```
  AAAAAA      RRRRRRRR        CCCCCCCC    HH      HH
  AAAAAA      RRRRRRRR        CCCCCCCC    HH      HH
AA      AA    RR      RR    CC            HH      HH
AA      AA    RR      RR    CC            HH      HH
AA      AA    RR      RR    CC            HH      HH
AA      AA    RR      RR    CC            HH      HH
AA      AA    RRRRRRRR      CC            HHHHHHHHHH
AA      AA    RRRRRRRR      CC            HHHHHHHHHH
AAAAAAAAAA    RR  RR        CC            HH      HH
AAAAAAAAAA    RR  RR        CC            HH      HH
AA      AA    RR    RR      CC            HH      HH
AA      AA    RR    RR      CC            HH      HH
AA      AA    RR      RR      CCCCCCCC    HH      HH
AA      AA    RR      RR      CCCCCCCC    HH      HH



LL            PPPPPPPP      TTTTTTTTTT                  44    44
LL            PPPPPPPP      TTTTTTTTTT                  44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PPPPPPPP          TT                      4444444444
LL            PPPPPPPP          TT                      4444444444
LL            PP                TT                            44
LL            PP                TT                            44
LL            PP                TT            ....            44
LL            PP                TT            ....            44
LLLLLLLLLL    PP                TT            ....            44
LLLLLLLLLL    PP                TT            ....            44
```

START Job ARCH Req #116 for EGB   Date 23-Jul-82 21:45:03 Monitor: Rational M
File RM:<SIM.DOC.ARCHITECTURE>ARCH.LPT.41, created: 4-Feb-82 17:52:08
printed: 23-Jul-82 21:45:03
Job parameters: Request created:23-Jul-82 21:30:32   Page limit:171   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:A

Rational Machines Architecture

DRAFT 14

February 4, 1982

Chapter 1
Introduction

This section introduces the Rational Machines Architecture and provides some of the motivation for its creation. The Ada language is also introduced.

1.1. Goals of the architecture

Over the last three decades, computer architecture and programming languages have evolved a great deal. Computer architecture has moved from small memory, irregular instruction set, special purpose machines to larger memory, more regular instruction set, general purpose machines. Improvements in technology have contributed greatly to this, as has the recognition of the role and importance of software and higher-level languages.

An important design parameter of computer architectures now is the ease with which high-level languages can be implemented under them. Thus, the evolutionary paths of languages have now become intertwined with those of machine architecture.

Programming languages have also been evolving extensively over the past several decades. Initially, languages were only slightly embellished versions of the order code of the machine. 'High-level' languages and the notion of machine independence were a major step toward programming languages that were closer to the problem than to the machine implementation. Subsequent evolutionary steps recognized the importance of problem-specific constructs in languages to ease the solution of certain classes of problems. This was tempered with the notion of 'general purpose' languages that would be applicable for many problems. The two ideas were compromised somewhat with the recognition that abstraction mechanisms whereby the programmer could define a data structure and set of operations that provided an 'integrated' facility not originally present in the language could provide many of the good features of both the special purpose and general purpose world.

In addition, a number of specific language characteristics and features were recognized to provide support for the construction of large, reliable programs.

The Ada language is a product of the current language evolutionary trend. It provides abstraction mechanisms and a number of other features currently recognized as supportive of the programming process.

The goal, then, of the Rational Machines Architecture is to take the next step in the combined evolution of computer architecture and programming languages and provide an architecture that strongly supports the implementation of modern programming languages such as Ada. As Ada is a step beyond current production languages, having an architecture that supports Ada well will make a system architecture that provides superior price/performance characteristics at its level of functionality.

A good programming environment is also a critical requirement for the rapid production of quality software. The Rational Machines Architecture must also support such a programming environment. Thus, requirements of the environment also influence the architecture.

1.2. General description of Ada

The Ada language has all sorts of stuff in it. See section 1.2 of the Ada manual.

1.3. Architectural implications for support of Ada

They're not small.

Chapter 2
Overview of the Architecture

This section gives a general overview of the architecture and distinguishes between architecture and implementation.

2.1. Architecture vs implementation

We must distinguish between the notions of 'architecture', 'architecture implementation', and pure 'implementation'. In general, 'architecture' refers to the abstract properties of the machine in terms of what is visible from the point of view of the instructions. This would include memory types (if distinguishable from the instructions), stacks, procedure and parameter mechanisms (though not specific structures of stack markers unless visible to the instructions), and other features of this nature. Binary machine instructions are not necessarily transportable between two machines conforming to the same architecture. High level programs, however, are. Different code generators may be required, and there may be different restrictions on maximums and minimums of various parameters, but machines conforming to the same architecture would be substantially similar in function.

'Architecture implemetation' includes issues such as address sizes, instruction encodings, and descriptor formats. Programs in binary are transportable between machines having the same architecture implementations. Such machines have the same functionality and encodings of program data.

'Implementation' refers to the internal structure of the machine. This structure is not visible to a running program in any way except for performance speed. Thus, implementation issues refer to uses of caches, pipelines, internal registers, and things of that nature.

In general, the term 'Rational Machines Architecture' refers to the architecture level, '<machine-name> architecture' (where <machine-name> is the name of a machine in the RMI product line) refers to an architecture implementation, and '<machine-name> implementation' refers to the implementation of the given machine.

2.2. General organization

Many basic concepts of Ada extend directly into the architecture.

Type checking
Every object in the system has a type. Whenever an instruction performs an action on an object, the type of the object is checked for compatibility with the operation being performed. Unacceptable types cause machine-raised exceptions. The exact function performed by the operation will depend on the type(s) of the operand(s). For example, ADD will add numbers of various types, performing the appropriate operation based on the actual type.

Early detection of inconsistencies in the program is facilitated by type checking and contributes to a decrease in debugging time and an increase in reliability. The type information provides a knowledge of data structure (eg, array, record) at the machine level and leads to more efficient access to such structures.

Abstract types
The machine enforces abstractions as defined in Ada. This is the primary protection mechanism. When an abstract type is declared, concrete operations on the abstract data objects (private types in Ada) can only be performed in the part of the program that defined the type. The machine knows if an abstract type is being operated upon and prevents unauthorized accesses.

Tasking
There are a number of features in the architecture to facilitate the Ada tasking functions. The architecture supports the notion of tasks, rendezvous, delays, and task initiation and termination. For example, the return-from-procedure-call instruction automatically waits until any local tasks have terminated before returning.

Collections
Collections are the memories that are accessed by access types. The archecture provides a special memory segment type for collections.

Access to data types
The basic structured types in Ada (arrays, records, variant records) are supported in the architecture with special instructions that provide efficient access and appropriate constraint checking.

Exception Handling
The Ada exception handling functions are also directly supported in the architecture.

