incomplete or otherwise unconstrained. OPERAND_CLASS_ERROR is raised if the component type is not valid or if the SUBPROGRAM_VAR is not valid.

BY_DERIVING

PURPOSE: Complete an ARRAY_CLASS type by deriving.

FUNCTION: Pop the incomplete type information off the top of the CONTROL_STACK. Determine if the type is already constrained or not. Pop the CONTROL_STACK to reference the indexed operand, and create a path to the parent type. Identify where the bounds of the completed type will be. Next, examine the parent type and determine the dimensionality of the child. Create a type path to the array information of each dimension. For each index, trace the type path of the parent for each dimension, and update the array information along the path of the new type. Write to the TYPE_STACK to reference the array information for each dimension, the

enclosing subprogram, and the type information of the new type.

STACKS: Preconditions: Top of CONTROL_STACK must contain the incomplete type definition. Top - 1 of the CONTROL_STACK must contain a reference to the parent type.

Postconditions: CONTROL_STACK is reduced by one.

EXCEPTIONS: CAPABILITY_ERROR is raised if the index operand is private. ELABORATION_ERROR is raised if the parent type information is not complete.

3.2.3. DISCRETE_CLASS

In the completion of a DISCRETE_CLASS type, the only values of COMPLETION_MODE that are relevant are BY_CONSTRAINING, BY_DEFINING, and BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_CONSTRAINING

PURPOSE: Complete a DISCRETE_CLASS type by constraining.

FUNCTION: Pop the incomplete DISCRETE_VAR off the top of the CONTROL_STACK. Get the class information from the parent type, and then pop the CONTROL_STACK twice to obtain the constrained bounds (maximum then minimum bounds). Compute the size of the completed type, and copy the type information and bounds information to the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains the incomplete DISCRETE_VAR. Top - 1 contains the maximum constraint. Top - 2 of the CONTROL_STACK contains the minimum constraint.

Postconditions: Top of CONTROL_STACK is reduced to below the constraint information. The TYPE_STACK is updated to reflect the completed type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid or cannot be constrained. CAPABILITY_ERROR is raised if the incomplete type is out of scope. CONSTRAINT_ERROR is raised if the constraints to not satisfy the parent. ELABORATION_ERROR is raised if the bounds are not complete.

BY_DEFINING

PURPOSE: Complete a DISCRETE_CLASS type by defining.

FUNCTION: Pop the incomplete DISCRETE_VAR off the top of the CONTROL_STACK. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop the CONTROL_STACK twice more to obtain the upper then lower bounds of the completed type. Determine the size of the type, then update the TYPE_STACK to reflect the type and bounds information.

STACKS: Preconditions: Top of CONTROL_STACK contains an incomplete DISCRETE_VAR. Top - 1 contains a SUBPROGRAM_VAR. Top - 2 contains the maximum bounds, and top - 3 of the CONTROL_STACK contains the minimum bounds.

Postconditions: The CONTROL_STACK is popped to below the bounds information. The TYPE_STACK is updated to reflect the new type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not valid. CAPABILITY_ERROR is raised if a bound is private. ELABORATION_ERROR is raised if the parent and incomplete type are not both signed or unsigned.

BY_DERIVING

PURPOSE: Complete a DISCRETE_CLASS type by deriving.

FUNCTION: Pop the incomplete DISCRETE_VAR off the top of the CONTROL_STACK. Pop the parent DISCRETE_VAR off the CONTROL_STACK. Copy the type information from the parent, including a determination of the size of the type. Write the completed class and type information in the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains the incomplete DISCRETE_VAR. Top - 1 contains the parent DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced by two.

EXCEPTIONS: TYPE_ERROR is raised if the parent type cannot be derived. ELABORATION_ERROR is raised if the parent and incomplete type are not both unsigned or signed.

3.2.4. FLOAT_CLASS

In the completion of a FLOAT_CLASS type, the only values of COMPLETION_MODE that are relevant are BY_CONSTRAINING, BY_DEFINING, and BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_CONSTRAINING

PURPOSE: Complete a FLOAT_CLASS type by constraining.

FUNCTION: Pop the incomplete FLOAT_VAR off the top of the CONTROL_STACK. Get the class information from the parent type, and then pop the CONTROL_STACK twice to obtain the constrained bounds (maximum then minimum bounds). Compute the size of the completed type, and copy the type information and bounds information to the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains the incomplete FLOAT_VAR. Top - 1 contains the maximum constraint. Top - 2 of the CONTROL_STACK contains the minimum constraint.

Postconditions: Top of CONTROL_STACK is reduced to below the constraint information. The TYPE_STACK is updated to reflect the completed type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid or cannot be constrained. CAPABILITY_ERROR is raised if the incomplete type is out of scope. CONSTRAINT_ERROR is raised if the constraints to not satisfy the parent.

BY_DEFINING

PURPOSE: Complete a FLOAT_CLASS type by defining.

FUNCTION: Pop the incomplete FLOAT_VAR off the top of the CONTROL_STACK. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop the CONTROL_STACK twice more to obtain the upper then lower bounds of the completed type. Determine the size of the type, then update the TYPE_STACK to reflect the type and bounds information.

STACKS: Preconditions: Top of CONTROL_STACK contains an incomplete FLOAT_VAR. Top - 1 contains a SUBPROGRAM_VAR. Top - 2 contains the maximum bounds, and top - 3 of the CONTROL_STACK contains the minimum bounds.

Postconditions: The CONTROL_STACK is popped to below the bounds information. The TYPE_STACK is updated to reflect the new type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not valid. CAPABILITY_ERROR is raised if a bound is private.

BY_DERIVING

PURPOSE: Complete a FLOAT_CLASS type by deriving.

FUNCTION: Pop the incomplete FLOAT_VAR off the top of the CONTROL_STACK. Pop the parent FLOAT_VAR off the CONTROL_STACK. Copy the type information from the parent, including a determination of the size of the type. Write the completed class and type information in the TYPE_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains the incomplete FLOAT_VAR. Top - 1 contains the parent FLOAT_VAR.

Postconditions: Top of CONTROL_STACK is reduced by two.

EXCEPTIONS: TYPE_ERROR is raised if the parent type cannot be derived. ELABORATION_ERROR is raised if the parent and incomplete type are not both unsigned or signed.

3.2.5. PACKAGE_CLASS

In the completion of a PACKAGE_CLASS type, the only COMPLETION_MODE values that are appropriate are BY_DEFINING and BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_DEFINING

PURPOSE: Complete a PACKAGE_CLASS type by defining.

FUNCTION: Pop the incomplete PACKAGE_VAR off the top of the CONTROL_STACK. Trace the class information of the type. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop a segment position off the CONTROL_STACK and reference the corresponding segment name and module start. Pop a value off the CONTROL_STACK as the generic count, and then another value as the import count. If there are any imports, then allocate an import space and transfer the imports from the parent module. Otherwise, create a null import link. Write to the TYPE_STACK the type information for the new type, as well as the class information.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incomplete PACKAGE_VAR. Top - 1 must contain the parent SUBPROGRAM_VAR, followed by a segment position on top - 2. Top - 3 of the CONTROL_STACK contains the generic count, and top - 4 contains the import count.

Postconditions: Top of CONTROL_STACK is reduced by 5. The TYPE_STACK is updated to reflect the new type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid.

CAPABILITY_ERROR is raised if the incomplete type is private. ELABORATION_ERROR is raised if the declaration is not legal, or if the imports cannot be copied from an unelaborated parent. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not found.

BY_DERIVING

PURPOSE: Complete a PACKAGE_CLASS type by deriving.

FUNCTION: Pop an incomplete PACKAGE_VAR off the top of the CONTROL_STACK. Pop a module object off the CONTROL_STACK. Get the class, type and subprogram information from the parent. Create an entry in the TYPE_STACK indicating the same class, type, and subprogram information for the completed type.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent module variable.

Postconditions: Top of CONTROL_STACK is reduced by one. TYPE_STACK is updated to reflect completed type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid, or if the parent cannot be derived. CAPABILITY_ERROR is raised if the incomplete type is private, or if the parent module is private. OPERAND_CLASS_ERROR is raised if the parent module is not found.

3.2.6. RECORD_CLASS

In the completion of a RECORD_CLASS type, the only COMPLETION_MODE values that are appropriate are BY_COMPONENT_COMPLETION, BY_DEFINING, and BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_COMPONENT_COMPLETION

PURPOSE: Complete a RECORD_CLASS type by component completion.

FUNCTION: Pop the incomplete RECORD_VAR off the top of the CONTROL_STACK. For each field in the type, trace the type path to the field definition, and update the parent type field information. Determine the size of the type, and update the type information for the now completed type in the TYPE_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK contains the incomplete RECORD_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the incomplete type is private.

BY_DEFINING

PURPOSE: Complete a RECORD_CLASS type by defining.

FUNCTION: Pop the incomplete RECORD_VAR from the CONTROL_STACK. Pop a SUBPROGRAM_VAR off the CONTROL_STACK referencing the enclosing subprogram. for each record component, record the type information for each filed into the TYPE_STACK, and then pop the CONTROL_STACK to the end of the field information. Write an entry to the TYPE_STACK indicating the type information for the new type.

STACKS: Preconditions: Top of CONTROL_STACK must contain an incomplete RECORD_VAR. A parent SUBPROGRAM_VAR is next, followed by N entries on the CONTROL_STACK with the type information for each field.

Postconditions: The CONTROL_STACK is popped to the end of the field information. The TYPE_STACK is updated to include a complete descriptor for the new type.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type in not valid or if there exists an invalid component type. CAPABILITY_ERROR is raised if the incomplete type is private.

BY_DERIVING

PURPOSE: Complete a RECORD_CLASS type by deriving.

FUNCTION: Pop an incomplete RECORD_VAR off the top of the CONTROL_STACK. Pop another RECORD_VAR as the parent type. Update the incomplete type to match the parent type information, and build a type path from the parent to the child on the TYPE_STACK. Pop the CONTROL_STACK to reference the enclosing SUBPROGRAM_VAR. Write the completed type information and field information into the TYPE_STACK for the completed type.

STACKS: Preconditions: Top of CONTROL_STACK must contain an incomplete RECORD_VAR. Top - 1 must contain the parent RECORD_VAR, followed by the enclosing SUBPROGRAM_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by three. The TYPE_STACK is updated to include a complete descriptor for the type.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid, or if the parent cannot be derived. CAPABILITY_ERROR is raised if the incomplete type is private.

3.2.7. TASK_CLASS

In the completion of a TASK_CLASS type, the only COMPLETION_MODE values that are appropriate are BY_DEFINING and BY_DERIVING. Any other combination raises the exception INSTRUCTION_ERROR.

BY_DEFINING

PURPOSE: Complete a TASK_CLASS type by defining.

FUNCTION: Pop the incomplete TASK_VAR off the top of the CONTROL_STACK. Trace the class information of the type. Pop a SUBPROGRAM_VAR off the CONTROL_STACK to reference the enclosing subprogram. Pop a segment position off the CONTROL_STACK and reference the corresponding segment name and module start. Pop a value off the CONTROL_STACK as the generic count, and then another value as the import count. If there are any imports, then allocate an import space and transfer the imports from the parent module. Otherwise, create a null import link. Write to the TYPE_STACK the type information for the new type, as well as the class information.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incomplete TASK_VAR. Top - 1 must contain the parent SUBPROGRAM_VAR, followed by a segment position on top - 2. Top - 3 of the CONTROL_STACK contains the generic count, and top - 4 contains the import count.

Postconditions: Top of CONTROL_STACK is reduced by 5. The TYPE_STACK is updated to reflect the new type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the incomplete type is private. ELABORATION_ERROR is raised if the declaration is not legal, or if the imports cannot be copied from an unelaborated parent. OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_VAR is not found.

BY_DERIVING

PURPOSE: Complete a TASK_CLASS type by deriving.

FUNCTION: Pop an incomplete TASK_VAR off the top of the CONTROL_STACK. Pop a module object off the CONTROL_STACK. Get the class, type and subprogram information from the parent. Create an entry in the TYPE_STACK indicating the same class, type, and subprogram information for the completed type.

STACKS: Preconditions: Top of CONTROL_STACK must contain a parent module variable.

Postconditions: Top of CONTROL_STACK is reduced by one. TYPE_STACK is updated to reflect completed type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid, or if the parent cannot be derived. CAPABILITY_ERROR is raised if the incomplete type is private, or if the parent module is private. OPERAND_CLASS_ERROR is raised if the parent module is not found.

3.2.8. VARIANT_RECORD_CLASS

In the completion of a VARIANT_RECORD_CLASS type, every value of COMPLETION_MODE is valid.

BY_COMPONENT_COMPLETION

PURPOSE: Complete a VARIANT_RECORD_CLASS type by component completion.

FUNCTION: Pop an incomplete VARIANT_RECORD_VAR off the CONTROL_STACK. Get the class information from the type. For each fixed field, trace the type paths and update the TYPE_STACK to reflect the completed type information. If there are any variant parts, do the same for the variant fields. Write entries into the TYPE_STACK to reflect the completed type information and class information.

STACKS: Preconditions: Top of the CONTROL_STACK contains an incomplete VARIANT_RECORD_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by one. The TYPE_STACK is updated to reflect the new type information.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the incomplete type is private.

BY_CONSTRAINING

PURPOSE: Complete a VARIANT_RECORD_CLASS type by constraining.

FUNCTION: Pop an incomplete VARIANT_RECORD_VAR from the CONTROL_STACK.. Determine if the type derives privacy, and trace the class information of the incomplete type. Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. For each discriminant, pop a value from the CONTROL_STACK, and constrain each field in turn. In the process, type information for each constrained discriminant and field is written into the TYPE_STACK. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the completed type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incomplete VARIANT_RECORD_VAR, followed by the parent VARIANT_RECORD_VAR. A constraint value for each field is next on the CONTROL_STACK.

Postconditions:Top of the CONTROL_STACK is reduced below the constraint information. The TYPE_STACK is updated to include a complete descriptor for the new type.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid, or if the parent type cannot be constrained. CAPABILITY_ERROR is raised if an attempt is made to constrain a private field, or if the incomplete type is private. ELABORATION_ERROR is raised if the parent type is not complete. INSTRUCTION_ERROR is raised if the number of constraints is not sufficient. CONSTRAINT_ERROR is raised if a constraint violated any parent field.

BY_DEFINING

PURPOSE: Complete a VARIANT_RECORD_CLASS type by defining.

FUNCTION: Pop an incomplete VARIANT_RECORD_VAR off the CONTROL_STACK. Pop the SUBPROGRAM_VAR indicating the enclosing subprogram, and then pop a value indicating the number of variant parts. Write the type information for all of the fixed fields, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicate the new field type information. Next, all of the variant fields are written, which requires that the CONTROL_STACK be popped for each field type, including initialization. The TYPE_STACK is also updated to indicate the new field type information. The TYPE_STACK is

also updated to include the class information and the type information for the basic type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an incomplete VARIANT_RECORD_VAR. Top - 1 contains a SUBPROGRAM_VAR indicating the enclosing subprogram. Top - 2 is a value indicating the number of variant parts. The next set of objects define the types and initialization of the fixed fields, followed by a set of objects for the variant fields.

Postconditions: Top of the CONTROL_STACK is reduced to the end of all the field information.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the number of discriminants or fields in not valid. TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the incomplete type is private.

BY_DERIVING

PURPOSE: Complete a VARIANT_RECORD_CLASS type by deriving.

FUNCTION: Pop an incomplete VARIANT_RECORD_VAR from the CONTROL_STACK.. Determine if the type derives privacy, and trace the class information of the incomplete type. Pop a VARIANT_RECORD_VAR from the CONTROL_STACK as the base type. Cop the parent type information for each constrained discriminant and field into the TYPE_STACK. A path from the parent to the child is created. Values are written into the TYPE_STACK to indicate the class and type information for the completed type.

STACKS: Preconditions: Top of the CONTROL_STACK must contain the incomplete VARIANT_RECORD_VAR, followed by the parent VARIANT_RECORD_VAR.

Postconditions:Top of the CONTROL_STACK is reduced by two. TYPE_STACK is updated to include a complete descriptor for the new type.

EXCEPTIONS: TYPE_ERROR is raised if the incomplete type is not valid. CAPABILITY_ERROR is raised if the incomplete type is private. ELABORATION_ERROR is raised if the parent type is not complete.

3.3. DECLARE_VARIABLE

The DECLARE_VARIABLE instruction creates an object of a given type.

DECLARE_VARIABLE takes the form

```
type DECLARE_VARIABLE_INSTRUCTION is
  record
    VARIABLE_CLASS      : OPERAND_CLASS;
    VARIABLE_OPTIONS    : VARIABLE_OPTION_SET;
    VARIABLE_VISIBILITY : VISIBILITY;
  end record;
```

While the VARIABLE_CLASS (of type OPERAND_CLASS) names the class of the target object to be created, a value of VARIABLE_OPTION_SET further defines the kind of object to be declared. Formally, VARIABLE_OPTION_SET is defined as:

```
type VARIABLE_OPTION_SET is
       record
         BY_ALLOCATION   : BOOLEAN;
         DATA_TASK       : BOOLEAN;
         DISTRIBUTOR     : BOOLEAN;
         DUPLICATE       : BOOLEAN;
         HEAP_TASK       : BOOLEAN;
         UNCHECKED       : BOOLEAN;
         WITH_CONSTRAINT : BOOLEAN;
         WITH_SUBTYPE    : BOOLEAN;
         WITH_VALUE      : BOOLEAN;
       end record;
```

As we will discuss, fields of VARIABLE_OPTION_SET are relevant only to specific classes of objects.

A value of VISIBILITY defines, obviously, the visibility of the declared object. Formally, VISIBILITY is defined as:

```
type VISIBILITY is (DEFAULT, IS_HIDDEN, IS_VISIBLE);
```

DECLARE_VARIABLE is appropriate for only objects of class:

* ACCESS_CLASS
* ANY_CLASS
* ARRAY_CLASS
* DISCRETE_CLASS

* ENTRY_CLASS
* FAMILY_CLASS
* FLOAT_CLASS
* MATRIX_CLASS
* PACKAGE_CLASS
* RECORD_CLASS
* SEGMENT_CLASS
* SELECT_CLASS
* TASK_CLASS
* VARIANT_RECORD_CLASS
* VECTOR_CLASS

The use of any other VARIABLE_CLASS will raise the exception INSTRUCTION_ERROR. In the following sections, we will treat each class in detail.

3.3.1. ACCESS_CLASS

In the declaration of ACCESS_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are BY_ALLOCATION, DUPLICATE, WITH_CONSTRAINT, WITH_SUBTYPE, and WITH_VALUE. All other options are ignored.

PURPOSE: Declare an ACCESS_CLASS variable.

FUNCTION: If declaration is BY_ALLOCATION, then pop the ACCESS_VAR off the top of the CONTROL_STACK, and get the class and type information for the type; according to the kind of the designated access objects, allocate a variable in the DATA_STACK; if the declaration is WITH_CONSTRAINT, the constraint information must be popped off the CONTROL_STACK, and the TYPE_STACK updated accordingly; if the declaration is WITH_SUBTYPE, the type information must be popped off the CONTROL_STACK, and they TYPE_STACK updated accordingly; if the declaration is WITH_VALUE, then the initial value must be popped of the CONTROL_STACK and the DATA_STACK updated accordingly. On the other hand, if the declaration is a DUPLICATE, the CONTROL_STACK is popped to reference an ACCESS_VAR. Finally, in all other option cases, the CONTROL_STACK is popped to reference an ACCESS_VAR; a variable is created with the appropriate privacy and visibility set. In all cases, the final action is to push an ACCESS_VAR on the CONTROL_STACK that references the declared object.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an ACCESS_VAR which denotes the type of the variable to be declared. The rest of the items on the CONTROL_STACK vary depending upon the kind of the designated access object, but may include constraint, subtype, and initial value information.