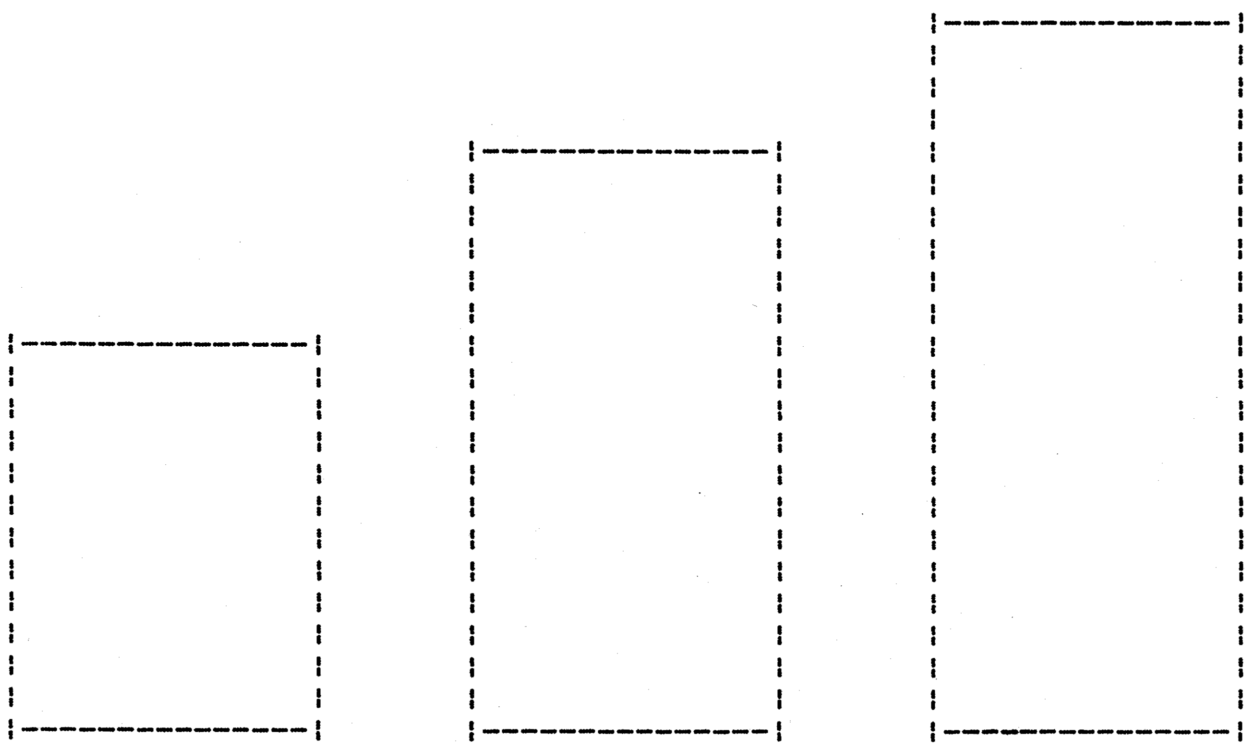
Subprograms/Packages
There are facilities in the architecture that support package instantiation, elaboration, and termination.

2.3. Run-Time Model

The run-time environment revolves around the Ada module (a package or task). Although modules could be implemented in different ways by different translators, the architecture reflects an intent that modules be implemented as follows.

There is one program segment for each module. This segment contains all the code for the module, and the names of the program segments for enclosed modules.

There is also one control stack and type stack for each module. As types are defined in the module, entries are made on the control and type stack. As objects are declared in the module, entries are made on the control stack with pointers to the type (in a type stack) and to the value (on the data stack) if the object is a structure. Procedure activations also are placed on a module's control stack but only if it is an 'active' object (a task or a package during package elaboration).



Type Stack Control Stack Data Stack

Chapter 3
Data Units

3.1. Introduction

This section describes the addressing and simple data types in the architecture. It also describes the R1000 architecture-defined formats of several of these types. Through most of this section, the R1000 architecture is described rather than the architecture in abstract form. Occasionally, the distinctions are noted.

3.2. Storage of Information in the R1000 Architecture

This section begins the description of the R1000 architecture. The R1000 memory is a typed segmented memory system. Memory is divided into a number of variable sized segments each of which has a class. The classes are: control stack (CS), data stack (DS), type stack (TS), program segment (PS), collection (COLL), and import segment (IS).

Each of the segments has a unique name. This name is composed of a segment id and the segment class. Thus, a type stack, for example, can be associated with a control stack by having the same segment id.

A control stack consists of a number of frames each of which contains a marker and data. A frame is pushed on the CS for each block that is entered (which is usually by a subprogram call). The marker contains return address, context, and block cleanup information.

The data part of the frame contains data objects. Each object is either a type variable, or a (type, value) pair. (A type variable is also a (type, value) pair, but the value part is ignored.) There is a type variable for each type introduced in an Ada program.

The type part of the (type, value) pair contains protection and access information and a pointer to a type descriptor on a type stack. The value part either contains the value of the object if it is a scalar small enough to fit (64 bits in the R1000 architecture), or else a pointer to the object on the data stack.

A type stack contains type descriptors for each of the data objects referenced by a control stack. Each control stack is associated with a unique type stack, however, a control stack can refer to type information on any type stack. Range and constraint information for scalars, arrays, and records are stored within a type stack. There are no instructions to directly address memory within a type stack. The type stack also contains resource allocation information for local tasks and collections.

Statically named structures (arrays, records, etc.) are allocated on the data stack. (R1000 Implementation note: The data stack is bit-packed for maximum storage density. Each element is allocated the minimum number bits based on its range.) Data stack elements are pointed to by CS entries.

Dynamically allocated data elements are allocated in collection segments. There is one collection segment for each access type. Pointers in Ada are used to access data that is stored in collection segments. The pointers themselves, however, may be stored in control stacks, data stacks, or collection segments.

Program segments contain executable code. Some exception handling information is imbedded within the program segement.

Import segments contain references to objects not declared within a module but accessed from within it.

(The abstract architecture provides for memory segments of type CS, DS, PS, COLL, and IS. Packing is strictly an implementation issue. The architecture does not specify the location of the type information.)

(Implementation note: Data are containerized in the CS, TS, and PS, meaning that objects are of fixed size and aligned. Information in collections and data stacks is not containerized, allowing optimal storage packing. The architecture is such that objects are first brought to the control stack before being operated on. This allows an implementation that optimizes operation on fixed sized objects; variable sized packed structures are extracted and replaced in their memory segments and converted to fixed sized objects prior to actual use. This provides advantages of both fast operation on fixed sized objects and excellect packing density with objects or arbitrary size.)

3.3. Addressing Structure

The architecture provides a variety of techniques for accessing objects. Since each object has a type, there are addressing modes specifically designed for accessing objects of particular types (eg, arrays, records). The following table summarizes the addressing modes:

Name	Parameters	Use
CS Object	Lex Level, Delta	Accessing any statically named object.
PS Relative	Offset	Branches within a program segment.
PS Absolute	Segment id, Offset	Program segment branches.
Array	subscripts	Array element access.
Record Field	field number	Record field access.
Package Field	field number	Inter-package access to data or subprogram.

3.4. Types

The following types are supported in the architecture:

DISCRETE_VAR: A signed or unsigned value. The minimum and maximum value are part of the type descriptor.

FLOAT_VAR: A floating point value. The minimum and maximum value are part of the type descriptor. Accuracy, etc???

ACCESS_VAR: A pointer to an object. The type is the actual type of the object pointed to (which is fixed) and the collection pointed to.

MODULE_VAR: A package or task type. The type consists of the name of the program segment that contains the code for the module, parameters (generic) of the module, and objects imported into the module from other modules.

RECORD_VAR: A union of several other objects which may be of different types. Each field of the record descriptor specifies a type, size, and location in the record for the record field.

VARIANT_RECORD_VAR: A discriminated union of objects which may be of different types. There is a fixed part of the record common to all variants which contains a discriminant field indicating which of the possible variants is contained in each instance. A variant record may be constrained by being bound to contain only one of several possible variants.

ARRAY_VAR: A union of several objects each of which is of the same type. The different objects are indexed by a DISCRETE type. The type descriptor includes the number of dimensions and the minimum, maximum, and type of each of the dimensions.

ENUMERATION_REF: A pointer to an enumeration object.

INTEGER_REF: A pointer to an integer object on the data stack of in a collection segment.

FLOAT_REF: A pointer to a floating point object.

ACCESS_REF: A pointer to a pointer object.

MODULE_REF: A pointer to a module object.

REFERENCE_VAR: An intermodule import, indicating the actual object and its context (pointer to control stack). Used for the importing of subprograms into one module from another, and other special things.

SUBPROGRAM_VAR: A callable object. Either a procedure, function, utility subprogram (used for structure initialization), or ?.

SELECT_VAR: An executable object that handles processing of the Ada SELECT construct.

ENTRY_VAR: An object corresponding to an Ada entry into a task. Used to keep track of entry calls.

EXCEPTION_VAR: Left on the control stack when an exception is raised. Objects of this type represent an exception, including its identity and where it was raised.

SEGMENT_VAR: Used to create code segments. Objects of this type are created and used by the compiler to transform a data segment of some form into executable code.

The following table summarizes R1000 architecture memory reference structures. Numbers by themselves indicate the number of bits in the field. Angle brakets enclose reference structures with substructure.

Program segment reference

```
REFERENCE:                <WORD: 9; INSTRUCTION: 3>          (12 bits)
ADDRESS: &lt;SEGMENT: 20; WORD: 9; INSTRUCTION: 3>              (32 bits)
JUMP_OFFSET: 11 (signed)
CASE_MAXIMUM: 9
```

Segment reference (of any class)

```
ID: 24
  MODULE_ID: 24 (w/ bit 23 = 0)
  COLLECTION_ID: 24 (w/ bit 23 = 1)
NAME: <PROCESSOR: 8; NUMBER: ID>                             (32 bits)
```

Control stack reference

```
OBJECT_REFERENCE: <LEX_LEVEL: 4; OFFSET: 9 (signed)>         (13 bits)
CONTROL_REFERENCE: <STACK: NAME; OFFSET: 20>                 (52 bits)
```

Type stack reference

```
TYPE_DISPLACEMENT: 20
TYPE_REFERENCE: <STACK: NAME; OFFSET: TYPE_DISPLACEMENT>     (52 bits)
```

Data stack reference

```
DATA_REFERENCE: <STACK: NAME; OFFSET: 32>                    (64 bits)

CHILD_POINTER: <OFFSET: TYPE_DISPLACEMENT; INDEX: 2>         (22 bits)
IMPORT_NAME: <PROCESSOR: 8; NUMBER: 24>                      (32 bits)
```

Chapter 4
Major functions

4.1. Introduction

This chapter describes the major architectural features of the Rational Machines Architecture. Each feature relates to the implementation of some programming language feature. The major run-time structures are outlined here; the specific instructions are described in Instruction Set Summary.

In most sections, a small example program is given along with the R1000 instructions that implement it. Some of the early examples may contain instructions not yet discussed or may be incomplete in other ways. The general idea of the instructions can be deduced, however. Just ignore the instructions not yet discussed or return to the example in a second reading.

Also, the full details of the instructions are not discussed. Refer to the Rational Machines Instruction Set document for additional details.

4.2. Run-time Environment

4.2.1. Memory Segments for a Module

As described in earlier chapters, each module has associated with it at least four segments: a program segment containing code, a control stack containing activation information, parameters, and most scalar variables, a data stack containing statically named strcututes declared in the module, and a type stack containing descriptors for types defined in the module.

For each object there is a control stack word that contains either the object's value or a pointer to it, and type information about the object which includes a pointer the the type descriptor for the object on the type stack of the module that defined the object's type.

Objects that are indirectly accessed (ie, not statically named) are allocated in collection segments. All objects in a collection are of the same type. Before an such an object can be accessed, an access variable that points the the object's value must be placed on the control stack. The access variable is then dereferenced, yielding the indirect variable's value.

4.2.2. Declaration of Types and Objects

Both types and variables must be created before they can be used. The DECL instruction is used to create and modify types. The VAR instruction is used to create variables. Variables and types can also be created by copying existing variables and types (typically with the VAL instruction).

Types are created by specifying their specific parameters to the DECL instruction. Once created, variables of that type can be created. The complete information about a variable consists of the type itself and access information. The access information is kept in the control stack entry for the variable and can be different for variables of the same type. The access information indicates whether the variable is private, is limited, is visibile, and is a constant.

4.3. General Object Addressing

4.3.1. Lexical Levels

The normal state of affairs of a running module involves a number of frames on its control stack. Each frame represents an activation of a subprogram or block and contains the objects declared by that activation. Each such activation is associated with a fixed lexical level that is based on the block's static position within the module text.

Lexical levels are numbered based on the position of an object within its declaring module. Level 1 is the outer-most level with level numbers increasing from there.

Objects are accessed by giving a lexical level and offset. The lexical level specifies a specific stack frame and the offset within that frame specifies an object. Positive offsets refer to objects declared within the specified activation; negative offsets refer to parameters passed to the activation.

Lexical levels are numbered from zero. Level zero specifies the import segment, and only positive offsets are allowed at this level. Level 1 is the first activation and contains the static objects declared at the module level. Only tasks will have frames at levels greater than one (except during package elaboration or reelaboration).

4.3.2. Access of Operands

The architecture defines a stack instruction set. Most instructions take their operands from the top of the control stack for the active task. There are instructions for loading values onto the stack and storing values from the top of the stack in other locations in memory.

Constant operands may be pushed on the stack from an immediate field in the instruction (SHORT_LIT) or from locations in the program segment (LIT, LONG_LIT).