Postconditions: The CONTROL_STACK is popped to the end of all the constraint, subtype, and initial value information. An ACCESS_VAR is pushed on the CONTROL_STACK which reference the newly created variable. The TYPE_STACK may be updated to reference any constraints of subtypes. The DATA_STACK collection associated with the particular ACCESS_VAR is updated to contain the newly designated access object, along with an initial value, if any.

EXCEPTIONS: TYPE_ERROR is raised if the ACCESS_VAR is invalid. OPERAND_CLASS_ERROR is raised if the ACCESS_VAR is not present. CAPABILITY_ERROR is raised if the ACCESS_VAR is private. CONSTRAINT_ERROR may be raised if any given constraints or initial values are not compatible with the parent. STORAGE_ERROR may be raised if there is not sufficient room in the collection for the new declaration.

3.3.2. ANY_CLASS

In the declaration of an ANY_CLASS variable, the class of the object is not determined until execution time. Thus, the effect of VARIABLE_VISIBILITY and VARIABLE_OPTIONS is not bound until execution.

PURPOSE: Declare an ANY_CLASS variable.

FUNCTION: Examine the top of the CONTROL_STACK to determine the kind of operand on the top. Recursively execute the DECLARE_VARIABLE instruction using this kind as the VARIABLE_CLASS.

STACKS: State of all stacks is dependent upon the kind of operand on top of the CONTROL_STACK.

EXCEPTIONS: Kinds of exceptions that may be raised depend upon the kind of operand on top of the CONTROL_STACK.

3.3.3. ARRAY_CLASS

In the declaration of ARRAY_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are DUPLICATE, UNCHECKED, and WITH_CONSTRAINT. All other options are ignored.

PURPOSE: Declare an ARRAY_CLASS variable.

FUNCTION: If declaration is DUPLICATE, then read the top of the CONTROL_STACK and copy the type information of the variable located there; allocate space in the DATA_STACK and copy the value from the original variable. If UNCHECKED or

WITH_CONSTRAINT, pop the indexed operand and get the type information of class; pop the CONTROL_STACK to get the dimensionality of the object, and allocate space in the DATA_STACK for the variable preface; pop the bounds information for each index from the CONTROL_STACK and allocate space in the DATA_STACK for the variable object; clear or set the initial value of the variable if necessary. In all other cases, pop the indexed operand and get the type information of the class; trace the type information of the class, and allocate space in the DATA_STACK for the variable object; clear or set the initial value or the variable if necessary. In all cases, the final action is to push an ARRAY_VAR on the CONTROL_STACK indicating the new variable.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an ARRAY_VAR. Stack may next contain dimensionality and bounds information.

Postconditions: Top of CONTROL_STACK is reduced below bounds information. an ARRAY_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK may be updated to include bounds information. The DATA_STACK has space allocated for the new variable.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal. CAPABILITY_ERROR is raised if the class is private. TYPE_ERROR is raised if the ARRAY_VAR is not valid. OPERAND_CLASS_ERROR is raised if the ARRAY_VAR in not present. STORAGE_ERROR is raised if there is no space left to allocate the variable.

3.3.4. DISCRETE_CLASS

In the declaration of DISCRETE_CLASS variables, the only field of VARIABLE_OPTIONS that is relevant is DUPLICATE. All other fields are ignored.

PURPOSE: Declare a DISCRETE_CLASS variable.

FUNCTION: If DUPLICATE, then read the top of the CONTROL_STACK to copy the DISCRETE_VAR. In not DUPLICATE, the pop the DISCRETE_VAR on top of the CONTROL_STACK, and set the visibility as appropriate. In both cases, the final action is to push a DISCRETE_VAR on top of the CONTROL_STACK with a null initial value.

STACKS: Preconditions: Top of CONTROL_STACK must contain a DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one if not DUPLICATE. A DISCRETE_VAR is pushed on top of the CONTROL_STACK with a null initial value.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the DISCRETE_VAR is not present. CAPABILITY_ERROR is raised if the DISCRETE_VAR is private.

3.3.5. ENTRY_CLASS

In the declaration of ENTRY_CLASS variables, all fields of VARIABLE_OPTIONS are ignored.

PURPOSE: Declare an ENTRY_CLASS variable.

FUNCTION: Create a new entry name. Pop the CONTROL_STACK to determine the number of parameters. Push an ENTRY_VAR variable on the CONTROL_STACK denoting the new variable.

STACKS: Preconditions: A value indicating the number of entry parameters is on top of the CONTROL_STACK.

Postconditions: The CONTROL_STACK is reduced by one, and then an ENTRY_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the parameter count is not present. CAPABILITY_ERROR is raised if the parameter count is private. ELABORATION_ERROR is raised if the context is not a task.

3.3.6. FAMILY_CLASS

In the declaration of FAMILY_CLASS variables, all fields of VARIABLE_OPTIONS are ignored.

PURPOSE: Declare a FAMILY_CLASS variable.

FUNCTION: Pop the CONTROL_STACK to get the family index. Trace the type link to get the bounds of the discrete type. Create a name for the family, and pop the CONTROL_STACK to determine the number of parameters for the entry. If the family size is zero, then simply create an empty

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the family index is not present. CAPABILITY_ERROR is raised if the index is private.

3.3.7. FLOAT_CLASS

In the declaration of FLOAT_CLASS variables, the only field of VARIABLE_OPTIONS that is relevant is DUPLICATE.

PURPOSE: Declare a FLOAT_CLASS variable.

FUNCTION: If DUPLICATE, then read the top of the CONTROL_STACK to copy the FLOAT_VAR. In not DUPLICATE, the pop the FLOAT_VAR on top of the CONTROL_STACK, and set the visibility as appropriate. In both cases, the final action is to push a FLOAT_VAR on top of the CONTROL_STACK with a null initial value.

STACKS: Preconditions: Top of CONTROL_STACK must contain a FLOAT_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one if not DUPLICATE. A FLOAT_VAR is pushed on top of the CONTROL_STACK with a null initial value.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the FLOAT_VAR is not present. CAPABILITY_ERROR is raised if the FLOAT_VAR is private.

3.3.8. MATRIX_CLASS

In the declaration of MATRIX_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are DUPLICATE, UNCHECKED, and WITH_CONSTRAINT. All other options are ignored.

PURPOSE: Declare an MATRIX_CLASS variable.

FUNCTION: If declaration is DUPLICATE, then read the top of the CONTROL_STACK and copy the type information of the variable located there; allocate space in the DATA_STACK and copy the value from the original variable. If UNCHECKED or WITH_CONSTRAINT, pop the indexed operand and get the type information of class; pop the CONTROL_STACK to get the dimensionality of the object, and allocate space in the DATA_STACK for the variable preface; pop the bounds information for each index from the CONTROL_STACK and allocate space in the DATA_STACK for the variable object; clear or set the initial value of the variable if necessary. In all other cases, pop the indexed operand and get the type information of the class; trace the type information of the class, and allocate space in the DATA_STACK for the variable object; clear or set the initial value or the variable if necessary. In all cases,

the final action is to push an MATRIX_VAR on the CONTROL_STACK indicating the new variable.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an MATRIX_VAR. Stack may next contain dimensionality and bounds information.

Postconditions: Top of CONTROL_STACK is reduced below bounds information. an MATRIX_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK may be updated to include bounds information. The DATA_STACK has space allocated for the new variable.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal. CAPABILITY_ERROR is raised if the class is private. TYPE_ERROR is raised if the MATRIX_VAR is not valid. OPERAND_CLASS_ERROR is raised if the MATRIX_VAR in not present. STORAGE_ERROR is raised if there is no space left to allocate the variable.

3.3.9. PACKAGE_CLASS

In the declaration of PACKAGE_CLASS variables, the only field of VARIABLE_OPTIONS that is relevant is DISTRIBUTOR.

PURPOSE: Declare a PACKAGE_CLASS variable.

FUNCTION: Pop a PACKAGE_VAR off the top of the CONTROL_STACK. Set the appropriate visibility. Push a PACKAGE_VAR back on the CONTROL_STACK. Get a module name, and send the message DECLARE_MODULE to the module.

STACKS: Preconditions: Top of CONTROL_STACK contains a PACKAGE_VAR.

Postconditions: CONTROL_STACK is reduced by one. A PACKAGE_VAR is pushed on top of the stack. Other stack changes occur as a result of normal message passing.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the PACKAGE_VAR is not present. CAPABILITY_ERROR is raised if the PACKAGE_VAR is private. ELABORATION_ERROR is raised if the module information is not yet elaborated. STORAGE_ERROR is raised if there is no space to allocate the variable.

3.3.10. RECORD_CLASS

In the declaration of RECORD_CLASS variables, the only field of VARIABLE_OPTIONS that is relevant is DUPLICATE.

PURPOSE: Declare a RECORD_CLASS variable.

FUNCTION: If DUPLICATE, then read the top of the CONTROL_STACK and trace the type information of the RECORD_VAR; copy the type information of the type, and create a duplicate variable by allocating space on the DATA_STACK, and copying the data from the original type. If not DUPLICATE, then pop a RECORD_VAR off the CONTROL_STACK; get the type information and set the visibility as appropriate; allocate space on the DATA_STACK and clear the structure if necessary. Push a RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a RECORD_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one, and then a RECORD_VAR is pushed on top of the STACK. DATA_STACK now contains space for the variable.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if reference item for the new variable is not null or if the RECORD_VAR is not found. ELABORATION_ERROR is raised if the declaration is not legal.

3.3.11. SEGMENT_CLASS

In the declaration of SEGMENT_CLASS variables, all values of VARIABLE_OPTIONS are ignored.

PURPOSE: Declare a SEGMENT_CLASS variable.

FUNCTION: Pop a SEGMENT_VAR off the top of the CONTROL_STACK. Set the visibility of the variable, and push the SEGMENT_VAR back on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a SEGMENT_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one, then a Segment_VAR is pushed on the stack.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SEGMENT_VAR is not found. CAPABILITY_ERROR is raised if the SEGMENT_VAR is private.

3.3.12. SELECT_CLASS

In the declaration of SELECT_CLASS variables, no field of VARIABLE_OPTIONS is relevant, and hence they are ignored.

PURPOSE: Declare a SELECT_CLASS variable.

FUNCTION: Pop a value off the CONTROL_STACK indicating the number of accept statement. Pop an other value indicating the number of delay statements. Allocate space in the DATA_STACK for the SELECT_CLASS variable. For each accept statement, build the paths to the corresponding entry in the DATA_STACK. This process involves reading down the CONTROL_STACK for a SUBPROGRAM_VAR the refers to an accept clause. Pop the CONTROL_STACK past all the accept information. For each delay statement, build the paths to the corresponding delay entry in the DATA_STACK. This process involves reading down the CONTROL_STACK for a SUBPROGRAM_VAR that refers to a delay clause. Pop the CONTROL_STACK past all the delay information. Push a SELECT_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a value indicating the number of accept statements. Top - 1 contains a value indicating the number of delay statements. A SUBPROGRAM_VAR follows for each accept and delay statement.

Postconditions: CONTROL_STACK is popped to below all SUBPROGRAM_VAR information. a SELECT_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if either the accept or delay count are not valid values, or if any SUBPROGRAM_VAR is not found. CAPABILITY_ERROR is raised if either the accept or delay count are private. STORAGE_ERROR is raised if there is no space to allocate in the DATA_STACK.

3.3.13. TASK_CLASS

In the declaration of TASK_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are DATA_TASK, HEAP_TASK, and DISTRIBUTOR.

PURPOSE: Declare a TASK_CLASS variable.

FUNCTION: If is DATA_TASK or HEAP_TASK, then pop a TASK_REF off the CONTROL_STACK; otherwise, pop a TASK_VAR off the CONTROL_STACK. Create a module name, and push a TASK_REF or TASK_VAR back on the CONTROL_STACK. If is DATA_TASK then send the message DECLARE_MODULE to the module reference. If is HELP_TASK then send the message ALLOCATE_MODULE to the module reference. Otherwise, send the message DECLARE_MODULE to the module.

STACKS: Preconditions: Top of CONTROL_STACK contains a TASK_REF or TASK_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one. A TASK_REF or TASK_VAR is pushed on top of the CONTROL_STACK. Other stack changes occur as a result of normal message passage.

EXCEPTIONS: ELABORATION_ERROR is raised if the reference task is not elaborated. OPERAND_CLASS_ERROR is raised if the TASK_VAR or TASK_REF is not present.

3.3.14. VARIANT_RECORD_CLASS

In the declaration of VARIANT_RECORD_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are DUPLICATE and WITH_CONSTRAINT.

PURPOSE: Declare a VARIANT_RECORD_CLASS variable.

FUNCTION: If DUPLICATE then read the CONTROL_STACK and trace the type part of the VARIANT_RECORD_VAR; allocate space on the DATA_STACK and copy information from the first variable. In not DUPLICATE, then pop a VARIANT_RECORD_VAR off the top of the CONTROL_STACK; trace the class and type information; if WITH_CONSTRAINT then pop the constraint information for each discriminant and allocate space on the DATA_STACK; in not WITH_CONSTRAINT then allocate the variable as is; initialize the variable, which may require values to be popped from the CONTROL_STACK. As the final action in any case, push a VARIANT_RECORD_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK must contain a VARIANT_RECORD_VAR. Constraint information may follow, as well as initialization information.

Postconditions: CONTROL_STACK is popped below all constraint and initialization information. A VARIANT_RECORD_VAR is pushed on top of the CONTROL_STACK. Space is allocated on the DATA_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIANT_RECORD_VAR is not found, or if any constraint or initialization information is not present. CONSTRAINT_ERROR is raised if any constraints do not satisfy a discriminant type STORAGE_ERROR is raised if there is insufficient room in the DATA_STACK.

3.3.15. VECTOR_CLASS

In the declaration of VECTOR_CLASS variables, the only fields of VARIABLE_OPTIONS that are relevant are DUPLICATE, UNCHECKED, and WITH_CONSTRAINT. All other options are ignored.

PURPOSE: Declare an VECTOR_CLASS variable.

FUNCTION: If declaration is DUPLICATE, then read the top of the CONTROL_STACK and copy the type information of the variable located there; allocate space in the DATA_STACK and copy the value from the original variable. If UNCHECKED or WITH_CONSTRAINT, pop the indexed operand and get the type information of class; pop the CONTROL_STACK to get the dimensionality of the object, and allocate space in the DATA_STACK for the variable preface; pop the bounds information for each index from the CONTROL_STACK and allocate space in the DATA_STACK for the variable object; clear or set the initial value of the variable if necessary. In all other cases, pop the indexed operand and get the type information of the class; trace the type information of the class, and allocate space in the DATA_STACK for the variable object; clear or set the initial value or the variable if necessary. In all cases, the final action is to push an VECTOR_VAR on the CONTROL_STACK indicating the new variable.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an VECTOR_VAR. Stack may next contain dimensionality and bounds information.

Postconditions: Top of CONTROL_STACK is reduced below bounds information. an VECTOR_VAR is pushed on top of the CONTROL_STACK. The TYPE_STACK may be updated to include bounds information. The DATA_STACK has space allocated for the new variable.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal. CAPABILITY_ERROR is raised if the class is private. TYPE_ERROR is raised if the VECTOR_VAR is not valid. OPERAND_CLASS_ERROR is raised if the VECTOR_VAR in not present. STORAGE_ERROR is raised if there is no space left to allocate the variable.

3.4. DECLARE_SUBPROGRAM

The DECLARE_SUBPROGRAM instruction creates a subprogram object.

Formally, DECLARE_SUBPROGRAM takes the form:

```
type DECLARE_SUBPROGRAM is
  record
    SUBPROGRAM_KIND       : SUBPROGRAM_SORT;
    SUBPROGRAM_STATE      : ELABORATION_STATE;
    SUBPROGRAM_VISIBILITY : VISIBILITY;
  end record;
```

A value of SUBPROGRAM_SORT defines the use for the target subprogram, and takes the form:

```
type SUBPROGRAM_SORT is
       (FOR_ACCEPT,    FOR_CALL,
        FOR_INTERFACE, FOR_OUTER_CALL,
        FOR_UTILITY);
```

A value of ELABORATION_STATE specifies the level to which the subprogram is to be elaborated, and is defined as:

```
type ELABORATION_STATE is (ACTIVE, INACTIVE. UNSPECIFIED);
```

Finally, as with DECLARE_VISIBILITY, a value of VISIBILITY defines the visibility of the declared subprogram. We have defined VISIBILITY in section 3.3.

PURPOSE: Declare a subprogram.

FUNCTION: Create a SUBPROGRAM_VAR and set the visibility, sort, and elaboration state as appropriate. Push the SUBPROGRAM_VAR on the CONTROL_STACK.

STACKS: Postconditions: A SUBPROGRAM_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: ELABORATION_ERROR is raised if the declaration is not legal in the given context.

Chapter 4
IMPERATIVE INSTRUCTIONS

An imperative instruction invokes an operation upon an object of a given class. Such instructions may be either classed (the object is referenced explicitly) or unclassed (the object is referenced implicitly). The semantics of these instructions form the key to defining and enforcing data abstraction and information hiding at the lowest levels of the architecture. As with strongly typed languages, the architecture permits only a well-defined set of operations for each class of objects.

Imperative instructions include the following opcodes:

* ACTION -- perform a system level operation
* EXECUTE -- perform an operation upon an object of a given class

In the following sections, we treat each opcode in detail.

4.1. ACTION

The ACTION instruction performs a system level operation. ACTION is an unclassed instruction, since the target object is referenced implicitly (in general, the referent is the subject module of the current thread of control). Formally, ACTION takes the form:

```
type ACTION_INSTRUCTION is
  record
    TO_PERFORM : UNCLASSED_ACTION;
  end record;
```

The operation TO_PERFORM is of the type UNCLASSED_ACTION, which we further define as:

```
type UNCLASSED_ACTION is
      ACCEPT_ACTIVATION,      ACTIVATE_TASKS,         ACTIVATE_SUBPROGRAM,
      ALTER_BREAK_MASK,       BREAK_OPTIONAL,         BREAK_UNCONDITIONAL,
      CALL_IMPORT,            CALL_REFERENCE,         DELETE_ITEM,
      DELETE_SUBPROGRAM,      ESTABLISH_FRAME,        EXIT_BREAK,
      IDLE,                   ILLEGAL,                INITIATE_DELAY,
      INTRODUCE_IMPORT,       MAKE_NULL_UTILITY,      MAKE_SELF,
      MARK_AUXILIARY,         MARK_DATA,              MARK_TYPE,
      NAME_MODULE,            NAME_PARTNER,           OVERWRITE_IMPORT,
      POP_AUXILIARY,          POP_CONTROL,            POP_DATA,
      POP_TYPE,               PROPAGATE_ABORT,        QUERY_BREAK_ADDRESS,
      QUERY_BREAK_CAUSE,      QUERY_BREAK_MASK,       QUERY_FRAME,
      QUERY_RESOURCE_LIMITS,  QUERY_RESOURCE_STATE,   READ_IMPORT,
```

```
    RECOVER_RESOURCES,      REFERENCE_IMPORT,      REMOVE_IMPORT,
    RETURN_RESOURCES,       SET_BREAK_MASK,        SET_INTERFACE_SCOPE,
    SET_INTERFACE_SUBPROG,  SET_NULL_ACCESS,       SET_RESOURCE_LIMITS,
    SET_VISIBILITY,         SIGNAL_ACTIVATED,      SIGNAL_COMPLETION,
    SWAP_CONTROL,           WRITE_IMPORT);
```

In the following sections, we provide a detailed description of each UNCLASSED_ACTION, categorized as:

ACTIVATION_ACTION
: Provides protocol for the activation of a module.