Three instructions that are used to access operands are VAL, STO, and REF. Each takes a lexical level number that specifies the frame in which the referenced object exists, and an offset to the object within that frame. VAL is used to load a value to the top of the control stack; STO pops the control stack and stores the value; REF forms a pointer to an object, pushing it on the control stack.

```
package body lex is                  -- A sample package.
    A, B, C: integer;                -- A, B, C at level 1
    procedure P is                   -- P at level 1
        D, E:    integer;            -- D, E at level 1
    begin
        A := 1;        B := A;
        D := B;        E := 10;
        C := 76543567865;
    end P;
end lex;
```

```
Location      Instruction                        Comment
0:            VAL      1,2                 -- push "integer" (magic)
1:            VAR      DISCRETE,HIDDEN     -- create A
              VAL      1,3
              VAR      DISCRETE,HIDDEN     -- create B
              VAL      1,3
              VAR      DISCRETE,HIDDEN     -- create C
        SHORT_LIT      3                   -- push address of P
          SUBPROG      FOR_CALL,HIDDEN     -- create P
           SYS_OP      SIGNAL_ACTIVATED    -- vis part fini (see modules)
2:         SYS_OP      SELECT_TERMINATION

-- Subprogram P
3:            VAL      1,2                 -- push "integer"
              VAR      DISCRETE,HIDDEN     -- create D
              VAL      2,2
              VAR      DISCRETE,HIDDEN     -- create E

4:      SHORT_LIT      1
              STO      1,3                 -- A := 1 (level 1 offset 3)

              VAL      1,3                 -- push A (level 1 offset 3)
              STO      1,4                 -- B := A (level 1 offset 4)

              VAL      1,4
              STO      2,2                 -- D := B (D: level 2 offset 2)

        SHORT_LIT      10
              STO      2,3                 -- E := 10

5:            LIT      DISCRETE,5,3
              STO      1,5                 -- C := 7824091129
        EXIT_PROC      0                   -- return from P

          LIT_VAL      DISCRETE,7824091129
```

4.3.3. Summary of Instructions in this Section

DECL — Declare or modify a type

VAR	Declare a variable
VAL	Push a value on the stack
STO	Pop a stack value and store
REF	Construct a reference and push on the stack
SHORT_LIT	Push a short literal constant
LIT	Push a longer constant loaded from the program segment

4.4. Types and Protection

4.4.1. Type Information

Type information is divided between the control stack and the type stack. Simple information needed to access the value is kept in the control stack so that it can be quickly accessed with the data object itself. This information includes:

- A pointer to the additional information in the type stack.

- The visibility of the object (which indicates if it is in the visible part of a module).

- Privacy information about the object indicating whether the object is public (representation available in packages other than where the object is defined), private (accessible only through abstract operations), limited-private (private and not copyable), or local (available only in the package where the object is defined).

- An indication of whether the object is based on another that was private. (derives privacy)

- An indication of whether the object is based on another that was limited. (derives limitation)

- An indication of whether the object is a constant.

- An indication of whether the object is constrained if the object is an array, variant record, or access variable.

4.4.2. Operand Type Checking

The access information forms the basis of the protection mechanism defined by the architecture. The architecture guarantees that objects will be accessed only in accordance with the way in which they are declared.

When an operand is accessed, some of the following checks are performed. Failure of any of them results in an exception being raised and the operation not being performed.

Type check. The type of the operand must be acceptable for the operation to be performed. This precludes things such as invoking data objects or indexing objects other than arrays.

Operand consistency. If the operation requires more than one operand, the operands supplied must be of consistent types. (OPERAND_CLASS_ERROR)

Operand Privacy. If the operand is private or limited private then the instruction must be executed in the "scope of privacy" (ie, in the body of the module where the type is defined). The instruction is executing in the scope of privacy if the segment id of the outer frame pointer (pointing to the global level) is the same as that of the type descriptor pointer, or if the segment id of a statically enclosing module is the same that of the type descriptor pointer. This means that the subprogram executing is declared in the same module as the type, or in a module enclosed by the module that declared the type. (CAPABILITY_ERROR)

Operand Visibility. When in inter-module reference is made, the referenced operand must be 'visible' from outside its defining module: it must have been declared in the visible part of the module (and hence have its 'visible' indication set). In addition, if the operand is a subprogram, its package must have been elaborated before it can be invoked. (VISIBILITY_ERROR, ELABORATION_ERROR)

Operand Limitation. Some operations (assignment, equality test) are only allowed on non-limited types. Any such operation on a limited type can only be done in the "scope of limitation" where the representation of the object is known (the module body where the type is defined). The instruction is executing in the scope of limitation if the outer frame pointer of the current activation (lex level 1) is the same as the scope-of-limitation field of the type information of the object.

Constant check. Certain operations cannot be performed on constants.

4.4.3. Data abstraction (privacy, limitation)

In Ada, objects can be declared public, private, limited-private, or local. Public objects can be accessed and concrete operations applied by any module that imports the object. Objects of a private type can be declared and accessed in any module, but only functions supplied by the defining module can apply concrete operations to the object. Limited private types are like private but assignments and equality tests cannot be performed. Local objects are known only within the defining module and cannot be accessed from outside.

The architecture supports these concepts directly. The tests made for operand access are described above.

The DECL instruction is used to create types and takes the level of privacy as an operand. In this way, the protection information is known for each object (as each object has a type). Objects are created with the VAR instruction. The visibility of the object (whether or not it is in the visible part of a module) is an operand of this instruction.

4.4.4. Constants

Part of the type information for each object indicates whether the object is a constant. This applies only to the object, not all objects of that type.

Constants are created by first creating a variable object, assigning it a value, and executing the MAKE_CONSTANT_OP operator. This changes the object to a constant. There is no subsequent way to alter the value of that object.

4.4.5. Creation and Alteration of Types

Type objects are created and altered by the DECL instruction. The instruction specifies the specific type which may be any of the types listed in the previous section. (?) New types can be created, derived types created based on existing types, or existing types constrained. The operands depend on the type being created.

Sometimes the type must be introduced and its actual content specified later. The COMPLETE_OP is used to complete type definitions.

4.4.6. Summary of Instruction in this Section

DECL	Create or modify a type
VAR	Create a variable
SUBPROG	Create a subprogram variable
OP MAKE_CONSTANT_OP	Make an object into a constant
OP COMPLETE_OP	Complete a type

4.5. Modules

4.5.1. Package/task correspondence

A module in Ada is either a package or a task.

Module types are prototypes for module instances. A module type corresponds to a task type, a generic package type, or a normal package (in which case there is only one module instance). Thus, a module may have parameters and is characterized by the objects that are declared in its visible part and by its body code.

A module instance is a 'runnable' version of a module type with any parameters filled in. A module instance is generated by a declaration of a

```
package test is
    type T1 is new integer;       -- T1 is a new (derived) type
    C1:  constant T1 := 23;       -- C1 is a constant
    type P is private;            -- P is a private type
    type P2 is limited private;   -- P2 is a limited-private type
private
    type P is new boolean;
    type P2 is new integer;
end test;

package body test is
    type T2 is range 1..3;        -- T2 is a new type
    subtype T3 is T2 range 2..3;  -- T3 is a constraint on T2
end test;
```

```
Location      Instruction                              Comment
0:            VAL    0,0
               OP    MODULE,FIELD_READ_OP,2            -- push "boolean"
              VAL    0,0
               OP    MODULE,FIELD_READ_OP,40           -- push "integer"

1:            VAL    1,3
             DECL    DISCRETE,PUBLIC,DERIVING          -- create T1

              VAL    1,4
              VAR    DISCRETE,VISIBLE,NONE             -- create C1
        SHORT_LIT    23
              STO    1,5                               -- C1 := 23
               OP    DISCRETE,MAKE_CONSTANT_OP         -- make C1 constant

              VAL    1,2                               -- push "boolean"
             DECL    DISCRETE,PRIVATE,DERIVING         -- create P

2:            VAL    1,3                               -- push "integer"
             DECL    DISCRETE,LIMITED_PRIVATE,DERIVING -- create P2
           SYS_OP    ACCEPT_ACTIVATION                 -- end:visible part

        SHORT_LIT    1
        SHORT_LIT    3
             DECL    DISCRETE,LOCAL,DECLARING          -- create T2

        SHORT_LIT    2
        SHORT_LIT    3
3:            VAL    1,8                               -- push "T2"
             DECL    DISCRETE,LOCAL,CONSTRAINING -- create T3

           SYS_OP    SIGNAL_ACTIVATED                  -- end: body
           SYS_OP    SELECT_TERMINATION
```

Figure 4-1: Example of types and protection

non-generic package, a generic package instantiation, a task declaration, or an instantiation of a task type.

4.5.2. Module memory space

There is a program segment associated with each module type. This segment contains the code for the module. There is a control stack associated with each module instance. This control stack contains the statically named objects declared in the visible part of the module and in the module body. The control stack is also used for stack frames of subprogram invocations. If the module is a package, frames created from calls during package elaboration go on the package's control stack. If the module is a task, then subprogram calls made by the task create frames on its control stack (which includes those made during task elaboration).

The base of the control stack contains information about imports into the package, scheduling information, resource utilization and limit information, and state information for the package or task associated with the stack. There are no instructions to directly access this information.

In addition to a control stack, the module instance's memory space includes a type stack, and a data stack. Other segments may be created by the module as it executes. The program segment and import segment of the module instance are those associated with the module type.

The type stack contains type descriptors for types defined by the module. Each object has an associated type and this type is characterized by a reference to its type descriptor on a type stack. The type pointer points to the type stack of the module in which the type is declared. If the type is defined in the same module as the object, then the pointer points to the module's own type stack.

Modules are implemented as closed scopes in the architecture. This requires an import segment for each module type that contains references to objects referenced by a module but declared in another. The import segment provides the initial addressing of the foreign object; once located, other references to it can be placed on the control stack and in other locations. The import segment is structured similarly to a control stack and contains entries that reference objects and types.

4.5.3. Program Segments

There is a program segment for each module of source program. Thus, the program space is divided into a number of program segments. Each program segment has a unique segment name and consists of some header information and one or more subprogram bodies.

The header (accessed at negative offsets from the PS address in the R1000 architecture) is a list of the program segment names of all modules declared immediately within this one. This is where the module finds the program segment names of contained modules when it creates them (see below).

The first subprogram handles module elaboration and is invoked during the module creation process. It is passed any generic parameters if the module is generic.

Each subprogram is broken into four parts. First (the first 8 bytes in the R1000 architecture) comes information used for exception handling. Next is code for elaborating the declarative part. Next comes the code for the subprogram body. Finally comes the exception handling code for the subprogram. Each instruction is 16 bits long in the R1000 architecture.

Long literal constants are intermixed with the code in program segments. They must not be placed in the path of execution or else.

The normal and exception handling code must be separated; the normal code is followed by the exception handling code. This is because when an exception is raised, the current instruction location determines whether the handler chosen will be in the current subprogram or its dynamic predecessor.

4.5.4. Creation and Deletion of Modules

New modules are instantiated through a multi-step process. Tasks and packages are instantiated in much the same way except for a few differences near the end of the process. In the following sequence, 'Parent' refers to the module declaring the new module and 'Child' refers to this new module.

- The Parent executes a VAR instruction to declare the Child. This causes a 'declare' message to be sent to the system which results in the creation of appropriate segments for the Child.

- The Child begins execution by executing the code to elaborate its visible part.

- The Child executes an ACCEPT_ACTIVATION SYS_OP to indicate that the elaboration of its visible part is complete. This causes a 'declared' message to be sent back to the Parent which then continues executing.

- The Parent executes an ACTIVATE_ALL op if the Child is a task, or an ACTIVATE op if the child is a package. This results in an 'activate' message being sent to the Child.

- The Child then elaborates its body and, if it is a package, executes its body. Then, it sends a 'signal activated' message back to the parent. If the Child is a task, concurrent execution begins at this point.

- The Parent continues sequential execution when it receives the 'signal activated' message. If the Child is a task, it continues executing as well.

An example of instructions dealing with modules follows.

4.5.5. Instructions dealing with Modules

DECL	Declare a module type
VAR	Create a module instance
OP FIELD_READ_OP	Read a field of a module (from its visible part)
OP FIELD_WRITE_OP	Write into a field of a module (from its visible part)
OP FIELD_EXE_OP	Invoke a visible field of a module (hopefully a subprogram)
OP ACTIVATE_OP	Elaborate a module
OP ACCESS_ACTIVATE_OP	Elaborate a module pointed to by an access variable
SYS_OP ACCEPT_ACTIVATION	Inform parent that elaboration of visible part is complete
SYS_OP ACTIVATE_ALL	Tell a child task to begin
SYS_OP SIGNAL_ACTIVATED	Inform parent that elaboration of body is complete
SYS_OP NAME_MODULE	Construct a module variable for the current module

4.6. Procedures and Parameters

Subprograms in Ada may be procedures or functions. In addition, the architecture defines a number of other subprograms, most related to tasking. This section discusses Ada procedures and functions primarily, but the general invocation and return sequences apply to most other types of subprograms.

4.6.1. Subprogram Invocation and Return

Subprogram invocations allocate a new stack frame on the control stack of the invoker. This frame contains a marker which contains the following information:

- Subprogram return address
- Data stack frame pointer
- Type stack frame pointer
- Package-level frame pointer (outer block scope)
- Enclosing frame pointer (most recent lexical level) ("static link")
- Current lexical level
- Previous frame pointer ("dynamic link")

The general approach is to create a new stack environment that will be eliminated when the subprogram returns, restoring the current environment. Subprograms may have local package or task objects declared within them. Restoration of the pre-invocation environment requires the termination and deallocation of any local modules.

4.6.2. Resource Control

All resources allocated must be accounted for. This is done by recording allocations on what is called the "child list" which lists resources that must be freed on subprogram return. (Implementation note: the child list is kept on the type stack.)

The specific resources accounted for on the child list are:
- Local tasks
- Local packages
- Collections associated with local access types.
- Import segments associated with local modules types.