CREATION_ACTION
: Provides facilities for the creation of new program units.

IMPORT_ACTION
: Provides facilities for manipulating the IMPORT_STACK.

INTERFACE_ACTION
: Provides an interface between modules and their external environment.

NULL_ACTION
: Defines null and illegal instructions.

REFERENCE_ACTION
: Provides facilities for the manipulation of remote subprograms.

RESOURCE_ACTION
: Provides facilities for allocating and recovering resources.

STACK_ACTION
: Provides facilities for manipulating various stacks.

TASK_ACTION
: Provides facilities for putting a module to sleep or aborting children tasks.

4.1.1. ACTIVATION_ACTION

These operations provide the protocol for the activation of a module (that is, a task or a package). Since the Rational Machines architecture treats each subprogram as subordinate to another package or task, subprogram activation is achieved with a different set of instructions, namely the CALL instruction for activating locally declared subprograms, and the ACTION REFERENCE_ACTION for activating visible but remotely declared subprograms.

ACTIVATION_ACTIONs include:

* ACCEPT_ACTIVATION
* ACTIVATE_TASKS

* SIGNAL_ACTIVATED
* SIGNAL_COMPLETION

ACCEPT_ACTIVATION

PURPOSE Signal that elaboration of the visible part of a module is complete; module is now ready to accept activation from the parent.

FUNCTION Change module current mode to ACTIVATING, and send the message NOTIFY_DECLARED to the parent module.

STACKS No change except due to message passage.

EXCEPTIONS None.

ACTIVATE_TASKS

PURPOSE: Signal all children tasks that they may begin execution.

FUNCTION: Send the message ACTIVATE_MODULE to each child task; execution of the current module may proceed once all children have been successfully activated.

STACKS: No change except due to message passage.

EXCEPTIONS: TASKING_ERROR may be raised if a child cannot be activated.

SIGNAL_ACTIVATED

PURPOSE: Signal the creator of a module that elaboration of the module body is complete. For a package module, this means that the package body has been executed; in the case of a task module, this means that the module is activated and is running concurrently with the parent.

FUNCTION: Change module current mode to EXECUTING, and send the message NOTIFY_ACTIVATED to the declaring module.

STACKS: No change except due to message passage.

EXCEPTIONS: None.

SIGNAL_COMPLETION

PURPOSE: Signal the creator of a module that processing of the module is complete.

FUNCTION: If module is a task, mark the current mode as TERMINATING and wait for all dependent children to terminate. Additionally, purge any entry queues and send the message END_RENDEZVOUS to any waiting callers. Once all children are terminated or are ready to terminate, send the message NOTIFY_TERMINATION to the declaring module. When deallocation of the dependent children and the module itself begins, the module current mode is marked as COMPLETED.

If module is a package, wait for all dependent children to terminate. Once all children are terminated, send the message NOTIFY_TERMINABLE to the declaring module. Start deallocation of the dependent children and the module itself, and mark the module current mode as TERMINATED.

STACKS: Postconditions: QUEUE_STACK is purged. No other change except due to message passage.

EXCEPTIONS: If module is a task, and callers are waiting in entry

queues, the message END_RENDEZVOUS has the side effect of raising TASKING_ERROR in any calling tasks.

4.1.2. CREATION_ACTION

These operations provide a facility for the creation of new subprograms, packages, or tasks within the current context.

CREATION_ACTIONs include:

* MAKE_NULL_UTILITY
* MAKE_SELF
* NAME_MODULE
* NAME_PARTNER

MAKE_NULL_UTILITY

PURPOSE: Create a null subprogram variable.

FUNCTION: Push a null SUBPROGRAM_VAR control word on the CONTROL_STACK of the current module.

STACKS: Postconditions: SUBPROGRAM_VAR word pushed on top of CONTROL_STACK.

EXCEPTIONS: None.

MAKE_SELF

PURPOSE: Create a null program module of the same kind as the current module.

FUNCTION: If the current module is a task, push a TASK_VAR on the CONTROL_STACK, else push a PACKAGE_VAR on the CONTROL_STACK.

STACKS: Postconditions: Either a TASK_VAR or a PACKAGE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

NAME_MODULE

PURPOSE: Get the name of the module currently on top of the CONTROL_STACK.

FUNCTION: Duplicate the name of the current module and push is as a DISCRETE_VAR on the CONTROL_STACK.

STACKS: Postconditions: A DISCRETE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

NAME_PARTNER

PURPOSE: Get the name of the module that is in rendezvous with the current module.

FUNCTION: Trace through the CONTROL_STACK to the reference of the module that is currently in a rendezvous with the current module, and push the name of that module as a DISCRETE_VAR on the CONTROL_STACK. If no rendezvous is in progress, push the representation for a null module.

STACKS: Postconditions: A DISCRETE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

4.1.3. IMPORT_ACTION

These operations provide facilities for manipulating the IMPORT_STACK.

IMPORT_ACTIONs include:

* CALL_IMPORT
* INTRODUCE_IMPORT
* OVERWRITE_IMPORT
* READ_IMPORT
* REFERENCE_IMPORT
* REMOVE_IMPORT
* WRITE_IMPORT

CALL_IMPORT

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

INTRODUCE_IMPORT

PURPOSE: Add an import item onto a given module.

FUNCTION: Pop the CONTROL_STACK to determine the target of import, and follow the type path to access its current import information on the corresponding TYPE_STACK.. Pop the CONTROL_STACK again to determine the entity that is to be imported, and add a reference to that entity in the target module's IMPORT_STACK, which is extended if necessary. A path is added to the target module's TYPE_STACK leading from the target modules' import information and referring to the imported entity.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 1 contains an IMPORT_VAR.

Postconditions: Top of CONTROL_STACK is reduced below the IMPORT_VAR. The new IMPORT_VAR is pushed on top of the IMPORT_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if the target module is private or if the module is not statically scoped.

INSTRUCTION_ERROR is raised if the target module is null. OPERAND_CLASS_ERROR is raised if the IMPORT_VAR is not found.

OVERWRITE_IMPORT

PURPOSE: Write over an import item in a given module.

FUNCTION: Pop the CONTROL_STACK to determine the target module of import, and follow the type path to access its current import information on the corresponding TYPE_STACK. Pop the CONTROL_STACK again to determine the site of the existing import that is to be overwritten. Pop the CONTROL_STACK to obtain the new IMPORT_VAR, and write the information at the given site.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 1 contains a VALUE_VAR indicating the IMPORT_STACK site. Top - 2 contains an IMPORT_VAR.

Postconditions: Top of CONTROL_STACK is reduced below the IMPORT_VAR. The IMPORT_STACK contains a new entry at the given site.

EXCEPTIONS: CAPABILITY_ERROR raised if the target module is private or if it is not statically nested. INSTRUCTION_ERROR is raised if the target module is null. OPERAND_CLASS_ERROR is raised if the IMPORT_STACK does not contain a delete key at the site of the overwrite or if the new import is not an IMPORT_VAR. CONSTRAINT_ERROR is raised if the intended site is not on the target IMPORT_STACK.

READ_IMPORT

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

REFERENCE_IMPORT

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

REMOVE_IMPORT

PURPOSE\Remove an import item to a given module.

FUNCTION: Pop the CONTROL_STACK to determine the target module of import, and follow the type path to access its current import information on the corresponding TYPE_STACK. Pop the CONTROL_STACK again to determine the site of the existing import that is to be removed. Pop the CONTROL_STACK to obtain the new DELETION_KEY, and write the deletion information at the given site.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 1 contains a VALUE_VAR indicating the IMPORT_STACK site. Top - 2 contains a DELETION_KEY.

Postconditions: Top of CONTROL_STACK is reduced below the DELETION_KEY. The IMPORT_STACK contains a deletion mark at the given site.

EXCEPTIONS: CAPABILITY_ERROR raised if the target module is private or if it is not statically nested. INSTRUCTION_ERROR is raised if the target module is null. OPERAND_CLASS_ERROR is raised if the IMPORT_STACK does not contain a delete key at the site of the deletion. CONSTRAINT_ERROR is raised if the intended site is not on the target IMPORT_STACK.

WRITE_IMPORT

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

4.1.4. INTERFACE_ACTION

These operations provide an interface between modules and their external environments. INTERFACE_ACTIONs are primarily used in support of the programming environment debugging facilities.

INTERFACE_ACTIONs include:

* ALTER_BREAK_MASK
* BREAK_OPTIONAL
* BREAK_UNCONDITIONAL
* ESTABLISH_FRAME
* EXIT_BREAK
* QUERY_BREAK_ADDRESS
* QUERY_BREAK_CAUSE
* QUERY_BREAK_MASK
* QUERY_FRAME
* SET_BREAK_MASK
* SET_INTERFACE_SCOPE
* SET_INTERFACE_SUBPROGRAM

ALTER_BREAK_MASK

PURPOSE: Change the value of the breakpoint mask.

FUNCTION: Pop a VARIABLE_REF off the CONTROL_STACK and trace the site of the INTERFACE_KEY. Pop the new break mask value off the CONTROL_STACK and write the value to the key site of the current module.

STACKS: Preconditions: Top of CONTROL_STACK contains a VARIABLE_REF. Top - 1 contains a breakpoint value.

Postconditions: Top of CONTROL_STACK reduced below the breakpoint value. A new breakpoint value is written at the key site on the CONTROL_STACK of the current module.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found. CAPABILITY_ERROR is raised if the site of the INTERFACE_KEY is not in the current module. INSTRUCTION_ERROR is raised if the breakpoint mask is not a legitimate value.

BREAK_OPTIONAL

PURPOSE: Cause a breakpoint exception if breakpoints are currently set.

FUNCTION: Read the enabled status of the breakpoints

for the current module. If breakpoints are set, raise the exception BREAKPOINT_ACTION.

STACKS: None.

EXCEPTIONS: BREAKPOINT_ACTION raised if breakpoints are set.

BREAK_UNCONDITIONAL

PURPOSE: Cause a breakpoint action.

FUNCTION:\Raise the exception BREAKPOINT_ACTION.

STACKS: None.

EXCEPTIONS: BREAKPOINT_ACTION is raised.

ESTABLISH_FRAME

PURPOSE: Create a new frame on the CONTROL_STACK.

FUNCTION: Pop a VARIABLE_REF off the CONTROL_STACK and trace the type information to the site of the INTERFACE_KEY. Pop a value off the CONTROL_STACK indicating the depth of search for the parent frame. Trace down the depth of the search for the site of the activation record of the frame. If the frame exists, note that if it is statically scoped, it is accessible and if it is not statically scoped, it is inaccessible. If the frame is accessible, create a SUBPROGRAM_VAR linked to the parent outer frame, indicating the visibility, start, and lexical level of the new subprogram. Push the SUBPROGRAM_VAR as a nonlocal frame.

STACKS: Preconditions: Top of CONTROL_STACK contains a VARIABLE_REF. Top - 1 contains a value indicating the depth of the parent frame.

Postconditions: Top of CONTROL_STACK is reduced below the depth value.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the

VARIABLE_REF is not found, if the INTERFACE_KEY is not found, or if the depth value is not found. CAPABILITY_ERROR is raised if the site of the INTERFACE_KEY is not in the current module.

EXIT_BREAK

PURPOSE: Return from a breakpoint action.

FUNCTION: Determine the location of the INTERFACE_KEY on the inner frame. Read the INTERFACE_KEY at that point and restore the breakpoint mask. Pop the frame on top of the CONTROL_STACK.

STACKS: Postcondition: The frame on top of the CONTROL_STACK is popped.

EXCEPTIONS: INSTRUCTION_ERROR raised if the location of the key is not in the CONTROL_STACK, if the INTERFACE_KEY is not found, or if the top frame cannot be popped.

QUERY_BREAK_ADDRESS

PURPOSE: Return the address of the breakpoint handler.

FUNCTION: Pop a VARIABLE_REF off the top of the CONTROL_STACK and trace it to the site of the INTERFACE_KEY. Push the address indicated in the INTERFACE_KEY in a DISCRETE_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a VARIABLE_VAR.

Postconditions: Pop the CONTROL_STACK by one, and then push an address value as a DISCRETE_VAR on the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found or if the INTERFACE_KEY is not found.

QUERY_BREAK_CAUSE

PURPOSE: Return the cause of the breakpoint action.

FUNCTION: Pop a VARIABLE_REF off the top of the CONTROL_STACK and trace it to the site of the INTERFACE_KEY. Push the break cause indicated in the INTERFACE_KEY in a DISCRETE_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a VARIABLE_VAR.

Postconditions: Pop the CONTROL_STACK by one, and then push a DISCRETE_VAR on the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found or if the INTERFACE_KEY is not found.

QUERY_BREAK_MASK

PURPOSE: Return the value of the breakpoint mask.

FUNCTION: Pop a VARIABLE_REF off the top of the CONTROL_STACK and trace it to the site of the INTERFACE_KEY. Push the break mask indicated in the INTERFACE_KEY in a DISCRETE_VAR on the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a VARIABLE_VAR.

Postconditions: Pop the CONTROL_STACK by one, and then push a DISCRETE_VAR on the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found or if the INTERFACE_KEY is not found.

QUERY_FRAME PURPOSE:
Determine the status of the target frame.

FUNCTION:
Pop a VARIABLE_REF off the CONTROL_STACK and trace the type information to the site of the INTERFACE_KEY. Pop a value off the CONTROL_STACK indicating the depth of search for the parent frame. Trace down the depth of the search for the site of the activation record of the frame. If the

frame exists, note that if it is statically scoped, it is accessible and if it is not statically scoped, it is inaccessible. If the parent frame is accessible, create a DISCRETE_VAR indicating a link to the parent as the outer frame. Push the DISCRETE_VAR on the CONTROL_STACK; create another DISCRETE_VAR indicating a return address and push the value on the CONTROL_STACK. In any case, if the parent frame exists of not, create a DISCRETE_VAR indicating the status of the parent, and push the value on the CONTROL_STACK.

STACKS:

Preconditions: Top of CONTROL_STACK contains a VARIABLE_REF. Top - 1 contains a value indicating the depth of the parent frame.

Postconditions: Top of CONTROL_STACK is reduced below the depth value. If the parent exists, a DISCRETE_VAR with frame link, followed by a DISCRETE_VAR with a return address, are pushed on the CONTROL_STACK. A DISCRETE_VAR is pushed on the CONTROL_STACK in all cases, indicating the status of the parent frame.

EXCEPTIONS:

OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found, if the INTERFACE_KEY is not found, or if the depth value is not found. CAPABILITY_ERROR is raised if the site of the INTERFACE_KEY is not in the current module.

SET_BREAK_MASK

PURPOSE: Set the breakpoint mask.

FUNCTION: Pop a MODULE_VAR off the CONTROL_STACK and trace the module path to find the name of the particular module instance. Pop a DISCRETE_VAR off the CONTROL_STACK as the new breakpoint mask. Write the debugging information and the control state to the CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 2 contains a DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced below the DISCRETE_VAR. Debugging information and control state is written into the CONTROL_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if the MODULE_VAR is private or if a particular instance is not found. OPERAND_CLASS_ERROR is raised if the MODULE_VAR is not found.

SET_INTERFACE_SCOPE

PURPOSE: Establish the scope of the debugging information.

FUNCTION: Pop a MODULE_VAR off the CONTROL_STACK and trace the module path to find the name of the particular module instance. Pop another MODULE_VAR off the CONTROL_STACK indicating the scope of the debugging information. Write the name of the second module to the debugging information of the first module in its CONTROL_STACK.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 1 contains a MODULE_VAR.

Postconditions: Top of CONTROL_STACK is below the second MODULE_VAR. Debugging information is written into the CONTROL_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if either MODULE_VAR is private or if a particular instance of the first module is not found. OPERAND_CLASS_ERROR is raised either MODULE_VAR is not found.

SET_INTERFACE_SUBPROGRAM

PURPOSE: Establish the name of the breakpoint interface subprogram.

FUNCTION: Pop a MODULE_VAR off the CONTROL_STACK and trace the module path to find the name of the particular module instance. Pop a SUBPROGRAM_VAR off the CONTROL_STACK as the name of the interface subprogram. Write the name of the interface subprogram to the CONTROL_STACK of the target module.

STACKS: Preconditions: Top of CONTROL_STACK contains a MODULE_VAR. Top - 1 contains a SUBPROGRAM_VAR.

Postconditions: Top of CONTROL_STACK is reduced below the SUBPROGRAM_VAR. The name of the interface subprogram is written to the CONTROL_STACK.

EXCEPTIONS: CAPABILITY_ERROR is raised if the MODULE_VAR is private or if a particular instance is not found. OPERAND_CLASS_ERROR is raised if the MODULE_VAR is not found. INSTRUCTION_ERROR is raised if the target subprogram is not designated as an interface subprogram, or if it is not defined at lexical level 2.

4.1.5. NULL_ACTION

These operations provide a definition for null and illegal instructions.

NULL_ACTIONs include:

* IDLE
* ILLEGAL

IDLE

PURPOSE: No operation.

FUNCTION: No operation.

STACKS: None.

EXCEPTIONS: None.

ILLEGAL

PURPOSE: Definition of an illegal instruction.

FUNCTION: Unconditionally raise the exception INSTRUCTION_ERROR.

STACKS: None.

EXCEPTIONS: INSTRUCTION_ERROR raised.

4.1.6. REFERENCE_ACTION

These operations provide facilities for manipulating remotely declared subprograms and variables.

REFERENCE_ACTIONS include:

* ACTIVATE_SUBPROGRAM
* CALL_REFERENCE
* DELETE_ITEM
* DELETE_SUBPROGRAM
* SET_NULL_ACCESS
* SET_VISIBILITY

ACTIVATE_SUBPROGRAM

PURPOSE: Activate a remotely declared subprogram.

FUNCTION: Pop a SUBPROGRAM_REF off the CONTROL_STACK. Trace the reference to the SUBPROGRAM_VAR on the CONTROL_STACK. If the SUBPROGRAM_VAR is defined for a call, set the subprogram active and write the reference to the SUBPROGRAM_VAR.

STACKS: Preconditions: Top of CONTROL_STACK contains a SUBPROGRAM_REF.

Postconditions: Top of CONTROL_STACK is reduced below the SUBPROGRAM_REF. The reference is written to the SUBPROGRAM_VAR.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_REF is not found. INSTRUCTION_ERROR is raised if the SUBPROGRAM_VAR is not defined for a call.

CALL_REFERENCE

PURPOSE: Call a remotely declared subprogram.

FUNCTION: Pop a SUBPROGRAM_REF off the CONTROL_STACK. If the SUBPROGRAM_VAR is defined for a call, trace the reference to the SUBPROGRAM_VAR on the CONTROL_STACK. If the target subprogram is active, create a new nonlocal frame for the subprogram.

STACKS: Preconditions: Top of CONTROL_STACK contains a SUBPROGRAM_REF.

Postconditions: Top of CONTROL_STACK is reduced below the SUBPROGRAM_REF. A new frame is created for the subprogram call.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_REF is not found. INSTRUCTION_ERROR is raised if the SUBPROGRAM_VAR is not defined for a call. ELABORATION_ERROR is raised if the target subprogram is not active.

DELETE_ITEM

PURPOSE: Delete a referenced variable.

FUNCTION: Pop a VARIABLE_REF off the CONTROL_STACK. Pop a DELETION_KEY off the CONTROL_STACK. Trace the reference to the item to be deleted. Set the item as locked, and write a deletion value to the variable site.

STACKS: Preconditions: Top of the CONTROL_STACK contains a VARIABLE_REF. Top - 1 contains a DELETION_KEY.