The accounting of resources created and deallocation on subprogram return are performed automatically by the instructions that create types and objects (DECL and VAR) and that handle subprogram return (EXIT_PROC, EXIT_FUNC, EXIT_ACCEPT).

4.6.3. Parameter Passing and Return

Parameters are passed on the control stack. The general parameter processing steps are as follows. First, the caller pushes result and value-result parameters. These must be processed after the called subprogram returns. Then, the caller pushes value parameters. These are removed by the called subprogram as part of the return step.

The invocation instruction is then issued. This creates a new stack frame and saves markers to restore the pre-call environment. The subprogram then runs.

When the subprogram executes a return instruction, the resources allocated by the subprogram are freed. If the subprogram is a procedure, then the stack is restored and the value parameters removed. The caller then processes the result (out) parameters.

If the subprogram is a function, the frame pointer for the stacks are restored to their pre-call values, but the top pointers are not restored and the function result is on the top of the stack. It must be processed before the stack is restored. When the statement that executed the function call has completed, it can execute the POP_DATA and POP_TYPE SYS_OP instructions to remove the activation frame of the invoked function.

The procedure calling sequence is:
- Push result parameters (out and in out scalars)
- Push value and reference parameters (in and structures)
- Issue call to the procedure.
    * a new stack frame is created
    * pointers to the tops of the type and data stack are saved
- The called procedure executes
    * parameters are accessed with negative offsets in the current lexical level
- The called procedure returns

    * execution is delayed until any local tasks terminate
    * local resources are freed
    * the stack frame is removed and the data and type stacks are restored to their pre-call points
- The caller pops and stores the result parameters

The function calling sequence is:
- Push value and reference parameters (all in parameters)
- Issue call to the function.
    * a new stack frame is created
    * pointers to the tops of the type and data stack are saved
- The called function executes
    * parameters are accessed using negative offsets in the current lexical level
- The called function returns
    * execution is delayed until any local tasks terminate
    * local resources are freed
    * the return value word on the control stack is popped, the stack marker and in parameters removed, and the return value pushed back on
    * the stack frame pointers are restored
    * the stack top pointers of the data and type stack are left unchanged
    * the caller may, at a later time, flatten (ie, remove any information left by the function) the data and type stack frames

4.6.4. Procedure Variables

They exist. (fill in later)

4.6.5. Subprogram Related Instructions

The following instructions are used to handle subprograms.

SUBPROG	Declare a subprogram variable
CALL	Invoke a subprogram (possibly waiting for a rendezvous)
OP RUN_UTILITY_OP	Invoke a utility subprogram (See Arrays)
OP FIELD_EXE_OP	Invoke a subprogram in another module
SYS_OP CALL_BY_REFERENCE	Call a non-local procedure
EXIT_PROC	Return from a procedure
EXIT_FUNC	Return from a function
EXIT_ACCEPT	Return from a rendezvous subprogram
POP_PROC	Multi-level procedure return
POP_FUNC	Multi-level function return
SYS_OP POP_DATA	Pop the data stack frame after a function return
SYS_OP POP_TYPE	Pop the type stack after a function return

4.7. Blocks

Blocks with no local objects or exception handlers are executed inline. Blocks with local objects or exceptions have the same properties as subprograms without parameters. They exist in their own stack frame. Blocks in Ada are implemented as subprograms without parameters. Thus, a subprogram containing several blocks is transformed into a series of calls to the "subprogramed" version of each block.

```
package body test is
begin
    declare x: integer;                         -- First block
    begin
      x := 1;
    end;
    declare x: integer;                         -- Second block
    begin
      x := 2;
    end;
end test;
```

```
Location     Instruction                        Comment
0:         HEADER     1, 3                     -- Procedure header
          HANDLER     0, 0
           LOCALS     1
           PARAMS     0
              VAL     0, 0
               OP     MODULE, FIELD_READ_OP, 40
           SYS_OP     ACCEPT_ACTIVATION        -- end vis elaboration
        SHORT_LIT     2
1:        SUBPROG     FOR_CALL, HIDDEN         -- 1st block subprog
        SHORT_LIT     4
          SUBPROG     FOR_CALL, HIDDEN         -- 2nd block subprog
             CALL     1, 3                     -- do 1st block
             CALL     1, 4                     -- do 2nd block
           SYS_OP     SIGNAL_ACTIVATED         -- pkg body completed
           SYS_OP     SELECT_TERMINATION       -- pkg completed

2:         HEADER     2, 6                     -- Header:1st block
          HANDLER     0, 0
           LOCALS     1
           PARAMS     0
              VAL     1, 2                     -- Push "integer"
              VAR     DISCRETE, HIDDEN, NONE   -- Create x
        SHORT_LIT     1                        -- x := 1
              STO     2, 2
3:      EXIT_PROC     0                        -- 1st block done
4:         HEADER     4, 6                     -- Header: 2nd block
          HANDLER     0, 0
           LOCALS     1
           PARAMS     0
              VAL     1, 2
              VAR     DISCRETE, HIDDEN, NONE   -- create x
        SHORT_LIT     2                        -- x := 2
              STO     2, 2
5:      EXIT_PROC     0                        -- 2nd block done
```

4.8. Control Transfer

The architecture supports conditional and unconditional jumps within a program segment, and a multi-way branch. The following instructions relate to control transfer:

JUMP	Unconditional control transfer
JUMPF	Conditional control transfer
JUMPT	Conditional control transfer
CASE_JUMP	Multi-way branch based on case index

4.9. Arrays

Arrays are collections of objects each of which has the same type. Each object in an array is called an array element. Each element is identified by one or more array indices, the number of which is the array's dimensionality.

4.9.1. Array Structure

4.9.2. Array Creation

Array types are created with the DECL instruction. This creates a new type, a derived type or a constrained type.

4.9.3. Array Instantiation

Arrays are instnatiated with the VAR instruction applied to an array type. This results in the allocation of sufficient data stack space to hold the array and initialization of this space.

4.9.4. Array Access

Arrays can be accessed as a single object for assignement or passing as a parameter. Sections ("slices") or arrays can be created and manipulated using the SLICE and SLICE_ASSIGN operators. Individual array elements are accessed using the FIELD_READ and FIELD_WRITE operators, giving them the array variable, and an appropriate number of subscripts.

Other type and index range information can be extracted from array variables and types.