Postconditions: Top of the CONTROL_STACK is reduced below the DELETION_KEY.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF or the DELETION_KEY is not found. CAPABILITY_ERROR is raised if the item is already locked.

DELETE_SUBPROGRAM

PURPOSE: Delete a referenced subprogram.

FUNCTION: Pop a SUBPROGRAM_REF off the CONTROL_STACK. Pop a DELETION_KEY off the CONTROL_STACK. Trace the reference to the item to be deleted. Set the item as locked, and write a deletion value to the subprogram site.

STACKS: Preconditions: Top of the CONTROL_STACK contains a SUBPROGRAM_REF. Top - 1 contains a deletion key.

Postconditions: Top of the CONTROL_STACK is reduced below the DELETION_KEY.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the SUBPROGRAM_REF or the DELETION_KEY is not found. CAPABILITY_ERROR is raised if the item is already locked.

SET_NULL_ACCESS

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

SET_VISIBILITY

PURPOSE: Set the visibility of a reference variable.

FUNCTION: Pop a VARIABLE_REF off the CONTROL_STACK. Trace the reference to the variable site. Set the visibility of the item as true, lock the item, and then write the reference to the variable site.

STACKS: Preconditions: Top of the CONTROL_stack contain a VARIABLE_REF.

Postconditions: Top of the CONTROL_STACK is reduced below the VARIABLE_REF. Visibility is written to the site of the variable.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the VARIABLE_REF is not found. INSTRUCTION_ERROR is raised if there is not a DELETION_KEY at the reference site. CAPABILITY_ERROR is raised if the context is a task.

4.1.7. RESOURCE_ACTION

These operations provide facilities for allocating and recovering resources.

RESOURCE_ACTIONs include:

* QUERY_RESOURCE_LIMITS
* QUERY_RESOURCE_STATE
* RECOVER_RESOURCES
* RETURN_RESOURCES
* SET_RESOURCE_LIMITS

QUERY_RESOURCE_LIMITS

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

QUERY_RESOURCE_STATE

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

RECOVER_RESOURCES

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

RETURN_RESOURCES

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

SET_RESOURCE_LIMITS

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

4.1.8. STACK_ACTION

These operations provide primitive facilities for manipulating the various stacks as defined by the architecture.

STACK_ACTIONs include:

* MARK_AUXILIARY

* MARK_DATA
* MARK_TYPE
* POP_AUXILIARY
* POP_CONTROL
* POP_DATA
* POP_TYPE
* SWAP_CONTROL

MARK_AUXILIARY

PURPOSE: Mark both the DATA_STACK and the TYPE_STACK.

FUNCTION: Save the state of the current frame. Push an AUXILIARY_MARK on the CONTROL_STACK indicating the TYPE_STACK mark and the DATA_STACK mark. Set the state of the current frame as marked, and if it is legal to export the value of the registers, mark their exports as illegal and create a link to the marked frame.

STACKS: Postconditions: An AUXILIARY_MARK is pushed on top of the CONTROL_STACK indicating the mark of the TYPE_STACK and the DATA_STACK.

EXCEPTIONS: None.

MARK_DATA

PURPOSE: Mark the DATA_STACK.

FUNCTION: Save the state of the current frame. Push an AUXILIARY_MARK on the CONTROL_STACK indicating the DATA_STACK mark. Set the state of the current frame as marked, and if it is legal to export the value of the registers, mark their exports as illegal and create a link to the marked frame.

STACKS: Postconditions: An AUXILIARY_MARK is pushed on top of the CONTROL_STACK indicating the mark of the DATA_STACK.

EXCEPTIONS: None.

MARK_TYPE

PURPOSE: Mark the TYPE_STACK.

FUNCTION: Save the state of the current frame. Push an AUXILIARY_MARK on the CONTROL_STACK indicating the TYPE_State of the

current frame as marked, and if it is legal to export the value of the registers, mark their exports as illegal and create a link to the marked frame.

STACKS: Postconditions: An AUXILIARY_MARK is pushed on top of the CONTROL_STACK indicating the mark of the TYPE_STACK.

EXCEPTIONS: None.

POP_AUXILIARY

PURPOSE: Pop the TYPE_STACK and the DATA_STACK down to the last mark.

FUNCTION\Read the top of the CONTROL_STACK for the AUXILIARY_MARK. Pop the TYPE_STACK and then the DATA_STACK down to the mark indicated in the AUXILIARY_MARK. Save the state of the current frame, and note that the AUXILIARY_MARK has a prior mark. If this is an export frame, mark the register exports as legal, and create a link to the current frame.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an AUXILIARY_MARK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the AUXILIARY_MARK is not found.

POP_CONTROL

PURPOSE: Pop the CONTROL_STACK by one item.

FUNCTION: Read the top word of the CONTROL_STACK and if it is a typed item, pop the stack by one word.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a typed word.

Postconditions: Top of CONTROL_STACK is reduced by one.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the top word is not a type item.

POP_DATA

PURPOSE: Pop the DATA_STACK down to the last mark.

FUNCTION\Read the top of the CONTROL_STACK for the AUXILIARY_MARK. Pop the DATA_STACK down to the mark indicated in the AUXILIARY_MARK. Save the state of the current frame, and note that the AUXILIARY_MARK has a prior mark. If this is an export frame, mark the register exports as legal, and create a link to the current frame.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an AUXILIARY_MARK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the AUXILIARY_MARK is not found.

POP_TYPE

PURPOSE: Pop the TYPE_STACK down to the last mark.

FUNCTION\Read the top of the CONTROL_STACK for the AUXILIARY_MARK. Pop the TYPE_STACK down to the mark indicated in the AUXILIARY_MARK. Save the state of the current frame, and note that the AUXILIARY_MARK has a prior mark. If this is an export frame, mark the register exports as legal, and create a link to the current frame.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an AUXILIARY_MARK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the AUXILIARY_MARK is not found.

SWAP_CONTROL

PURPOSE: Exchange the top two elements on the CONTROL_STACK.

FUNCTION: Read the top word on the CONTROL_STACK and read the next word also. If both words are typed, write the first word to the top - 1 of the CONTROL_STACK and write the second word to the top of the CONTROL_STACK.

STACKS: Preconditions: Top two items on the CONTROL_STACK must be typed.

Postconditions: Top two items on the CONTROL_STACK are swapped.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised of the top two words are not typed.

4.1.9. TASK_ACTION

These operations provide facilities for putting a module to sleep or aborting children tasks.

TASK_ACTIONs include:

* INITIATE_DELAY
* PROPAGATE_ABORT

INITIATE_DELAY

PURPOSE: Delay the current thread of control.

FUNCTION: Pop a value off the CONTROL_STACK indicating the delay time. Insert the name of the current module in the delay queue for that time. Mark that module as delaying, and initiate a context swap.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a VALUE_VAR.

Postconditions: No change except due to a normal context swap.

EXCEPTIONS: None.

PROPAGATE_ABORT

PURPOSE: Currently unimplemented instruction.

FUNCTION: Currently unimplemented instruction.

STACKS: Currently unimplemented instruction.

EXCEPTIONS: Currently unimplemented instruction.

4.2. EXECUTE

The EXECUTE instruction performs an operation upon a classed object.

Formally, EXECUTE takes the form:

```
type EXECUTE_INSTRUCTION is
  record
    OPERATOR : OPERATOR_SPEC;
  end record;
```

The OPERATOR is of the type OPERATOR_SPEC, which discriminates among classes of operands that require a field specification, namely objects of the class PACKAGE_CLASS, RECORD_CLASS, SELECT_CLASS, TASK_CLASS, and VARIANT_RECORD_CLASS. We may further define the OPERATOR_SPEC as:

```
type OPERATOR_SPEC(CLASS : OPERAND_CLASS := DISCRETE_CLASS;
                   OP    : OPERATION     := EQUAL_OP) is
  record
    case OP is
      when FIELD_OPERATION =>
             FIELD : FIELD_SPEC(CLASS, OP);
      when others          =>
             null;
    end case;
  end record;
```

A FIELD_OPERATION is a subtype of the type OPERATION, and includes the operators COND_CALL_OP, ENTRY_CALL_OP, FAMILY_CALL_OP, FAMILY_COND_OP, FAMILY_TIMED_OP, FIELD_EXECUTE_OP, FIELD_READ_OP, FIELD_REFERENCE_OP, FIELD_TYPE_OP, FIELD_WRITE_OP, GUARD_WRITE_OP, SET_BOUNDS_OP, SET_VARIANT_OP, and TIMED_CALL_OP.

Formally, FIELD_SPEC takes the form:

```
type FIELD_SPEC(CLASS : OPERAND_CLASS := DISCRETE_CLASS;
                OP    : OPERATION     := EQUAL_OP) is
  record
    case OP is
      when PACKAGE_CLASS | TASK_CLASS   =>
             OFFSET    : FIELD_INDEX;
      when RECORD_CLASS  | SELECT_CLASS =>
             NUMBER    : FIELD_INDEX;
      when VARIANT_RECORD_CLASS         =>
             COMPONENT : ACCESS_SPEC(OP);
             INDEX     : VARIANT_RECORD_INDEX;
      when others                       =>
             null;
    end case;
  end record;
```

The FIELD_INDEX and VARIANT_RECORD_INDEX define the type of the index into an object of a class that requires a field specification, and are formally defined as:

```
type    FIELD_INDEX is new INTEGER range implementation defined;
subtype VARIANT_RECORD_INDEX is FIELD_INDEX  range 1 .. FIELD_INDEX
```

The type ACCESS_SPEC further discriminates the components of a VARIANT_RECORD_CLASS object, and can be formally expressed as:

```
type ACCESS_SPEC(OP : OPERATION := EQUAL_OP) is
  record
    case OP is
      when COMPONENT_OPERATION =>
             KIND : FIELD_SORT;
             MODE : FIELD_MODE;
      when others              =>
             null;
    end case;
  end record;
```

A COMPONENT_OPERATION is a subtype of FIELD_OP, and includes the operations FIELD_READ_OP, FIELD_REFERENCE_OP, and FIELD_WRITE_OP.

Finally, we define the FIELD_MODE and FIELD_SORT as:

```
type FIELD_MODE is (DIRECT, INDIRECT);
type FIELD_SORT is (FIXED, VARIANT);
```

Whereas the type ACCESS_SPEC, FIELD_SPEC, and OPERATOR_SPEC relate the EXECUTE instruction to a particular class, the basis action if defined through the type OPERATION. In general, the target operand will reside on top of the CONTROL_STACK. Or course, since the architecture is strongly typed, only well defined operations are defined for each OPERAND_CLASS. If an attempt to EXECUTE an OPERATION that is not appropriate for the target class, the exception INSTRUCTION_ERROR is raised. If the OPERATION is appropriate, but the object of the target class is not found, then the exception OPERAND_CLASS_ERROR is raised. Finally, if the target if found but the object is private or otherwise out of scope, the exception CAPABILITY_ERROR is raised.

Formally, we define the type OPERATION as:

```
type OPERATION is
       ABORT_OP,                ABOVE_RANGE_OP,          ABSOLUTE_VALUE_OP,
       ACTIVATE_OP,             ADDRESS_OP,              ALL_READ_OP,
       ALL_REFERENCE_OP,        ALL_WRITE_OP,            AND_OP,
       APPEND_OP,               AUGMENT_IMPORTS_OP,      BELOW_RANGE_OP,
       BOUNDS_CHECK_OP,         BOUNDS_OP,               CATENATE_OP,
       CHECK_IN_ROOT_TYPE_OP,   CHECK_IN_TYPE_OP,        COND_CALL_OP,
```

```
        CONTINUE_OP,              CONVERT_ACTUAL_OP,      CONVERT_OP,
        COUNT_OP,                 DECREMENT_OP,           DIVIDE_OP,
        ELABORATE_OP,             ELEMENT_TYPE_OP,        ENTRY_CALL_OP,
        EQUAL_OP,                 FAMILY_CALL_OP,         FAMILY_COND_OP,
        FAMILY_TIMED_OP,          FIELD_EXECUTE_OP,       FIELD_READ_OP,
        FIELD_REFERENCE_OP,       FIELD_TYPE_OP,          FIELD_WRITE_OP,
        FIRST_OP,                 GET_SUBUNIT_OP,         GET_SUBUNIT_COUNT_OP,
        GREATER_EQUAL_OP,         GREATER_EQUAL_ZERO_OP,  GREATER_OP,
        GREATER_ZERO_OP,          GUARD_WRITE_OP,         IN_RANGE_OP,
        IN_TYPE_OP,               INCREMENT_OP,           INSTRUCTION_READ_OP,
        INSTRUCTION_WRITE_OP,     INTERRUPT_OP,           IS_CALLABLE_OP,
        IS_CONSTRAINED_OP,        IS_NULL_OP,             IS_TERMINATED_OP,
        IS_ZERO_OP,               LAST_OP,                LENGTH_OP,
        LESS_EQUAL_OP,            LESS_EQUAL_ZERO_OP,     LESS_OP,
        LESS_ZERO_OP,             MAKE_ADDRESS_OP,        MAKE_ALIGNED_OP,
        MAKE_CONSTANT_OP,         MAKE_CONSTRAINED_OP,    MAKE_VISIBLE_OP,
        MINUS_OP,                 MODULO_OP,              NAME_OP,
        NOT_EQUAL_OP,             NOT_IN_RANGE_OP,        NOT_IN_TYPE_OP,
        NOT_NULL_OP,              NOT_OP,                 NOT_ZERO_OP,
        OR_OP,                    PLUS_OP,                PREDECESSOR_OP,
        PREPEND_OP,               RAISE_OP,               REMAINDER_OP,
        RENDEZVOUS_OP,            REVERSE_BOUNDS_OP,      RUN_UTILITY_OP,
        SCOPE_OF_RAISE_OP,        SET_BOUNDS_OP,          SET_CONSTRAINT_OP,
        SET_SUBUNIT_OP,           SET_SUBUNIT_COUNT_OP,   SET_VARIANT_OP,
        SIZE_OP,                  SLICE_READ_OP,          SLICE_WRITE_OP,
        SUBARRAY_OP,              SUCCESSOR_OP,           TIMED_CALL_OP,
        TIMES_OP,                 UNARY_MINUS_OP,         XOR_OP,
        WORD_WRITE_OP);
```

In the following sections, we provide a detailed description of each OPERATION, categorized as:

ACCESS_OPERATION
: Defines operations specific to access objects.

ARITHMETIC_OPERATION
: Provides primitive mathematical facilities.

ARRAY_OPERATION Defines operations specific to array objects.

ATTRIBUTE_OPERATION
: Provides facilities for accessing attributes of an entity.

BOUNDS_OPERATION
: Provides facilities for manipulating the bounds of an iterator.

CONVERSION_OPERATION
: Provides facilities for explicit conversions from one class to another.

EXCEPTION_OPERATION
Defines operations specific to exception objects.

FIELD_OPERATION Provide operations on classes of operands that require a field specification.

LOGICAL_OPERATION
Provides primitive boolean operations.

MEMBERSHIP_OPERATION
Provides primitive type checking operations.

MODULE_OPERATION
Provides primitive operations for manipulating modules and segments.

RANDOM_OPERATION
Provides unique operations for various classes of operands.

RANGE_OPERATION Provides facilities for manipulating the range of a class.

RELATIONAL_OPERATION
Provides primitive relational operations.

Since we have already mentioned the conditions under which the exceptions CAPABILITY_ERROR, INSTRUCTION_ERROR, and OPERAND_CLASS_ERROR can be raised, we will omit references to such exceptions in the following sections.

4.2.1. ACCESS_OPERATION

These instructions provide actions specific to access objects.

ACCESS_OPERATIONs include:

* ALL_READ_OP
* ALL_REFERENCE_OP
* ALL_WRITE_OP
* IS_NULL_OP
* NOT_NULL_OP

ALL_READ_OP

PURPOSE:

ON CLASS: ACCESS_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ALL_REFERENCE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ALL_WRITE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IS_NULL_OP

PURPOSE:

ON CLASS: ACCESS_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_NULL_OP

PURPOSE:

ON CLASS: ACCESS_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.2. ARITHMETIC_OPERATION

These instructions provide several primitive mathematical manipulation operations.

ARITHMETIC_OPERATIONs include:

* ABSOLUTE_VALUE_OP
* DECREMENT_OP
* DIVIDE_OP
* INCREMENT_OP
* MINUS_OP
* MODULE_OP
* PLUS_OP
* REMAINDER_OP
* TIMES_OP
* UNARY_MINUS_OP

ABSOLUTE_VALUE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

DECREMENT_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

DIVIDE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

INCREMENT_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MINUS_OP

PURPOSE:

ON CLASS: discrete_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MODULO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

PLUS_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

REMAINDER_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

TIMES_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

UNARY_MINUS_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.3. ARRAY_OPERATION

These instructions provide actions specific to array objects.

ARRAY_OPERATIONs include:

* APPEND_OP
* CATENATE_OP
* ELEMENT_TYPE_OP
* PREPEND_OP

* SLICE_READ_OP
* SLICE_WRITE_OP
* SUBARRAY_OP

APPEND_OP

PURPOSE:

ON CLASS: VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

CATENATE_OP

PURPOSE:

ON CLASS: VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ELEMENT_TYPE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ARRAY_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

PREPEND_OP

PURPOSE:

ON CLASS: VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SLICE_READ_OP

PURPOSE:

ON CLASS: VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SLICE_WRITE_OP

PURPOSE:

ON CLASS: VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SUBARRAY_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, MATRIX_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.4. ATTRIBUTE_OPERATION

These instructions facilitate accessing various attributes of an entity.

ATTRIBUTE_OPERATIONs include:

* ADDRESS_OP
* COUNT_OP
* FIRST_OP
* IS_CALLABLE_OP
* IS_CONSTRAINED_OP
* IS_TERMINATED_OP
* LAST_OP
* LENGTH_OP
* PREDECESSOR_OP
* SIZE_OP
* SUCCESSOR_OP

ADDRESS_OP

PURPOSE:

ON CLASS: ANY_CLASS, EXCEPTION_CLASS, SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

COUNT_OP

PURPOSE:

ON CLASS: ENTRY_CLASS, FAMILY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIRST_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IS_CALLABLE_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IS_CONSTRAINED_OP

PURPOSE:

ON CLASS: VARIANT_RECORD_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IS_TERMINATED_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LAST_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LENGTH_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

PREDECESSOR_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SIZE_OP

PURPOSE:

ON CLASS: ANY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SUCCESSOR_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.5. BOUNDS_OPERATION

These operations provide facilities for manipulating the bounds of an iterator.

BOUNDS_OPERATIONs include:

* BOUNDS_CHECK_OP
* BOUNDS_OP
* REVERSE_BOUNDS_OP

BOUNDS_CHECK_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

BOUNDS_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, DISCRETE_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

REVERSE_BOUNDS_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, DISCRETE_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.6. CONVERSION_OPERATION

These operations provide facilities for explicit conversion from one class to another.

CONVERSION_OPERATIONs include:

* CONVERT_ACTUAL_OP
* CONVERT_OP

CONVERT_ACTUAL_OP

PURPOSE:

ON CLASS: ANY_CLASS, ARRAY_CLASS, MATRIX_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

CONVERT_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ANY_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, MODULE_CLASS, RECORD_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.7. EXCEPTION_OPERATION

These operations define actions specific to exception objects.

EXCEPTION_OPERATIONs include:

* NAME
* RAISE_OP
* SCOPE_OF_RAISE

NAME

PURPOSE:

ON CLASS: EXCEPTION_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

RAISE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SCOPE_OF_RAISE

PURPOSE:

ON CLASS: EXCEPTION_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.8. FIELD_OPERATION

These operations provide actions specific to classes of operands that require a field designation.