4.9.5. Array related Instructions

DECL	Create an array type
VAR	Create an array variable
OP IS_CONSTRAINED_OP	Check if a type is constrained
OP LENGTH_OP	Get the number of elements in an array
OP RUN_UTILITY_OP	Invoke the utility subprogram of an array

```
package test is
        A:      array(1..100) of boolean;
end test;
package body test is
        B:      array (1..10, 20..30) of integer;
        type T is array(INTEGER range <>) of integer;
        C:      T(2..8);
        E:      B'RANGE(2);     -- type is 20..30
begin
        A(2) := true;
        B(3,12) := C(4);
end test;

        VAL         0,-256
        OP          MODULE, FIELD_READ_OP,40        -- INTEGER
        VAL         0,-256
        OP          MODULE, FIELD_READ_OP,2         -- BOOLEAN
-- A: array (1 .. 100) of BOOLEAN;
        SHORT_LIT   1                               -- 1
        SHORT_LIT   100                             -- 100
        VAL         1,3                             -- BOOLEAN
        SHORT_LIT   1
        VAL         0,-256
        OP          MODULE,FIELD_READ_OP,136
        DECL        ARRAY, PUBLIC, DECLARING
        VAL         1,4
        VAR         ARRAY, VISIBLE                  -- A
        SYS_OP      ACCEPT_ACTIVATION
-- B: array (1 .. 10, 20 .. 30) of INTEGER;
        SHORT_LIT   1                               -- 1
        SHORT_LIT   10                              -- 10
        SHORT_LIT   20                              -- 20
        SHORT_LIT   30                              -- 30
        VAL         1,2                             -- INTEGER
        SHORT_LIT   2
        VAL         0,-256
        OP          MODULE,FIELD_READ_OP,136
        DECL        ARRAY, LOCAL, DECLARING
        VAL         1,6
        VAR         ARRAY, HIDDEN                   -- B
```

Figure 4-2: Array Example

```
OP CHECK_CONSTRAINT_OP
OP SLICE_OP             Construct an array slice
OP SLICE_ASSIGN_OP      Assign an array slice
OP CONCATENATE_OP       Concatenate two one-dimensional arrays
OP ELEMENT_TYPE_OP      Get array element type
OP RANGE_TYPE_OP        Get type of an array index
OP FIELD_READ_OP        Read an element of an array
```

```
-- type T is array (INTEGER range <>) of INTEGER;
        VAL             1,2                             -- INTEGER
        OP              DISCRETE, BOUNDS_OP
        VAL             1,2                             -- INTEGER
        SHORT_LIT       1
        VAL             0,-256
        OP              MODULE,FIELD_READ_OP,136
        DECL            ARRAY, LOCAL, DECLARING         -- T
-- C: T (2 .. 8);
        SHORT_LIT       2                               -- 2
        SHORT_LIT       8                               -- 8
        VAL             1,8                             -- T
        VAR             ARRAY, HIDDEN, WITH_PARAM       -- C

-- E: B'RANGE (2);
        SHORT_LIT       1
        VAL             1,7                             -- B
        OP              ARRAY, RANGE_TYPE_OP            -- B'RANGE (2)
        VAR             DISCRETE, HIDDEN                -- E
begin

-- A(2) := TRUE;
        SHORT_LIT       1                               -- TRUE
        SHORT_LIT       2                               -- 2
        VAL             1,5                             -- A
        OP              ARRAY, FIELD_WRITE_OP,0         -- A (2)

-- B(3, 12) := C (4);
        SHORT_LIT       4                               -- 4
        VAL             1,9                             -- C
        OP              ARRAY, FIELD_READ_OP,0          -- C (4)
        SHORT_LIT       3                               -- 3
        SHORT_LIT       12                              -- 12
        VAL             1,7                             -- B
        OP              ARRAY, FIELD_WRITE_OP,0         -- B (3, 12)
        SYS_OP          SIGNAL_ACTIVATED
        SYS_OP          SELECT_TERMINATION
end module
```

Figure 4-2, continued

OP FIELD_WRITE_OP   Write into an element of an array

4.10. Records

4.10.1. Record Structure

Records consist of one or more fields, each of which is a typed object. The type decsriptor for a record contains descirptors for each of the fields.

4.10.2. Record Creation

Record types are created with the DECL instruction. A complete description of the types of all fields must be given at that time.

Variant records are structures that are based on one or more discriminant fields which determine the structure of the remainder of the record. They are similarly created with the DECL instruction.

4.10.3. Record Instantiation

Record instances are created with the VAR instruction. The discriminant fields must be defined at that point. If it is possible to assign to the record, the maximum space must be allocated to the record so that its largest variant can be accomodated.

4.10.4. Record Access

Record fields are accessed with the FIELD_READ and FIELD_WRITE operations. These read or write the contents of a specified record field.

4.10.5. Record related Instructions

DECL	Create a record type
VAR	Create a record variable
OP CHECK_CONSTRAINT_OP	Check the constraint on a variant
OP SET_VARIANT_OP	Set the variant of a record
OP FIELD_READ_OP	Read a field of a record
OP FIELD_WRITE_OP	Write into the field of a record
OP FIELD_TYPE_OP	Get the type of a record field

4.10.6. Access Types

4.10.7. Access Structure

4.10.8. Access Creation

4.10.9. Access Instantiation

4.10.10. Access Operation

The only operation on objects of access type are assignment, equality test, and dereference.

```
package body test is

type R is record          -- A record type
         F1: integer;
         F2: integer;
         F3: boolean;
     end record;

Example: R;               -- A record variable

begin
     Example.F3 := true;
     Example.F2 := Example.F1;
end test;

         VAL          0,-256
         OP           MODULE,FIELD_READ_OP,2          -- BOOLEAN
         VAL          0,-256
         OP           MODULE,FIELD_READ_OP,40         -- INTEGER
         SYS_OP       ACCEPT_ACTIVATION

-- type R is
         VAL          1,3                             -- INTEGER
         VAL          1,3                             -- INTEGER
         VAL          1,2                             -- BOOLEAN
         SHORT_LIT    3
         VAL          0,-256
         OP           MODULE,FIELD_READ_OP,136
         DECL         RECORD,LOCAL,DECLARING          -- R

-- EXAMPLE: R;
         VAL          1,4                             -- R
         VAR          RECORD,HIDDEN                   -- EXAMPLE
begin

-- EXAMPLE.F3 := TRUE;
         SHORT_LIT    1                               -- TRUE
         VAL          1,5                             -- EXAMPLE
         OP           RECORD,FIELD_WRITE_OP,2         -- F3

-- EXAMPLE.F2 := EXAMPLE.F1;
         VAL          1,5                             -- EXAMPLE
         OP           RECORD,FIELD_READ_OP,0          -- F1
         VAL          1,5                             -- EXAMPLE
         OP           RECORD,FIELD_WRITE_OP,1         -- F2
         SYS_OP       SIGNAL_ACTIVATED
         SYS_OP       SELECT_TERMINATION
end module
```

Figure 4-3: Record Examples

Note: The code for this example is not quite right.
Details pending.

```
package body test is
    type Ind is access integer;
    A: Ind;
    B: integer;
begin
    A := new integer;          -- Allocate a new integer
    A.all := 5;
    B := A.all;
end test;

        VAL         0,0
        OP          MODULE, FIELD_READ_OP,40  -- INTEGER
        SYS_OP      ACCEPT_ACTIVATION

--   1,1   type IND is access INTEGER;
        DECL        ACCESS, LOCAL, DECLARING  -- IND
        VAL         1,2                       -- INTEGER
        VAL         1,3
        LIT         16777216
        SHORT_LIT   0
        OP          ACCESS, COMPLETE_OP       -- IND

--   1,7   A : IND;
        VAL         1,3                       -- IND
        VAR         ACCESS, HIDDEN            -- A

--   1,8   B : INTEGER;
        VAL         1,2                       -- INTEGER
        VAR         DISCRETE, HIDDEN          -- B
begin

--   1,10  A := new INTEGER;
        VAL         1,3
        VAL         1,2                       -- INTEGER
        OP          ACCESS, NEW_OP            -- new INTEGER
        STO         1,4                       -- A

--   1,11  A.all := 5;
        SHORT_LIT   5                         -- 5
        VAL         1,4                       -- A
        OP          ACCESS, ALL_WRITE_OP      -- A.all

--   1,12  B := A.all;
        VAL         1,4                       -- A
        OP          ACCESS, ALL_READ_OP       -- A.all
        STO         1,5                       -- B
        SYS_OP      SIGNAL_ACTIVATED
        SYS_OP      SELECT_TERMINATION
end module
```

4.10.11. Access related Instructions

DECL	Create an access type
VAR	Create an access variable
OP NULL_OP	Create a null pointer
OP NEW_OP	Allocate a new object of the specified type
OP ALL_READ_OP	Dereference a pointer
OP ALL_WRITE_OP	Assign to access referent
OP ALL_REF_OP	Construct a reference to an indirect scalar

4.11. Tasking

Tasking prvides a fcaility for the creation, management, and synchronization of multiple threads of control through a program.

4.11.1. Task Instances

Tasks may be instantiated by eloborating the declarative part of a package, task, subprogram, or block. Tasks may also be created dynamically through a task type in Ada. The instructions used are outlined in the section on modules in the first case, and NEW_OP and ACCESS_ACTIVATE_OP in the latter.

4.11.2. Task Instantiation

Tasks are instantiated by using the VAR instruction on a task type. Ada tasks and packages are treated almost identically. The sequence of steps described in the section on modules covers the important aspects of task intantitation.

4.11.3. Task Initiation

4.11.4. Rendezvous

```
task test is
    entry foo;                                 -- Two visible entries
    entry bar;
end test;

task body test is
begin
    accept foo do null; end foo;               -- A simple rendezvous
    accept bar do null; end bar;
end test;

--  1,19  entry FOO;
        SHORT_LIT    0
        VAR          ENTRY,VISIBLE             -- create FOO

--  1,8   entry BAR;
        SHORT_LIT    0
        VAR          ENTRY,VISIBLE             -- create BAR
        SYS_OP       ACCEPT_ACTIVATION

--  1,10  accept FOO do
        SHORT_LIT    2
        <push ref to subprog #1>               -- create a subprogram
        SUBPROG      FOR_ACCEPT,HIDDEN         -- to handle the accept

--  1,10  accept BAR do
        SHORT_LIT    3
        <push ref to subprog #2>
        SUBPROG      FOR_ACCEPT,HIDDEN
        SYS_OP       SIGNAL_ACTIVATED
begin

--  1,10  accept FOO do
        REF          1,4
        REF          1,2
        OP           ENTRY,RENDEZVOUS_OP

--  1,16  accept BAR do
        REF          1,5
        REF          1,3
        OP           ENTRY,RENDEZVOUS_OP
        SYS_OP       ACCEPT_TERMINATION

begin -- foo rendezvous body subprogram   -- body goes here
        EXIT_ACCEPT 0
end subprogram

begin -- bar rendezvous body subprogram   -- body goes here
        EXIT_ACCEPT 0
end subprogram
end module
```

4.11.5. Select

A special subprogram is created to handle the select statement.

4.11.6. Termination

Tasks terminate when they are aborted explicitly, reach their last statement, or select termination in concert with their siblings, parents, and children.

4.11.7. Task related Instructions

SUBPROG	Create a subprogram variable for various tasking functions
OP IS_TERMINATED_OP	Determine if a task is terminated
OP COUNT_OP	Determine number of queued entry calls
OP FIELD_GUARD_OP	Set guard field of a select structure
OP ENTRY_CALL_OP	Peroform an entry call
OP COND_CALL_OP	Perform conditional entry call
OP TIMED_CALL_OP	Perform timed entry call
OP FAMILY_CALL_OP	Perform an entry call to a family member
OP FAMILY_COND_OP	Perform a conditional entry call to a family member
OP FAMILY_TIMED_OP	Perform a timed entry call to a family member
OP DELAY_OP	Delay for a specified time

4.12. Exception Handling

4.12.1. Raising exceptions

4.12.2. Location of exception handlers

4.12.3. Interrogation of exception information

4.12.4. Machine-Raised Exceptions

OPERAND_CLASS_ERROR
VISIBILITY_ERROR
CAPABILITY_ERROR
CONSTRAINT_ERROR
TYPE_ERROR
UNIMPLEMENTED_OPERATION
ILLEGAL_INSTRUCTION

4.12.5. Exception related Instructions

OP RAISE_OP	Raise an exception
OP RAISED_EXCEPTION_OP	Determine number of raised exception

```
package body test is
    Mistake: exception;
    Val:     integer;
begin
    Val := 3;
    raise Mistake;                                  -- Raise an exception
    null;
exception
    when Mistake => Val := 1;
    when others  => null;
end test;
        VAL         0,0
        OP          MODULE, FIELD_READ_OP,40 -- INTEGER
        SYS_OP      ACCEPT_ACTIVATION
--  1,1   MISTAKE : exception;
--  1,5   VAL : INTEGER;
        VAL         1,2                     -- INTEGER
        VAR         DISCRETE, HIDDEN        -- VAL
--  1,7   VAL := 3;
        SHORT_LIT   3                       -- 3
        STO         1,3                     -- VAL
--  1,8   raise MISTAKE;
        LIT         310                     -- value of "Mistake"
        VAR         EXCEPTION, HIDDEN       -- create exception var
        OP          EXCEPTION, RAISE_OP
--  1,9   null;
        SYS_OP      SIGNAL_ACTIVATED
        SYS_OP      SELECT_TERMINATION
exception
--  1,11  when MISTAKE =>
        LIT         310
        VAR         EXCEPTION, HIDDEN
        VAL         1,4
        OP          EXCEPTION, EQUAL_OP
        JUMPF       6
--  1,11  VAL := 1;
7:      SHORT_LIT   1                       -- 1
        STO         1,3                     -- VAL
        SYS_OP      SIGNAL_ACTIVATED
        SYS_OP      SELECT_TERMINATION
--  1,12  when others =>
--  1,12  null;
6:      SYS_OP      SIGNAL_ACTIVATED
        SYS_OP      SELECT_TERMINATION
end module
```

Figure 4-4: Exception Example

```
    OP RAISED_ADDRESS_OP  Determine address where exception was raised
    OP RAISED_SCOPE_OP    Determine scope where exception was raised
```

4.13. Program Creation

4.13.1. Program creation related Instructions

```
OP LOAD_OP             Create a prorgam segment
OP SEGMENT_NUMBER_OP   Get unique integer for segment
```

4.14. Re-elaboration

Table of Contents

List of Figures