FIELD_OPERATIONs include:

* COND_CALL_OP
* ENTRY_CALL_OP
* FAMILY_CALL_OP
* FAMILY_COND_OP
* FAMILY_TIMED_OP
* FIELD_EXECUTE_OP
* FIELD_READ_OP
* FIELD_REFERENCE_OP
* FIELD_TYPE_OP
* FIELD_WRITE_OP
* GUARD_WRITE_OP
* SET_BOUNDS_OP
* SET_VARIANT_OP
* TIMED_CALL_OP

COND_CALL_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ENTRY_CALL_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FAMILY_CALL_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FAMILY_COND_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FAMILY_TIMED_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIELD_EXECUTE_OP

PURPOSE:

ON CLASS: PACKAGE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIELD_READ_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, MATRIX_CLASS, PACKAGE_CLASS, SUBARRAY_CLASS, SUBMATRIX_CLASS, SUBVECTOR_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIELD_REFERENCE_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, MATRIX_CLASS, PACKAGE_CLASS, RECORD_CLASS, SUBARRAY_CLASS, SUBMATRIX_CLASS, SUBVECTOR_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIELD_TYPE_OP

PURPOSE:

ON CLASS: RECORD_CLASS, VARIANT_RECORD_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

FIELD_WRITE_OP

PURPOSE:

ON CLASS: ARRAY_CLASS, MATRIX_CLASS, PACKAGE_CLASS, RECORD_CLASS, SELECT_CLASS, SUBARRAY_CLASS, SUBMATRIX_CLASS, SUBVECTOR_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GUARD_WRITE_OP

PURPOSE:

ON CLASS: SELECT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SEI_BOUNDS_OP

PURPOSE:

ON CLASS: VARIANT_RECORD_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SEI_VARIANT_OP

PURPOSE:

ON CLASS: VARIANT_RECORD_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

TIMED_CALL_OP

PURPOSE:

ON CLASS: TASK_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.9. LOGICAL_OPERATION

These operations provide primitive boolean actions.

LOGICAL_OPERATIONs include:

* AND_OP
* NOT_OP
* OR_OP
* XOR_OP

AND_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

OR_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

XOR_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.10. MEMBERSHIP_OPERATION

These operations provide primitive type checking actions.

MEMBERSHIP_OPERATIONs include:

* CHECK_IN_ROOT_TYPE_OP
* CHECK_IN_TYPE_OP
* IN_TYPE_OP
* NOT_IN_TYPE_OP

CHECK_IN_ROOT_TYPE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

CHECK_IN_TYPE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IN_TYPE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_IN_TYPE_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.11. MODULE_OPERATION

These operations provide primitive actions for manipulating modules and segments.

MODULE_OPERATIONs include:

* ABORT_OP
* ACTIVATE_OP
* AUGMENT_IMPORTS_OP
* CONTINUE_OP
* ELABORATE_OP
* GET_SUBUNIT_OP
* GET_SUBUNIT_COUNT_OP
* INSTRUCTION_READ_OP
* INSTRUCTION_WRITE_OP
* INTERRUPT_OP
* MAKE_ADDRESS_OP
* SET_SUBUNIT_OP
* SET_SUBUNIT_COUNT_OP
* WORD_WRITE_OP

ABORT_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ACTIVATE_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

AUGMENT_IMPORTS_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

CONTINUE_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

ELABORATE_OP

PURPOSE:

ON CLASS:

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GET_SUBUNIT_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GET_SUBUNIT_COUNT_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

INSTRUCTION_READ_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

INSTRUCTION_WRITE_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

INTERRUPT_OP

PURPOSE:

ON CLASS: MODULE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MAKE_ADDRESS_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SET_SUBUNIT_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SET_SUBUNIT_COUNT_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

WORD_WRITE_OP

PURPOSE:

ON CLASS: SEGMENT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.12. RANDOM_OPERATION

These operations define unique actions upon various operand classes.

RANDOM_OPERATIONs include:

* MAKE_ALIGNED_OP
* MAKE_CONSTANT_OP
* MAKE_CONSTRAINED_OP
* MAKE_VISIBLE_OP
* RENDEZVOUS_OP
* RUN_UTILITY_OP
* SET_CONSTRAINT_OP

MAKE_ALIGNED_OP

PURPOSE:

ON CLASS: ANY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MAKE_CONSTANT_OP

PURPOSE:

ON CLASS: ANY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MAKE_CONSTRAINED_OP

PURPOSE:

ON CLASS: VARIANT_RECORD_CLASS .

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

MAKE_VISIBLE_OP

PURPOSE:

ON CLASS: ANY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

RENDEZVOUS_OP

PURPOSE:

ON CLASS: ENTRY_CLASS, FAMILY_CLASS, SELECT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

RUN_UTILITY_OP

PURPOSE:

ON CLASS: ANY_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

SET_CONSTRAINT_OP

PURPOSE:

ON CLASS: VARIANT_RECORD_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.13. RANGE_OPERATION

These operations provide facilities for manipulating the range of a class.

RANGE_OPERATIONs include:

* ABOVE_RANGE_OP
* BELOW_RANGE_OP
* IN_RANGE_OP
* NOT_IN_RANGE_OP

ABOVE_RANGE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

BELOW_RANGE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IN_RANGE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_IN_RANGE_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

4.2.14. RELATIONAL_OPERATION

These operations provide primitive comparison actions.

RELATIONAL_OPERATIONs include:

* EQUAL_OP
* GREATER_OP
* GREATER_EQUAL_OP
* GREATER_EQUAL_ZERO_OP
* GREATER_ZERO_OP
* IS_ZERO_OP
* LESS_OP
* LESS_EQUAL_OP
* LESS_EQUAL_ZERO_OP
* LESS_ZERO_OP
* NOT_EQUAL_OP
* NOT_ZERO_OP

EQUAL_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ANY_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, RECORD_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GREATER_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GREATER_EQUAL_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GREATER_EQUAL_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

GREATER_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

IS_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LESS_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LESS_EQUAL_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS, FLOAT_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LESS_EQUAL_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

LESS_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_EQUAL_OP

PURPOSE:

ON CLASS: ACCESS_CLASS, ANY_CLASS, ARRAY_CLASS, DISCRETE_CLASS, FLOAT_CLASS, MATRIX_CLASS, RECORD_CLASS, VARIANT_RECORD_CLASS, VECTOR_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

NOT_ZERO_OP

PURPOSE:

ON CLASS: DISCRETE_CLASS

FUNCTION:

STACKS: Preconditions:

Postconditions:

EXCEPTIONS:

Chapter 5
DATA MOVEMENT INSTRUCTIONS

A data movement instruction provides facilities for setting, using, and referencing values of objects. Values may not be assigned indiscriminately to any object, but the architecture guarantees that values are compatible with the class of the target object.

Data movement instructions include the following opcodes:

* LOAD -- push the value of a named object on top of the CONTROL_STACK
* LOAD_TOP -- push the value of an object already on the CONTROL_STACK on top of the CONTROL_STACK
* STORE -- pop a value from the CONTROL_STACK and save it into named object
* REFERENCE -- push a reference to a named object on top of the CONTROL_STACK

In the case of the instructions LOAD, STORE, and REFERENCE, each requires the specification of an OBJECT_REFERENCE, which we may formally express as:

```
type OBJECT_REFERENCE(LEVEL : LEXICAL_LEVEL := 0) is
  record
    case LEVEL is
      when 0 .. 1 => SCOPE_OBJECT : SCOPE_DELTA;
      when others => FRAME_OFFSET : FRAME_DELTA;
    end case;
  end record;
```

The OBJECT_REFERENCE thus specifies an object declared on the IMPORT_STACK or outer frame (lex level 0 and 1 respectively) or an object declared in an inner frame. Formally, we may define these types as:

```
type FRAME_DELTA   is new INTEGER range implementation defined
type LEXICAL_LEVEL is new INTEGER range implementation defined
type SCOPE_DELTA   is new INTEGER range implementation defined
```

In the case of the LOAD_TOP instruction, an object on the CONTROL_STACK must be designated as a STACK_TOP_OFFSET. Formally, we have:

```
type STACK_TOP_OFFSET is new INTEGER range implementation defined
```

In the following sections, we treat each opcode in detail.

5.1. LOAD

The LOAD instruction pushes the value of a named object on top of the CONTROL_STACK. Formally, LOAD takes the form:

```
type LOAD_INSTRUCTION is
  record
    OBJECT : OBJECT_REFERENCE;
  end record;
```

PURPOSE: Push the value of a named object on top of the CONTROL_STACK.

FUNCTION: If the object resides on the IMPORT_STACK, then read the variable reference at the stated SCOPE_OFFSET, validating the display if necessary. If the object resides on some frame, then resolve the object reference at the stated FRAME_OFFSET, tracing through the display registers as needed. The final action in either case requires that the variable be made hidden, any renaming is traced back to the parent, and then the value is pushed on top of the CONTROL_STACK.

STACKS: Postcondition: A value is pushed on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the import display is not valid and the object resides on the IMPORT_STACK, or if the referenced object is not located at all. Finally, INSTRUCTION_ERROR is raised if the referenced object is not a TYPED_VAR, VALUE_VAR, SELECT_VAR, MODULE_KEY, EXCEPTION_VAR, or DELETION_KEY.

5.2. LOAD_TOP

The LOAD_TOP instruction pushes the value of an object on the CONTROL_STACK. Formally, LOAD_TOP takes the form:

```
type LOAD_TOP_INSTRUCTION is
  record
    AT_OFFSET : STACK_TOP_OFFSET;
  end record;
```

PURPOSE: Push the value of an object on the CONTROL_STACK.

FUNCTION: Trace through the reference on the CONTROL_STACK from the STACK_TOP_OFFSET to determine the site of the variable. If

the object resides on the IMPORT_STACK, then read the variable reference at the stated SCOPE_OFFSET, validating the display if necessary. If the object resides on some frame, then resolve the object reference at the stated FRAME_OFFSET, tracing through the display registers as needed. The final action in either case requires that the variable be made hidden, any renaming is traced back to the parent, and then the value is pushed on top of the CONTROL_STACK.

STACKS: Postcondition: A value is pushed on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the import display is not valid and the object resides on the IMPORT_STACK, or if the referenced object is not located at all. Finally, INSTRUCTION_ERROR is raised if the referenced object is not a TYPED_VAR, VALUE_VAR, SELECT_VAR, MODULE_KEY, EXCEPTION_VAR, or DELETION_KEY.

5.3. STORE

The store instruction pops a value off the CONTROL_STACK into a named object. Formally, STORE takes the form:

```
type STORE_INSTRUCTION is
  record
    OBJECT : OBJECT_REFERENCE;
  end record;
```

PURPOSE: Store a value from the CONTROL_STACK into a named object.

FUNCTION: If the object resides on the IMPORT_STACK, then read the variable reference at the stated SCOPE_OFFSET, validating the display if necessary. If the object resides on some frame, then resolve the object reference at the stated FRAME_OFFSET, tracing through the display registers as needed. Read the old value through this reference, and also read the new value from the top of the CONTROL_STACK. Copy the value over the old value, and pop the CONTROL_STACK by one value.

STACKS: Preconditions: A value must reside on top of the CONTROL_STACK.

Postconditions: A value is stored in the site of the referenced object. The CONTROL_STACK is popped by one.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the value to be stored is

not compatible with the target object. CAPABILITY_ERROR is raised of an attempt is made to store a value to a PACKAGE_VAR, TASK_VAR, or SEGMENT_VAR.

5.4. REFERENCE

The REFERENCE instruction pushes a reference to a named object on top of the CONTROL_STACK. Formally, REFERENCE takes the form:

```
type REFERENCE_INSTRUCTION is
  record
    OBJECT : OBJECT_REFERENCE;
  end record;
```

PURPOSE: Push a reference to a named object.

FUNCTION: If the object resides on the IMPORT_STACK, then read the variable reference at the stated SCOPE_OFFSET, validating the display if necessary. If the object resides on some frame, then resolve the object reference at the stated FRAME_OFFSET, tracing through the display registers as needed. Copy the reference to the object. If the object is an INDIRECT_VAR, VARIABLE_REF, or SUBPROGRAM_REF, make the reference hidden. Push the new reference on top of the CONTROL_STACK.

STACKS Postconditions: Push a reference on top of the CONTROL_STACK.

EXCEPTIONS :INSTRUCTION_ERROR is raised if the import display is not valid and the object resides on the IMPORT_STACK, or if the referenced object is not located at all. Finally, INSTRUCTION_ERROR is raised if the reference is not of the class VALUE_VAR, ENTRY_VAR, INTERFACE_KEY, INDIRECT_VAR, VARIABLE_REF, SUBPROGRAM_REF, or SUBPROGRAM_VAR.

Chapter 6
CONTROL TRANSFER INSTRUCTIONS

A control transfer instruction provides facilities for conditional an unconditional change in the thread of control. Briefly, these instructions provide simple branching and subprogram invocation.

Control transfer instructions include the following opcodes:

* CALL -- invoke a subprogram object
* JUMP -- branch unconditionally
* JUMP_NONZERO -- branch conditionally (if not zero)
* JUMP_ZERO -- branch conditionally (if zero)
* JUMP_CASE -- branch computed on the value of an expression

In the case of JUMP, JUMP_NONZERO, and JUMP_ZERO, the address of the branch is specified as a PC_OFFSET. Formally, PC_OFFSET is defined as:

```
type PC_OFFSET is new INTEGER range implementation defined;
```

In the case of the CALL instruction, the target module is designated by an OBJECT_REFERENCE as we defined in Section 5. Finally, the JUMP_CASE instruction branches to a part of a case statement, specified as some CASE_MAXIMUM, which we define formally as:

```
type CASE_MAXIMUM is new INTEGER range implementation defined;
```

In the following sections, we tread opcode in detail.

6.1. CALL

The CALL instruction invokes a subprogram object.

Formally, CALL takes the form:

```
type CALL_INSTRUCTION is
  record
     OBJECT : OBJECT.REFERENCE;
  end record;
```

PURPOSE: Invoke a subprogram object.

FUNCTION: Trace the OBJECT_REFERENCE to find the corresponding

SUBPROGRAM_VAR. If the object is not in the import space and the subprogram is active, mark the CONTROL_STACK to indicate the creation of a new frame, and transfer control to the first instruction of the SUBPROGRAM_VAR. If on the other hand the reference is to a SUBPROGRAM_REF, trace the reference through the CONTROL_STACK to determine the subprogram site; if the subprogram is active, mark the CONTROL_STACK to indicate the creation of a new frame, and transfer control to the first instruction of the referenced SUBPROGRAM_VAR.

STACKS: Postconditions: Add a new frame on the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raise if the SUBPROGRAM_VAR is not found. ELABORATION_ERROR is raised if the SUBPROGRAM_VAR is not active.

6.2. JUMP

The JUMP instruction causes an unconditional branch in the current thread of control.

Formally, JUMP is defined as:

```
type JUMP_INSTRUCTION is
  record
    RELATIVE : PC_OFFSET;
  end record;
```

PURPOSE: Cause an unconditional branch.

FUNCTION: Add the PC_OFFSET to the current program counter value.

STACKS No stacks are affected.

EXCEPTIONS: None.

6.3. JUMP_NONZERO

The JUMP_NONZERO instruction causes a branch only if the value on top of the CONTROL_STACK is not zero.

Formally, JUMP_NONZERO is defined as:

```
type JUMP_NONZERO is
  record
    RELATIVE : PC_OFFSET;
  end record;
```

PURPOSE: Cause a branch only if the value on top of the CONTROL_STACK is not zero.

FUNCTION: Pop a value off the CONTROL_STACK. If the DISCRETE_VAR has a nonzero value, add the PC_OFFSET to the current program counter value.

STACKS: Preconditions: Top of CONTROL_STACK must contain a DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the DISCRETE_VAR is not found. CAPABILITY_ERROR is raised if the DISCRETE_VAR is private.

.

6.4. JUMP_ZERO

The JUMP_ZERO instruction causes a branch only if the value on top of the CONTROL_STACK is zero.

Formally, JUMP_ZERO is defined as:

```
type JUMP_ZERO is
  record
    RELATIVE : PC_OFFSET;
  end record;
```

PURPOSE: Cause a branch only if the value on top of the CONTROL_STACK is zero.

FUNCTION: Pop a value off the CONTROL_STACK. If the DISCRETE_VAR has a zero value, add the PC_OFFSET to the current program counter value.

STACKS: Preconditions: Top of CONTROL_STACK must contain a DISCRETE_VAR.

Postconditions: Top of CONTROL_STACK is reduced by one.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the DISCRETE_VAR is not found. CAPABILITY_ERROR is raised if the DISCRETE_VAR is private.

.

6.5. JUMP_CASE

The JUMP_CASE instruction causes a branch computed on the value of an expression.

Formally, JUMP_CASE is defined as:

```
type JUMP_CASE is
  record
    CASE_MAX : CASE_MAXIMUM;
  end record;
```

PURPOSE: Cause a branch computed on the value of an expression.

FUNCTION: Pop a value off the CONTROL_STACK. Add the offset to the current program counter, which will now point to another unconditional branch instruction.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a DISCRETE_VAR.

Postconditions: Top of the CONTROL_STACK is reduced by one.

EXCEPTIONS: CONSTRAINT_ERROR is raised if the DISCRETE_VAR value is not between 0 and the value of CASE_MAX.

Chapter 7
CONTROL RETURN INSTRUCTIONS

A control return instruction provides facilities for return from a subordinate thread of control. It is important to note that these instructions follow the semantics of Ada for return of control, in that unit may not be exited until all dependencies with subordinate units are resolved.

Control return instruction include the following opcodes:

* EXIT_PROCEDURE -- return from a procedure
* EXIT_FUNCTION -- return from a function
* EXIT_ACCEPT -- return from an accept clause
* EXIT_UTILITY -- return from a utility subprogram
* POP_BLOCK -- return from a block
* POP_BLOCK_RESULT -- return from a block with a result

In the case of the instructions EXIT_PROCEDURE, EXIT_FUNCTION, EXIT_ACCEPT, and EXIT_UTILITY, the amount by which the CONTROL_STACK is popped is specified as an INNER_FRAME_DELTA, which we formally express as:

```
type INNER_FRAME_DELTA is new INTEGER range implementation defined;
```

In the case of the instructions POP_BLOCK and POP_BLOCK_RESULT, the mark to which the CONTROL_STACK is popped is specified as a TARGET_LEX, which we formally express as:

```
type TARGET_LEX is new INTEGER range implementation defined;
```

In the following sections, we treat each opcode in detail.

7.1. EXIT_PROCEDURE

The EXIT_PROCEDURE instruction returns the thread of control from a procedure.

Formally, EXIT_PROCEDURE is defined as:

```
type EXIT_PROCEDURE_INSTRUCTION is
  record
    POP_AMOUNT : INNER_BLOCK_DELTA;
  end record;
```

PURPOSE: Return from a procedure.

FUNCTION: If the frame has no children, then simply pop the frame over the number of parameters indicated in POP_AMOUNT. If the frame does have children, then if all the children are completed, iteratively deallocate each child, and then deallocate the current module; if all the children are not complete, then set the current state of the module as waiting for children, and set the micro state of the module to repeat the current instruction; finally, then command a context swap.

STACKS: Postconditions: CONTROL_STACK micro state and child paths will be altered due to the deallocation; upon completion, the CONTROL_STACK will pop to the previous frame.

EXCEPTIONS: None.

7.2. EXIT_FUNCTION

The EXIT_FUNCTION instruction returns the thread of control from a function.

Formally^, EXIT_FUNCTION is defined as:

```
type EXIT_FUNCTION_INSTRUCTION is
  record
    POP_AMOUNT : INNER_BLOCK_DELTA;
  end record;
```

PURPOSE: Return from a function.

FUNCTION: If the frame has no children, then first copy the function result from the top of the CONTROL_STACK, pop the frame over the number of parameters indicated in POP_AMOUNT, then push the result back on the CONTROL_STACK. If the frame does have children, then if all the children are completed, iteratively deallocate each child, and then copy the result, deallocate the current module and push the result back on the CONTROL_STACK; if all the children are not complete, then set the current state of the module as waiting for children, and set the micro state of the module to repeat the current instruction; finally, then command a context swap.

STACKS: Postconditions: CONTROL_STACK micro state and child paths will be altered due to the deallocation; upon completion, the CONTROL_STACK will pop to the previous frame.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the result on top of the CONTROL_STACK is not a typed value. TYPE_ERROR is raised if the type of the result does not persist after potential deallocation.

7.3. EXIT_ACCEPT

The EXIT_ACCEPT instruction returns the thread of control from an accept clause.

Formally, EXIT_ACCEPT is defined as:

```
type EXIT_ACCEPT_INSTRUCTION is
  record
    POP_AMOUNT : INNER_BLOCK_DELTA;
  end record;
```

PURPOSE: Return from an accept clause.

FUNCTION: If the frame has now children, or if the deallocation of the children is complete then complete the following: Pop the frame over zero parameters, and then read the current ACCEPT_LINK on top of the CONTROL_STACK. Pop the CONTROL_STACK over the POP_AMOUNT, and then pop down over all the out parameters. This last action requires that the out values be copied to a block. If the accept clause is part of a select statement, then push a DISCRETE_VAR on top of the CONTROL_STACK indicating which accept of the select has been exited. Send a message to the accept link partner indicating END_RENDEZVOUS, and include the out values copied to a block.

STACKS: Preconditions: Top of the CONTROL_STACK must contain an ACCEPT_LINK.

Postconditions: CONTROL_STACK is popped to below all the accept parameters. A DISCRETE_VAR may be pushed on top of the CONTROL_STACK. Stacks are also affected through normal message transmission.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the out parameters are not typed values.

7.4. EXIT_UTILITY

The EXIT_UTILITY instruction returns the thread of control from a utility subprogram.

Formally, EXIT_UTILITY is defined as:

```
type EXIT_UTILITY_INSTRUCTION is
  record
    POP_AMOUNT : INNER_BLOCK_DELTA;
  end record;
```

PURPOSE: Return from a utility subprogram.

FUNCTION: Simply pop the frame over the number of parameters indicated in POP_AMOUNT.

STACKS: The CONTROL_STACK will pop to the previous frame.

EXCEPTIONS: None.

7.5. POP_BLOCK

The POP_BLOCK instruction returns the thread of control from a block.

Formally, POP_BLOCK is defined as:

```
type POP_BLOCK_INSTRUCTION is
  record
    TO_LEVEL : TARGET_LEX;
  end record;
```

PURPOSE: Return from a block.

FUNCTION: Check that a pop to the target level is legal. If the frame does not have any children, or if the deallocation of the children is completed, then repeatedly pop the CONTROL_STACK until the target lex level is reached.

STACKS: Postconditions: The CONTROL_STACK is popped to the target lexical level.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the target level does not exist.

7.6. POP_BLOCK_RESULT

The POP_BLOCK_RESULT instruction returns the thread of control from a block with a result.

Formally, POP_BLOCK_RESULT is defined as:

```
type POP_BLOCK_INSTRUCTION is
  record
    TO_LEVEL : TARGET_LEX;
  end record;
```

PURPOSE: Return from a block.

FUNCTION: Copy the result from the top of the CONTROL_STACK. Check that a pop to the target level is legal. If the frame does not have any children, or if the deallocation of the children is completed, then repeatedly pop the CONTROL_STACK until the target lex level is reached. Finally, push the result back on top of the CONTROL_STACK.

STACKS: Preconditions: Top of the CONTROL_STACK must contain a typed value.

Postconditions: The CONTROL_STACK is popped to the target lexical level. A copy of the initial typed value is pushed back on top of the CONTROL_STACK.

EXCEPTIONS: INSTRUCTION_ERROR is raised if the target level does not exist. OPERAND_CLASS_ERROR is raised if the result on top of the CONTROL_STACK is not a typed value. TYPE_ERROR is raised if the type of the result does not persist after potential deallocation.

Chapter 8
LITERAL DECLARATIONS

A literal declaration instruction defines simple and complex literal values for use on the CONTROL_STACK.

Literal declaration instruction include the following opcodes:

* LITERAL_VALUE -- marks a discrete, floating point, or array value
* SHORT_LITERAL -- pushes a short discrete value
* INDIRECT_LITERAL -- pushes a referenced LITERAL_VALUE

In the following section, we treat each opcode in detail.

8.1. LITERAL_VALUE

The LITERAL_VALUE instruction is not executable, but rather serves to mark a discrete, floating point, or array value.

Formally, LITERAL_VALUE is defined as:

```
type LITERAL_VALUE is
  record
    VALUE : LITERAL;
  end record;
```

We further define LITERAL as:

```
type LITERAL(OF_KIND : OPERAND_CLASS := DISCRETE_CLASS) is
  record
    case OF_KIND is
      when ARRAY_CLASS    =>
             ARRAY_LITERAL    : BASE.ARRAY_LITERAL;
      when DISCRETE_CLASS =>
             DISCRETE_LITERAL : BASE.DISCRETE;
      when FLOAT_CLASS    =>
             FLOAT_LITERAL    : BASE.FLOAT;
    end case;
  end record;
```

The type of each BASE literal is implementation defined.

8.2. SHORT_LITERAL

The SHORT_LITERAL instruction pushes a short literal on top of the CONTROL_STACK.

Formally, the SHORT_LITERAL instruction is defined as:

```
type SHORT_LITERAL_INSTRUCTION is
  record
    SHORT_VALUE : BASE.SHORT_LITERAL;
  end record;
```

The type of the BASE literal is implementation defined.

PURPOSE: Push a short literal on the CONTROL_STACK.

FUNCTION: Push the SHORT_VALUE on the CONTROL_STACK.

STACKS: Postcondition: A DISCRETE_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: None.

8.3. INDIRECT_LITERAL

The INDIRECT_LITERAL instruction pushes a referenced LITERAL_VALUE on the CONTROL_STACK.

Formally, INDIRECT_LITERAL is defined as:

```
type INDIRECT_LITERAL_INSTRUCTION is
  record
    VALUE_CLASS    : OPERAND_CLASS;
    VALUE_RELATIVE : PC_OFFSET;
  end record;
```

The type PC_OFFSET is defined in Chapter 6.

PURPOSE: Push a LITERAL_VALUE on the CONTROL_STACK.

FUNCTION: Take the VALUE_RELATIVE to reference a LITERAL_VALUE. Follow the reference, and push the corresponding value on the CONTROL_STACK.

STACKS: Postcondition: A DISCRETE_VAR, FLOAT_VAR, or ARRAY_VAR is pushed on top of the CONTROL_STACK.

EXCEPTIONS: OPERAND_CLASS_ERROR is raised if the reference does not lead to a LITERAL_VALUE.

Chapter 9
MODULE LABELS

Module labels are not executable instructions, but rather serve to mark structures in a given module.

Module labels include the following opcodes:

* SEGMENT_HEADER -- define a list of segment names
* SEGMENT_TYPE -- define the name of a current segment
* SEGMENT_VALUE -- define the location of a segment
* BLOCK_BEGIN -- mark the start of code of a segment
* BLOCK_HANDLER -- mark the segment exception handler
* END_LOCALS -- mark the end of local entities

In the following sections, we treat each opcode in detail.

9.1. SEGMENT_HEADER

The SEGMENT_HEADER defines a list of all segment names defined within the current segment.

Formally, SEGMENT_HEADER takes the form:

```
type SEGMENT_HEADER_INSTRUCTION is
  record
    DESCRIPTION : SEGMENTS.HEADER;
  end record;
```

SEGMENT.HEADER is an implementation defined list of segment names.

9.2. SEGMENT_TYPE

The SEGMENT_TYPE defines the name of a segment.

Formally, SEGMENT_TYPE is defined as:

```
type SEGMENT_TYPE is
  record
    TYPE_NAME : SEGMENTS.TYPE_NAME;
  end record;
```

SEGMENTS.TYPE_NAME is a unique implementation defined name.

9.3. SEGMENT_VALUE

The SEGMENT_VALUE defines the address of a segment.

Formally, SEGMENT_VALUE is defined as:

```
type SEGMENT_VALUE is
  record
    MODULE_START : SEGMENTS.ADDRESS;
  end record;
```

SEGMENTS.ADDRESS is an implementation defined address.

9.4. BLOCK_BEGIN

The BLOCK_BEGIN marks the start of the executable code for a segment.

Formally, BLOCK_BEGIN is defined as:

```
type BLOCK_BEGIN is
  record
    LOCATION : SEGMENTS.REFERENCE;
  end record;
```

SEGMENTS.REFERENCE is an implementation defined label.

9.5. BLOCK_HANDLER

The BLOCK_HANDLER marks the location of the exception handler for the segment.

Formally, BLOCK_HANDLER is defined as:

```
type BLOCK_HANDLER is
  record
    LOCATION : SEGMENTS.REFERENCE;
  end record;
```

SEGMENTS.REFERENCE is an implementation defined label.

9.6. END_LOCALS

END_LOCALS marks the end of any locally declared entities on the CONTROL_STACK.

Formally, END_LOCALS is defined as:

```
type END_LOCALS is
  record
    OFFSET : SCOPE_DELTA;
  end record;
```

SCOPE_DELTA is defined in Chapter 5.

Appendix A
INSTRUCTION SET SUMMARY

This appendix provides a formal definition of the Rational Machines instruction set. The packages BASE and SEGMENT are omitted since they define implementation dependencies; their implementation may be found in a corresponding processor reference manual.

```
with BASE,
     SEGMENTS;

package CODE is

  VERSION : constant := 1;

  type OP_CODE is
        (ACTION,          BLOCK_BEGIN,       BLOCK_HANDLER,
         CALL,            COMPLETE_TYPE,     DECLARE_SUBPROGRAM,
         DECLARE_TYPE,    DECLARE_VARIABLE,  END_LOCALS,
         EXECUTE,         EXIT_ACCEPT,       EXIT_FUNCTION,
         EXIT_PROCEDURE,  EXIT_UTILITY,      INDIRECT_LITERAL,
         JUMP,            JUMP_CASE,         JUMP_NONZERO,
         JUMP_ZERO,       LITERAL_VALUE,     LOAD,
         LOAD_TOP,        POP_BLOCK,         POP_BLOCK_RESULT,
         REFERENCE,       SEGMENT_HEADER,    SEGMENT_TYPE,
         SEGMENT_VALUE,   SHORT_LITERAL,     STORE);

  type OPERAND_CLASS is
        (ACCESS_CLASS,          ANY_CLASS,         ARRAY_CLASS,
         DISCRETE_CLASS,        ENTRY_CLASS,       EXCEPTION_CLASS,
         FAMILY_CLASS,          FLOAT_CLASS,       MATRIX_CLASS,
         MODULE_CLASS,          PACKAGE_CLASS,     RECORD_CLASS,
         SEGMENT_CLASS,         SELECT_CLASS,      SUBARRAY_CLASS,
         SUBMATRIX_CLASS,       SUBVECTOR_CLASS,   TASK_CLASS,
         VARIANT_RECORD_CLASS,  VECTOR_CLASS);

  type OPERATION is
        ABORT_OP,                ABOVE_RANGE_OP,          ABSOLUTE_VALUE_OP,
        ACTIVATE_OP,             ADDRESS_OP,              ALL_READ_OP,
        ALL_REFERENCE_OP,        ALL_WRITE_OP,            AND_OP,
        APPEND_OP,               AUGMENT_IMPORTS_OP,      BELOW_RANGE_OP,
        BOUNDS_CHECK_OP,         BOUNDS_OP,               CATENATE_OP,
        CHECK_IN_ROOT_TYPE_OP,   CHECK_IN_TYPE_OP,        COND_CALL_OP,
        CONTINUE_OP,             CONVERT_ACTUAL_OP,       CONVERT_OP,
        COUNT_OP,                DECREMENT_OP,            DIVIDE_OP,
        ELABORATE_OP,            ELEMENT_TYPE_OP,         ENTRY_CALL_OP,
        EQUAL_OP,                FAMILY_CALL_OP,          FAMILY_COND_OP,
        FAMILY_TIMED_OP,         FIELD_EXECUTE_OP,        FIELD_READ_OP,
        FIELD_REFERENCE_OP,      FIELD_TYPE_OP,           FIELD_WRITE_OP,
        FIRST_OP,                GET_SUBUNIT_OP,          GET_SUBUNIT_COUNT_OP
        GREATER_EQUAL_OP,        GREATER_EQUAL_ZERO_OP,   GREATER_OP,
```

```
        GREATER_ZERO_OP,        GUARD_WRITE_OP,         IN_RANGE_OP,
        IN_TYPE_OP,             INCREMENT_OP,           INSTRUCTION_READ_OP,
        INSTRUCTION_WRITE_OP,   INTERRUPT_OP,           IS_CALLABLE_OP,
        IS_CONSTRAINED_OP,      IS_NULL_OP,             IS_TERMINATED_OP,
        IS_ZERO_OP,             LAST_OP,                LENGTH_OP,
        LESS_EQUAL_OP,          LESS_EQUAL_ZERO_OP,     LESS_OP,
        LESS_ZERO_OP,           MAKE_ADDRESS_OP,        MAKE_ALIGNED_OP,
        MAKE_CONSTANT_OP,       MAKE_CONSTRAINED_OP,    MAKE_VISIBLE_OP,
        MINUS_OP,               MODULO_OP,              NAME_OP,
        NOT_EQUAL_OP,           NOT_IN_RANGE_OP,        NOT_IN_TYPE_OP,
        NOT_NULL_OP,            NOT_OP,                 NOT_ZERO_OP,
        OR_OP,                  PLUS_OP,                PREDECESSOR_OP,
        PREPEND_OP,             RAISE_OP,               REMAINDER_OP,
        RENDEZVOUS_OP,          REVERSE_BOUNDS_OP,      RUN_UTILITY_OP,
        SCOPE_OF_RAISE_OP,      SET_BOUNDS_OP,          SET_CONSTRAINT_OP,
        SET_SUBUNIT_OP,         SET_SUBUNIT_COUNT_OP,   SET_VARIANT_OP,
        SIZE_OP,                SLICE_READ_OP,          SLICE_WRITE_OP,
        SUBARRAY_OP,            SUCCESSOR_OP,           TIMED_CALL_OP,
        TIMES_OP,               UNARY_MINUS_OP,         XOR_OP,
        WORD_WRITE_OP);

type UNCLASSED_ACTION is
        ACCEPT_ACTIVATION,      ACTIVATE_TASKS,         ACTIVATE_SUBPROGRAM,
        ALTER_BREAK_MASK,       BREAK_OPTIONAL,         BREAK_UNCONDITIONAL,
        CALL_IMPORT,            CALL_REFERENCE,         DELETE_ITEM,
        DELETE_SUBPROGRAM,      ESTABLISH_FRAME,        EXIT_BREAK,
        IDLE,                   ILLEGAL,                INITIATE_DELAY,
        INTRODUCE_IMPORT,       MAKE_NULL_UTILITY,      MAKE_SELF,
        MARK_AUXILIARY,         MARK_DATA,              MARK_TYPE,
        NAME_MODULE,            NAME_PARTNER,           OVERWRITE_IMPORT,
        POP_AUXILIARY,          POP_CONTROL,            POP_DATA,
        POP_TYPE,               PROPAGATE_ABORT,        QUERY_BREAK_ADDRESS,
        QUERY_BREAK_CAUSE,      QUERY_BREAK_MASK,       QUERY_FRAME,
        QUERY_RESOURCE_LIMITS,  QUERY_RESOURCE_STATE,   READ_IMPORT,
        RECOVER_RESOURCES,      REFERENCE_IMPORT,       REMOVE_IMPORT,
        RETURN_RESOURCES,       SET_BREAK_MASK,         SET_INTERFACE_SCOPE,
        SET_INTERFACE_SUBPROG,  SET_NULL_ACCESS,        SET_RESOURCE_LIMITS,
        SET_VISIBILITY,         SIGNAL_ACTIVATED,       SIGNAL_COMPLETION,
        SWAP_CONTROL,           WRITE_IMPORT);

type ELABORATION_STATE is (ACTIVE, INACTIVE, UNSPECIFIED);

type LITERAL (OF_KIND : OPERAND_CLASS := DISCRETE_CLASS) is
      record
        case OF_KIND is
              when ARRAY_CLASS    =>
                ARRAY_LITERAL    : BASE.ARRAY_LITERAL;
              when DISCRETE_CLASS =>
                DISCRETE_LITERAL : BASE.DISCRETE;
              when FLOAT_CLASS    =>
                FLOAT_LITERAL    : BASE.REAL;
              when others =>
```

```
                    null;
            end case;
          end record;

  type SUBPROGRAM_SORT is
         (FOR_ACCEPT,    FOR_CALL,
          FOR_INTERFACE, FOR_OUTER_CALL,
          FOR_UTILITY);

  type TYPE_COMPLETION_MODE is
         (BY_COMPONENT_COMPLETION, BY_CONSTRANING,
          BY_DEFINING,             BY_DERIVING);
  type TYPE_OPTION_SET is
         record
           BOUNDS_WITH_OBJECT : BOOLEAN;
           CONSTRAINED        : BOOLEAN;
           DERIVED_PRIVATE    : BOOLEAN;
           UNSIGNED           : BOOLEAN;
           WITH_ENTRYS        : BOOLEAN;
         end record;
  type TYPE_PRIVACY is (IS_LOCAL, IS_PRIVATE, IS_PUBLIC);
  type TYPE_SORT is
         (CONSTRAINED, CONSTRAINED_INCOMPLETE,
          DEFINED,     DEFINED_INCOMPLETE,
          DERIVED,     DERIVED_INCOMPLETE,
          INCOMPLETE);

  type VARIABLE_OPTION_SET is
         record
           BY_ALLOCATION   : BOOLEAN;
           DATA_TASK       : BOOLEAN;
           DISTRIBUTOR     : BOOLEAN;
           DUPLICATE       : BOOLEAN;
           HEAP_TASK       : BOOLEAN;
           UNCHECKED       : BOOLEAN;
           WITH_CONSTRAINT : BOOLEAN;
           WITH_SUBTYPE    : BOOLEAN;
           WITH_VALUE      : BOOLEAN;
         end record;

  type VISIBILITY is (DEFAULT, IS_HIDDEN, IS_VISIBLE);

  type CASE_MAXIMUM      is new INTEGER   range implementation_defined;
  type FIELD_INDEX       is new INTEGER   range implementation_defined;
  type FRAME_DELTA       is new INTEGER   range implementation_defined;
  type INNER_FRAME_DELTA is new INTEGER   range implementation_defined;
  type LEXICAL_LEVEL     is new INTEGER   range implementation_defined;
  type PC_OFFSET         is new INTEGER   range implementation_defined;
  type SCOPE_DELTA       is new INTEGER   range implementation_defined;
  type STACK_TOP_OFFSET  is new INTEGER   range implementation_defined;
  subtype TARGET_LEX     is LEXICAL_LEVEL range implementation_defined;
  subtype VARIANT_RECORD_INDEX is FIELD_INDEX range  1 .. FIELD_INDEX'LAST;
```

```
type FIELD_SORT  is (FIXED, VARIANT);
type FIELD_MODE  is (DIRECT, INDIRECT);

type ACCESS_SPEC (OP : OPERATION := EQUAL_OP) is
       record
         case OP is
                when COMPONENT_OPERATIONS =>
                  KIND : FIELD_SORT;
                  MODE : FIELD_MODE;
                when others               =>
                  null;
         end case;
       end record;

type FIELD_SPEC (CLASS : OPERAND_CLASS := DISCRETE_CLASS;
                 OP    : OPERATION     := EQUAL_OP) is
       record
         case CLASS is
                when PACKAGE_CLASS | TASK_CLASS =>
                  OFFSET : FIELD_INDEX;
                when RECORD_CLASS | SELECT_CLASS =>
                  NUMBER : FIELD_INDEX;
                when VARIANT_RECORD_CLASS =>
                  COMPONENT : ACCESS_SPEC (OP);
                  INDEX     : VARIANT_RECORD_INDEX;
                when others =>
                  null;
         end case;
       end record;

type OPERATOR_SPEC (CLASS : OPERAND_CLASS := DISCRETE_CLASS;
                    OP    : OPERATION     := EQUAL_OP) is
       record
         case OP is
                when FIELD_OPERATIONS =>
                  FIELD : FIELD_SPEC (CLASS, OP);
                when others           =>
                  null;
         end case;
       end record;


type OBJECT_REFERENCE (LEVEL : LEXICAL_LEVEL := 0) is
       record
         case LEVEL is
                when 0 .. 1 => SCOPE_OFFSET : SCOPE_DELTA;
               when others => FRAME_OFFSET : FRAME_DELTA;
         end case;
       end record;

type INSTRUCTION (FOR_OP : OP_CODE := ACTION) is
```

```
record
  case FOR_OP is
        when ACTION                          =>
          TO_PERFORM              : UNCLASSED_ACTION;
        when BLOCK_BEGIN | BLOCK_HANDLER     =>
          LOCATION                : SEGMENTS.REFERENCE;
        when CALL | LOAD | REFERENCE | STORE =>
          OBJECT                  : OBJECT_REFERENCE;
        when COMPLETE_TYPE                   =>
          COMPLETION_CLASS        : OPERAND_CLASS;
          COMPLETION_MODE         : TYPE_COMPLETION_MODE;
        when DECLARE_SUBPROGRAM              =>
          SUBPROGRAM_KIND         : SUBPROGRAM_SORT;
          SUBPROGRAM_STATE        : ELABORATION_STATE;
          SUBPROGRAM_VISIBILITY   : VISIBILITY;
        when DECLARE_TYPE                    =>
          PRIVACY                 : TYPE_PRIVACY;
          TYPE_CLASS              : OPERAND_CLASS;
          TYPE_KIND               : TYPE_SORT;
          TYPE_OPTIONS            : TYPE_OPTION_SET;
        when DECLARE_VARIABLE                =>
          VARIABLE_CLASS          : OPERAND_CLASS;
          VARIABLE_OPTIONS        : VARIABLE_OPTION_SET;
          VARIABLE_VISIBILITY     : VISIBILITY;
        when END_LOCALS                      =>
          OFFSET                  : SCOPE_DELTA;
        when EXECUTE                         =>
          OPERATOR                : OPERATOR_SPEC;
        when EXIT_ACCEPT     | EXIT_FUNCTION |
             EXIT_PROCEDURE | EXIT_UTILITY   =>
          POP_AMOUNT              : INNER_FRAME_DELTA;
        when INDIRECT_LITERAL                =>
          VALUE_CLASS             : OPERAND_CLASS;
          VALUE_RELATIVE          : PC_OFFSET;
        when JUMP | JUMP_NONZERO | JUMP_ZERO =>
          RELATIVE                : PC_OFFSET;
        when JUMP_CASE                       =>
          CASE_MAX                : CASE_MAXIMUM;
        when LITERAL_VALUE                   =>
          VALUE                   : LITERAL;
        when  LOAD_TOP                       =>
          AT_OFFSET               : STACK_TOP_OFFSET;
        when POP_BLOCK | POP_BLOCK_RESULT    =>
          TO_LEVEL                : TARGET_LEX;
        when SEGMENT_HEADER                  =>
          DESCRIPTOR              : SEGMENTS.HEADER;
        when SEGMENT_TYPE                    =>
          TYPE_NAME               : SEGMENTS.TYPE_NAME;
        when SEGMENT_VALUE                   =>
          MODULE_START            : SEGMENTS.ADDRESS;
        when SHORT_LITERAL                   =>
          SHORT_VALUE             : BASE.SHORT_LITERAL;
```

```
            end case;
          end record;

  type WORD    is array (SEGMENTS.INSTRUCTION_INDEX)      of INSTRUCTION;
  type SEGMENT is array (SEGMENTS.DISPLACEMENT range <>) of WORD;
end CODE;
```

Appendix B
OBJECT/OPERATION CROSS-REFERENCE

This appendix lists each class of objects recognized by the architecture and the instructions that operate upon them.

B.1. CLASSED INSTRUCTIONS

ACCESS_CLASS

COMPLETE_TYPE,ACCESS,BY_DEFINING
COMPLETE_TYPE,ACCESS,BY_DERIVING
COMPLETE_TYPE,ACCESS,BY_CONSTRAINING

DECLARE_TYPE,ACCESS,CONSTRAINED,LOCAL
DECLARE_TYPE,ACCESS,CONSTRAINED,PRIVATE
DECLARE_TYPE,ACCESS,CONSTRAINED,PUBLIC
DECLARE_TYPE,ACCESS,DEFINED,LOCAL
DECLARE_TYPE,ACCESS,DEFINED,PRIVATE
DECLARE_TYPE,ACCESS,DEFINED,PUBLIC
DECLARE_TYPE,ACCESS,DERIVED,LOCAL
DECLARE_TYPE,ACCESS,DERIVED,PRIVATE
DECLARE_TYPE,ACCESS,DERIVED,PUBLIC
DECLARE_TYPE,ACCESS,INCOMPLETE,LOCAL
DECLARE_TYPE,ACCESS,INCOMPLETE,PRIVATE
DECLARE_TYPE,ACCESS,INCOMPLETE,PUBLIC

DECLARE_VARIABLE,ACCESS,DUPLICATE
DECLARE_VARIABLE,ACCESS,HIDDEN
DECLARE_VARIABLE,ACCESS,HIDDEN,BY_ALLOCATION
DECLARE_VARIABLE,ACCESS,HIDDEN,BY_ALLOCATION,WITH_CONSTRAINT
DECLARE_VARIABLE,ACCESS,HIDDEN,BY_ALLOCATION,WITH_SUBTYPE
DECLARE_VARIABLE,ACCESS,HIDDEN,BY_ALLOCATION,WITH_VALUE
DECLARE_VARIABLE,ACCESS,VISIBLE
DECLARE_VARIABLE,ACCESS,VISIBLE,BY_ALLOCATION
DECLARE_VARIABLE,ACCESS,VISIBLE,BY_ALLOCATION,WITH_CONSTRAINT
DECLARE_VARIABLE,ACCESS,VISIBLE,BY_ALLOCATION,WITH_SUBTYPE
DECLARE_VARIABLE,ACCESS,VISIBLE,BY_ALLOCATION,WITH_VALUE

EXECUTE,ACCESS,ALL_READ
EXECUTE,ACCESS,ALL_REFERENCE
EXECUTE,ACCESS,ALL_WRITE
EXECUTE,ACCESS,CHECK_IN_TYPE
EXECUTE,ACCESS,CONVERT
EXECUTE,ACCESS,ELEMENT_TYPE
EXECUTE,ACCESS,EQUAL
EXECUTE,ACCESS,IN_TYPE
EXECUTE,ACCESS,IS_NULL
EXECUTE,ACCESS,NOT_EQUAL
EXECUTE,ACCESS,NOT_IN_TYPE

```
  EXECUTE,ACCESS,NOT_NULL

ANY_CLASS

  DECLARE_TYPE,ANY,DERIVED,LOCAL
  DECLARE_TYPE,ANY,DERIVED,PRIVATE
  DECLARE_TYPE,ANY,DERIVED,PUBLIC

  DECLARE_VARIABLE,ANY,HIDDEN
  DECLARE_VARIABLE,ANY,VISIBLE

  EXECUTE,ANY,ADDRESS
  EXECUTE,ANY,CONVERT
  EXECUTE,ANY,CONVERT_ACTUAL
  EXECUTE,ANY,EQUAL
  EXECUTE,ANY,MAKE_ALIGNED
  EXECUTE,ANY,MAKE_CONSTANT
  EXECUTE,ANY,MAKE_VISIBLE
  EXECUTE,ANY,NOT_EQUAL
  EXECUTE,ANY,RUN_UTILITY
  EXECUTE,ANY,SIZE

ARRAY_CLASS

  COMPLETE_TYPE,ARRAY,BY_COMPONENT_COMPLETION
  COMPLETE_TYPE,ARRAY,BY_CONSTRAINING
  COMPLETE_TYPE,ARRAY,BY_DEFINING
  COMPLETE_TYPE,ARRAY,BY_DERIVING

  DECLARE_TYPE,ARRAY,CONSTRAINED,LOCAL
  DECLARE_TYPE,ARRAY,CONSTRAINED,LOCAL,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,CONSTRAINED,PRIVATE
  DECLARE_TYPE,ARRAY,CONSTRAINED,PRIVATE,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,CONSTRAINED,PUBLIC
  DECLARE_TYPE,ARRAY,CONSTRAINED,PUBLIC,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,LOCAL
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,LOCAL,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,PRIVATE,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,PUBLIC
  DECLARE_TYPE,ARRAY,CONSTRAINED_INCOMPLETE,PUBLIC,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED,LOCAL
  DECLARE_TYPE,ARRAY,DEFINED,LOCAL,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED,PRIVATE
  DECLARE_TYPE,ARRAY,DEFINED,PRIVATE,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED,PUBLIC
  DECLARE_TYPE,ARRAY,DEFINED,PUBLIC,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,LOCAL
  DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,LOCAL,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,PRIVATE,BOUNDS_WITH_OBJECT
  DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,PUBLIC
```

DECLARE_TYPE,ARRAY,DEFINED_INCOMPLETE,PUBLIC,BOUNDS_WITH_OBJECT
DECLARE_TYPE,ARRAY,DERIVED,LOCAL
DECLARE_TYPE,ARRAY,DERIVED,PRIVATE
DECLARE_TYPE,ARRAY,DERIVED,PUBLIC
DECLARE_TYPE,ARRAY,DERIVED_INCOMPLETE,LOCAL
DECLARE_TYPE,ARRAY,DERIVED_INCOMPLETE,PRIVATE
DECLARE_TYPE,ARRAY,DERIVED_INCOMPLETE,PUBLIC
DECLARE_TYPE,ARRAY,INCOMPLETE,LOCAL
DECLARE_TYPE,ARRAY,INCOMPLETE,LOCAL,BOUNDS_WITH_OBJECT
DECLARE_TYPE,ARRAY,INCOMPLETE,PRIVATE
DECLARE_TYPE,ARRAY,INCOMPLETE,PRIVATE,BOUNDS_WITH_OBJECT
DECLARE_TYPE,ARRAY,INCOMPLETE,PUBLIC
DECLARE_TYPE,ARRAY,INCOMPLETE,PUBLIC,BOUNDS_WITH_OBJECT

DECLARE_VARIABLE,ARRAY,DUPLICATE
DECLARE_VARIABLE,ARRAY,HIDDEN
DECLARE_VARIABLE,ARRAY,HIDDEN,UNCHECKED
DECLARE_VARIABLE,ARRAY,HIDDEN,WITH_CONSTRAINT
DECLARE_VARIABLE,ARRAY,VISIBLE
DECLARE_VARIABLE,ARRAY,VISIBLE,UNCHECKED
DECLARE_VARIABLE,ARRAY,VISIBLE,WITH_CONSTRAINT

EXECUTE,ARRAY,BOUNDS
EXECUTE,ARRAY,CHECK_IN_TYPE
EXECUTE,ARRAY,CONVERT
EXECUTE,ARRAY,CONVERT_ACTUAL
EXECUTE,ARRAY,ELEMENT_TYPE
EXECUTE,ARRAY,EQUAL
EXECUTE,ARRAY,FIELD_READ
EXECUTE,ARRAY,FIELD_REFERENCE
EXECUTE,ARRAY,FIELD_WRITE
EXECUTE,ARRAY,FIRST
EXECUTE,ARRAY,IN_TYPE
EXECUTE,ARRAY,LAST
EXECUTE,ARRAY,LENGTH
EXECUTE,ARRAY,NOT_EQUAL
EXECUTE,ARRAY,NOT_IN_TYPE
EXECUTE,ARRAY,REVERSE_BOUNDS
EXECUTE,ARRAY,SUBARRAY

DISCRETE_CLASS

COMPLETE_TYPE,DISCRETE,BY_CONSTRAINING
COMPLETE_TYPE,DISCRETE,BY_DEFINING
COMPLETE_TYPE,DISCRETE,BY_DERIVING

DECLARE_TYPE,DISCRETE,CONSTRAINED,LOCAL
DECLARE_TYPE,DISCRETE,CONSTRAINED,PRIVATE
DECLARE_TYPE,DISCRETE,CONSTRAINED,PUBLIC
DECLARE_TYPE,DISCRETE,DEFINED,LOCAL
DECLARE_TYPE,DISCRETE,DEFINED,PRIVATE
DECLARE_TYPE,DISCRETE,DEFINED,PUBLIC

DECLARE_TYPE,DISCRETE,DERIVED,LOCAL
DECLARE_TYPE,DISCRETE,DERIVED,PRIVATE
DECLARE_TYPE,DISCRETE,DERIVED,PUBLIC
DECLARE_TYPE,DISCRETE,INCOMPLETE,LOCAL
DECLARE_TYPE,DISCRETE,INCOMPLETE,LOCAL,UNSIGNED
DECLARE_TYPE,DISCRETE,INCOMPLETE,PRIVATE
DECLARE_TYPE,DISCRETE,INCOMPLETE,PRIVATE,UNSIGNED
DECLARE_TYPE,DISCRETE,INCOMPLETE,PUBLIC
DECLARE_TYPE,DISCRETE,INCOMPLETE,PUBLIC,UNSIGNED

DECLARE_VARIABLE,DISCRETE,DUPLICATE
DECLARE_VARIABLE,DISCRETE,HIDDEN
DECLARE_VARIABLE,DISCRETE,VISIBLE

EXECUTE,DISCRETE,ABOVE_RANGE
EXECUTE,DISCRETE,ABSOLUTE_VALUE
EXECUTE,DISCRETE,AND
EXECUTE,DISCRETE,BELOW_RANGE
EXECUTE,DISCRETE,BOUNDS
EXECUTE,DISCRETE,BOUNDS_CHECK
EXECUTE,DISCRETE,CHECK_IN_TYPE
EXECUTE,DISCRETE,CHECK_IN_ROOT_TYPE
EXECUTE,DISCRETE,CONVERT
EXECUTE,DISCRETE,DECREMENT
EXECUTE,DISCRETE,DIVIDE
EXECUTE,DISCRETE,EQUAL
EXECUTE,DISCRETE,FIRST
EXECUTE,DISCRETE,GREATER
EXECUTE,DISCRETE,GREATER_EQUAL
EXECUTE,DISCRETE,GREATER_EQUAL_ZERO
EXECUTE,DISCRETE,GREATER_ZERO
EXECUTE,DISCRETE,IN_RANGE
EXECUTE,DISCRETE,IN_TYPE
EXECUTE,DISCRETE,INCREMENT
EXECUTE,DISCRETE,IS_ZERO
EXECUTE,DISCRETE,LAST
EXECUTE,DISCRETE,LESS
EXECUTE,DISCRETE,LESS_EQUAL
EXECUTE,DISCRETE,LESS_EQUAL_ZERO
EXECUTE,DISCRETE,LESS_ZERO
EXECUTE,DISCRETE,MODULO
EXECUTE,DISCRETE,MINUS
EXECUTE,DISCRETE,NOT
EXECUTE,DISCRETE,NOT_EQUAL
EXECUTE,DISCRETE,NOT_IN_RANGE
EXECUTE,DISCRETE,NOT_IN_TYPE
EXECUTE,DISCRETE,NOT_ZERO
EXECUTE,DISCRETE,OR
EXECUTE,DISCRETE,PLUS
EXECUTE,DISCRETE,PREDECESSOR
EXECUTE,DISCRETE,RAISE
EXECUTE,DISCRETE,REMAINDER

EXECUTE/DISCRETE/REVERSE_BOUNDS
EXECUTE/DISCRETE/TIMES
EXECUTE/DISCRETE/SUCCESSOR
EXECUTE/DISCRETE/UNARY_MINUS
EXECUTE/DISCRETE/XOR

ENTRY_CLASS

DECLARE_VARIABLE/ENTRY

EXECUTE/ENTRY/COUNT
EXECUTE/ENTRY/RENDEZVOUS

EXCEPTION_CLASS

EXECUTE/EXCEPTION/ADDRESS
EXECUTE/EXCEPTION/NAME
EXECUTE/EXCEPTION/SCOPE_OF_RAISE

FAMILY_CLASS

DECLARE_VARIABLE/FAMILY

EXECUTE/FAMILY/COUNT
EXECUTE/FAMILY/RENDEZVOUS

FLOAT_CLASS

COMPLETE_TYPE/FLOAT/BY_CONSTRAINING
COMPLETE_TYPE/FLOAT/BY_DEFINING
COMPLETE_TYPE/FLOAT/BY_DERIVING

DECLARE_TYPE/FLOAT/CONSTRAINED/LOCAL
DECLARE_TYPE/FLOAT/CONSTRAINED/PRIVATE
DECLARE_TYPE/FLOAT/CONSTRAINED/PUBLIC
DECLARE_TYPE/FLOAT/DEFINED/LOCAL
DECLARE_TYPE/FLOAT/DEFINED/PRIVATE
DECLARE_TYPE/FLOAT/DEFINED/PUBLIC
DECLARE_TYPE/FLOAT/DERIVED/LOCAL
DECLARE_TYPE/FLOAT/DERIVED/PRIVATE
DECLARE_TYPE/FLOAT/DERIVED/PUBLIC
DECLARE_TYPE/FLOAT/INCOMPLETE/LOCAL
DECLARE_TYPE/FLOAT/INCOMPLETE/PRIVATE
DECLARE_TYPE/FLOAT/INCOMPLETE/PUBLIC

DECLARE_VARIABLE/FLOAT/DUPLICATE
DECLARE_VARIABLE/FLOAT/HIDDEN
DECLARE_VARIABLE/FLOAT/VISIBLE

EXECUTE/FLOAT/ABSOLUTE_VALUE
EXECUTE/FLOAT/CHECK_IN_TYPE
EXECUTE/FLOAT/CONVERT

EXECUTE,FLOAT,DIVIDE
EXECUTE,FLOAT,EQUAL
EXECUTE,FLOAT,FIRST
EXECUTE,FLOAT,GREATER
EXECUTE,FLOAT,GREATER_EQUAL
EXECUTE,FLOAT,IN_TYPE
EXECUTE,FLOAT,LAST
EXECUTE,FLOAT,LESS
EXECUTE,FLOAT,LESS_EQUAL
EXECUTE,FLOAT,MINUS
EXECUTE,FLOAT,NOT_EQUAL
EXECUTE,FLOAT,NOT_IN_TYPE
EXECUTE,FLOAT,PLUS
EXECUTE,FLOAT,TIMES
EXECUTE,FLOAT,UNARY_MINUS

MATRIX_CLASS

EXECUTE,MATRIX,BOUNDS
EXECUTE,MATRIX,CHECK_IN_TYPE
EXECUTE,MATRIX,CONVERT
EXECUTE,MATRIX,CONVERT_ACTUAL
EXECUTE,MATRIX,ELEMENT_TYPE
EXECUTE,MATRIX,EQUAL
EXECUTE,MATRIX,FIELD_READ
EXECUTE,MATRIX,FIELD_REFERENCE
EXECUTE,MATRIX,FIELD_WRITE
EXECUTE,MATRIX,FIRST
EXECUTE,MATRIX,IN_TYPE
EXECUTE,MATRIX,LAST
EXECUTE,MATRIX,LENGTH
EXECUTE,MATRIX,NOT_EQUAL
EXECUTE,MATRIX,NOT_IN_TYPE
EXECUTE,MATRIX,REVERSE_BOUNDS
EXECUTE,MATRIX,SUBARRAY

MODULE_CLASS

EXECUTE,MODULE,ABORT
EXECUTE,MODULE,ACTIVATE
EXECUTE,MODULE,AUGMENT_IMPORTS
EXECUTE,MODULE,CONTINUE
EXECUTE,MODULE,CONVERT
EXECUTE,MODULE,INTERRUPT
EXECUTE,MODULE,IS_CALLABLE
EXECUTE,MODULE,IS_TERMINATED

PACKAGE_CLASS

COMPLETE_TYPE,PACKAGE,BY_DEFINING
COMPLETE_TYPE,PACKAGE,BY_DERIVING

```
  DECLARE_TYPE,PACKAGE,DEFINED,LOCAL
  DECLARE_TYPE,PACKAGE,DEFINED,PRIVATE
  DECLARE_TYPE,PACKAGE,DEFINED,PUBLIC
  DECLARE_TYPE,PACKAGE,DERIVED,LOCAL
  DECLARE_TYPE,PACKAGE,DERIVED,PRIVATE
  DECLARE_TYPE,PACKAGE,DERIVED,PUBLIC
  DECLARE_TYPE,PACKAGE,INCOMPLETE,LOCAL
  DECLARE_TYPE,PACKAGE,INCOMPLETE,PRIVATE
  DECLARE_TYPE,PACKAGE,INCOMPLETE,PUBLIC

  DECLARE_VARIABLE,PACKAGE,HIDDEN
  DECLARE_VARIABLE,PACKAGE,HIDDEN,DISTRIBUTOR
  DECLARE_VARIABLE,PACKAGE,VISIBLE
  DECLARE_VARIABLE,PACKAGE,VISIBLE,DISTRIBUTOR

  EXECUTE,PACKAGE,FIELD_EXECUTE,@FIELD_NUMBER
  EXECUTE,PACKAGE,FIELD_READ,@FIELD_NUMBER
  EXECUTE,PACKAGE,FIELD_REFERENCE,@FIELD_NUMBER
  EXECUTE,PACKAGE,FIELD_WRITE,@FIELD_NUMBER

RECORD_CLASS

  COMPLETE_TYPE,RECORD,BY_COMPONENT_COMPLETION
  COMPLETE_TYPE,RECORD,BY_DEFINING
  COMPLETE_TYPE,RECORD,BY_DERIVING

  DECLARE_TYPE,RECORD,DEFINED,LOCAL
  DECLARE_TYPE,RECORD,DEFINED_INCOMPLETE,LOCAL
  DECLARE_TYPE,RECORD,DEFINED,PRIVATE
  DECLARE_TYPE,RECORD,DEFINED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,RECORD,DEFINED,PUBLIC
  DECLARE_TYPE,RECORD,DEFINED_INCOMPLETE,PUBLIC
  DECLARE_TYPE,RECORD,DERIVED,LOCAL
  DECLARE_TYPE,RECORD,DERIVED,PRIVATE
  DECLARE_TYPE,RECORD,DERIVED,PUBLIC
  DECLARE_TYPE,RECORD,INCOMPLETE,LOCAL
  DECLARE_TYPE,RECORD,DERIVED_INCOMPLETE,LOCAL
  DECLARE_TYPE,RECORD,INCOMPLETE,PRIVATE
  DECLARE_TYPE,RECORD,DERIVED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,RECORD,INCOMPLETE,PUBLIC
  DECLARE_TYPE,RECORD,DERIVED_INCOMPLETE,PUBLIC

  DECLARE_VARIABLE,RECORD,DUPLICATE
  DECLARE_VARIABLE,RECORD,HIDDEN
  DECLARE_VARIABLE,RECORD,VISIBLE

  EXECUTE,RECORD,CONVERT
  EXECUTE,RECORD,EQUAL
  EXECUTE,RECORD,FIELD_READ,@FIELD_NUMBER
  EXECUTE,RECORD,FIELD_REFERENCE,@FIELD_NUMBER
  EXECUTE,RECORD,FIELD_TYPE,@FIELD_NUMBER
  EXECUTE,RECORD,FIELD_WRITE,@FIELD_NUMBER
```

```
  EXECUTE,RECORD,NOT_EQUAL

SEGMENT_CLASS

  DECLARE_TYPE,SEGMENT,DEFINED,LOCAL
  DECLARE_TYPE,SEGMENT,DEFINED,PRIVATE
  DECLARE_TYPE,SEGMENT,DEFINED,PUBLIC

  DECLARE_VARIABLE,SEGMENT,HIDDEN
  DECLARE_VARIABLE,SEGMENT,VISIBLE

  EXECUTE,SEGMENT,ADDRESS
  EXECUTE,SEGMENT,GET_SUBUNIT
  EXECUTE,SEGMENT,GET_SUBUNIT_COUNT
  EXECUTE,SEGMENT,INSTRUCTION_READ
  EXECUTE,SEGMENT,INSTRUCTION_WRITE
  EXECUTE,SEGMENT,MAKE_ADDRESS
  EXECUTE,SEGMENT,SET_SUBUNIT
  EXECUTE,SEGMENT,SET_SUBUNIT_COUNT
  EXECUTE,SEGMENT,WORD_WRITE

SELECT_CLASS

  DECLARE_VARIABLE,SELECT

  EXECUTE,SELECT,FIELD_WRITE,@FIELD_NUMBER
  EXECUTE,SELECT,GUARD_WRITE,@FIELD_NUMBER
  EXECUTE,SELECT,RENDEZVOUS

SUBARRAY_CLASS

  EXECUTE,SUBARRAY,FIELD_READ
  EXECUTE,SUBARRAY,FIELD_REFERENCE
  EXECUTE,SUBARRAY,FIELD_WRITE

SUBMATRIX_CLASS

  EXECUTE,SUBMATRIX,FIELD_READ
  EXECUTE,SUBMATRIX,FIELD_REFERENCE
  EXECUTE,SUBMATRIX,FIELD_WRITE

SUBVECTOR_CLASS

  EXECUTE,SUBVECTOR,FIELD_READ
  EXECUTE,SUBVECTOR,FIELD_REFERENCE
  EXECUTE,SUBVECTOR,FIELD_WRITE

TASK_CLASS

  COMPLETE_TYPE,TASK,BY_DEFINING
  COMPLETE_TYPE,TASK,BY_DERIVING
```

```
  DECLARE_TYPE,TASK,DEFINED,LOCAL
  DECLARE_TYPE,TASK,DEFINED,LOCAL,WITH_ENTRYS
  DECLARE_TYPE,TASK,DEFINED,PRIVATE
  DECLARE_TYPE,TASK,DEFINED,PRIVATE,WITH_ENTRYS
  DECLARE_TYPE,TASK,DEFINED,PUBLIC
  DECLARE_TYPE,TASK,DEFINED,PUBLIC,WITH_ENTRYS
  DECLARE_TYPE,TASK,DERIVED,LOCAL
  DECLARE_TYPE,TASK,DERIVED,PRIVATE
  DECLARE_TYPE,TASK,DERIVED,PUBLIC
  DECLARE_TYPE,TASK,INCOMPLETE,LOCAL
  DECLARE_TYPE,TASK,INCOMPLETE,LOCAL,WITH_ENTRYS
  DECLARE_TYPE,TASK,INCOMPLETE,PRIVATE
  DECLARE_TYPE,TASK,INCOMPLETE,PRIVATE,WITH_ENTRYS
  DECLARE_TYPE,TASK,INCOMPLETE,PUBLIC
  DECLARE_TYPE,TASK,INCOMPLETE,PUBLIC,WITH_ENTRYS

  DECLARE_VARIABLE,TASK,DATA_TASK
  DECLARE_VARIABLE,TASK,DATA_TASK,DISTRIBUTOR
  DECLARE_VARIABLE,TASK,HEAP_TASK
  DECLARE_VARIABLE,TASK,HEAP_TASK,DISTRIBUTOR
  DECLARE_VARIABLE,TASK,HIDDEN
  DECLARE_VARIABLE,TASK,HIDDEN,DISTRIBUTOR
  DECLARE_VARIABLE,TASK,VISIBLE
  DECLARE_VARIABLE,TASK,VISIBLE,DISTRIBUTOR

  EXECUTE,TASK,COND_CALL,@FIELD_NUMBER
  EXECUTE,TASK,ENTRY_CALL,@FIELD_NUMBER
  EXECUTE,TASK,FAMILY_CALL,@FIELD_NUMBER
  EXECUTE,TASK,FAMILY_COND,@FIELD_NUMBER
  EXECUTE,TASK,FAMILY_TIMED,@FIELD_NUMBER
  EXECUTE,TASK,TIMED_CALL,@FIELD_NUMBER

VARIANT_RECORD_CLASS

  COMPLETE_TYPE,VARIANT_RECORD,BY_COMPONENT_COMPLETION
  COMPLETE_TYPE,VARIANT_RECORD,BY_CONSTRAINING
  COMPLETE_TYPE,VARIANT_RECORD,BY_DEFINING
  COMPLETE_TYPE,VARIANT_RECORD,BY_DERIVING

  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED_INCOMPLETE,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,CONSTRAINED_INCOMPLETE,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,DEFINED,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,DEFINED_INCOMPLETE,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,DEFINED,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,DEFINED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,DEFINED,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,DEFINED_INCOMPLETE,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,DERIVED,LOCAL
```

```
  DECLARE_TYPE,VARIANT_RECORD,DERIVED_INCOMPLETE,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,DERIVED,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,DERIVED_INCOMPLETE,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,DERIVED,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,DERIVED_INCOMPLETE,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,LOCAL
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,LOCAL,CONSTRAINED
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,LOCAL,DERIVED_PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PRIVATE,CONSTRAINED
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PRIVATE,DERIVED_PRIVATE
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PUBLIC
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PUBLIC,CONSTRAINED
  DECLARE_TYPE,VARIANT_RECORD,INCOMPLETE,PUBLIC,DERIVED_PRIVATE

  DECLARE_VARIABLE,VARIANT_RECORD,DUPLICATE
  DECLARE_VARIABLE,VARIANT_RECORD,HIDDEN
  DECLARE_VARIABLE,VARIANT_RECORD,HIDDEN,WITH_CONSTRAINT
  DECLARE_VARIABLE,VARIANT_RECORD,VISIBLE
  DECLARE_VARIABLE,VARIANT_RECORD,VISIBLE,WITH_CONSTRAINT

  EXECUTE,VARIANT_RECORD,CHECK_IN_TYPE
  EXECUTE,VARIANT_RECORD,CONVERT
  EXECUTE,VARIANT_RECORD,EQUAL
  EXECUTE,VARIANT_RECORD,FIELD_READ,FIXED,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_READ,FIXED,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_READ,VARIANT,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_READ,VARIANT,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_REFERENCE,FIXED,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_REFERENCE,FIXED,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_REFERENCE,VARIANT,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_REFERENCE,VARIANT,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_WRITE,VARIANT,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_WRITE,FIXED,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_WRITE,FIXED,INDIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_WRITE,VARIANT,DIRECT,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,FIELD_TYPE,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,IS_CONSTRAINED
  EXECUTE,VARIANT_RECORD,IN_TYPE
  EXECUTE,VARIANT_RECORD,MAKE_CONSTRAINED
  EXECUTE,VARIANT_RECORD,NOT_EQUAL
  EXECUTE,VARIANT_RECORD,NOT_IN_TYPE
  EXECUTE,VARIANT_RECORD,SET_BOUNDS,@FIELD_NUMBER
  EXECUTE,VARIANT_RECORD,SET_CONSTRAINT
  EXECUTE,VARIANT_RECORD,SET_VARIANT,@FIELD_NUMBER

VECTOR_CLASS

  EXECUTE,VECTOR,APPEND
  EXECUTE,VECTOR,AND
  EXECUTE,VECTOR,BOUNDS
  EXECUTE,VECTOR,CATENATE
```

EXECUTE,VECTOR,CHECK_IN_TYPE
EXECUTE,VECTOR,CONVERT
EXECUTE,VECTOR,CONVERT_ACTUAL
EXECUTE,VECTOR,ELEMENT_TYPE
EXECUTE,VECTOR,EQUAL
EXECUTE,VECTOR,FIELD_READ
EXECUTE,VECTOR,FIELD_REFERENCE
EXECUTE,VECTOR,FIELD_WRITE
EXECUTE,VECTOR,FIRST
EXECUTE,VECTOR,IN_TYPE
EXECUTE,VECTOR,LAST
EXECUTE,VECTOR,LENGTH
EXECUTE,VECTOR,NOT
EXECUTE,VECTOR,NOT_EQUAL
EXECUTE,VECTOR,NOT_IN_TYPE
EXECUTE,VECTOR,OR
EXECUTE,VECTOR,PREPEND
EXECUTE,VECTOR,REVERSE_BOUNDS
EXECUTE,VECTOR,SLICE_READ
EXECUTE,VECTOR,SLICE_WRITE
EXECUTE,VECTOR,XOR

B.2. UNCLASSED INSTRUCTIONS

ACTION

ACTION,ACCEPT_ACTIVATION
ACTION,ACTIVATE_SUBPROGRAM
ACTION,ACTIVATE_TASKS
ACTION,ALTER_BREAK_MASK
ACTION,BREAK_OPTIONAL
ACTION,BREAK_UNCONDITIONAL
ACTION,CALL_IMPORT
ACTION,CALL_REFERENCE
ACTION,DELETE_ITEM
ACTION,DELETE_SUBPROGRAM
ACTION,ESTABLISH_FRAME
ACTION,EXIT_BREAK
ACTION,IDLE
ACTION,ILLEGAL
ACTION,INITIATE_DELAY
ACTION,INTRODUCE_IMPORT
ACTION,MAKE_NULL_UTILITY
ACTION,MAKE_SELF
ACTION,MARK_AUXILIARY
ACTION,MARK_DATA
ACTION,MARK_TYPE
ACTION,NAME_MODULE
ACTION,NAME_PARTNER

ACTION,OVERWRITE_IMPORT
ACTION,POP_AUXILIARY
ACTION,POP_CONTROL
ACTION,POP_DATA
ACTION,POP_TYPE
ACTION,PROPAGATE_ABORT
ACTION,QUERY_BREAK_ADDRESS
ACTION,QUERY_BREAK_CAUSE
ACTION,QUERY_BREAK_MASK
ACTION,QUERY_FRAME
ACTION,QUERY_RESOURCE_LIMITS
ACTION,QUERY_RESOURCE_STATE
ACTION,READ_IMPORT
ACTION,RECOVER_RESOURCES
ACTION,REFERENCE_IMPORT
ACTION,REMOVE_IMPORT
ACTION,RETURN_RESOURCES
ACTION,SET_BREAK_MASK
ACTION,SET_INTERFACE_SCOPE
ACTION,SET_INTERFACE_SUBPROG
ACTION,SET_NULL_ACCESS
ACTION,SET_RESOURCE_LIMITS
ACTION,SET_VISIBILITY
ACTION,SIGNAL_ACTIVATED
ACTION,SIGNAL_COMPLETION
ACTION,SWAP_CONTROL
ACTION,WRITE_IMPORT

CALL

CALL,@LEX_LEVEL_DELTA

DECLARE_SUBPROGRAM

DECLARE_SUBPROGRAM,FOR_ACCEPT
DECLARE_SUBPROGRAM,FOR_CALL,HIDDEN,ACTIVE
DECLARE_SUBPROGRAM,FOR_CALL,HIDDEN,INACTIVE
DECLARE_SUBPROGRAM,FOR_CALL,VISIBLE,ACTIVE
DECLARE_SUBPROGRAM,FOR_CALL,VISIBLE,INACTIVE
DECLARE_SUBPROGRAM,FOR_INTERFACE
DECLARE_SUBPROGRAM,FOR_OUTER_CALL,HIDDEN
DECLARE_SUBPROGRAM,FOR_OUTER_CALL,VISIBLE
DECLARE_SUBPROGRAM,FOR_UTILITY

EXIT_ACCEPT

EXIT_ACCEPT,@NEW_TOP_OFFSET

EXIT_FUNCTION

EXIT_FUNCTION,@NEW_TOP_OFFSET

EXIT_PROCEDURE

  EXIT_PROCEDURE,@NEW_TOP_OFFSET

EXIT_UTILITY

  EXIT_UTILITY,@NEW_TOP_OFFSET

INDIRECT_LITERAL

  INDIRECT_LITERAL,ARRAY,@PC_DISPLACEMENT
  INDIRECT_LITERAL,DISCRETE,@PC_DISPLACEMENT
  INDIRECT_LITERAL,FLOAT,@PC_DISPLACEMENT

JUMP

  JUMP,@PC_DISPLACEMENT

JUMP_CASE

  JUMP_CASE,@CASE_MAXIMUM

JUMP_NONZERO

  JUMP_NONZERO,@PC_DISPLACEMENT

JUMP_ZERO

  JUMP_ZERO,@PC_DISPLACEMENT

LOAD

  LOAD,@LEX_LEVEL_DELTA

LOAD_TOP

  LOAD_TOP,@STACK_TOP_OFFSET

POP_BLOCK

  POP_BLOCK,@TARGET_LEX

POP_BLOCK_RESULT

  POP_BLOCK_RESULT,@TARGT_LEX

SHORT_LITERAL

  SHORT_LITERAL,@LITERAL_VALUE

STORE

STORE/@LEX_LEVEL_DELTA

REFERENCE

REFERENCE/@LEX_LEVEL_DELTA

Appendix C
GLOSSARY

CLASS — Refers to membership in a set of data which are recognized and manipulated by the Rational Machines architecture. The class of an object determines its representation in memory, and hence the primitive means for interpreting the object. The definition of a class specifies a set of operations applicable to objects of that class; no other operations on an object of a given class are legal, and furthermore, objects of incompatible classes may not implicitly operate with one another.

EXCEPTION — An event that causes suspension of normal program execution.

KIND — Used to distinguish distinct varieties of memory (i.e. program, control, type, data, import, and queue), and to distinguish a particular representation for a word in memory.

MODULE — Used to indicate the logical group of segments which are associated with the value of a package or task, and which share the same name. A given module will always have a control segment and program segment allocated, but may or may not have type, data, import, or queue segments allocated.

OBJECT — An entity together with some means for identification and interpretation of the entity. The means of interpretation may provide selection of attributes of the entity, such as value, text representation, etc., may provide transformation into other objects, or may provide the mechanism for the construction and interpretation of other objects. An object is an instance of a particular class and hence is associated with a set of applicable operations. An object in the sense of the Rational Machines architecture has a broader definition than in Ada; for example, the architecture recognizes program units such as packages as tasks as objects.

REFERENCE — Refers to an address either in the form of lex level/delta in an instruction (an object reference) or in the form of processor/memory kind/offset/index (a machine logical address).

SEGMENT — A portion of the full logical address space of the architecture, referenced using a specific processor/memory kind/. The term STACK is synonymous with SEGMENT in the case of control, type, and data memory kinds.

SPACE — Used to either describe the total logically addressed memory of the architecture or that portion which may be referenced using a logical name.

Index

Table of Contents