```
  IIIIII      NN      NN    TTTTTTTTTT    RRRRRRRR        OOOOOO
  IIIIII      NN      NN    TTTTTTTTTT    RRRRRRRR        OOOOOO
    II        NN      NN        TT        RR      RR    OO      OO
    II        NN      NN        TT        RR      RR    OO      OO
    II        NNNN    NN        TT        RR      RR    OO      OO
    II        NNNN    NN        TT        RR      RR    OO      OO
    II        NN  NN  NN        TT        RRRRRRRR      OO      OO
    II        NN  NN  NN        TT        RRRRRRRR      OO      OO
    II        NN    NNNN        TT        RR  RR        OO      OO
    II        NN    NNNN        TT        RR  RR        OO      OO
    II        NN      NN        TT        RR    RR      OO      OO
    II        NN      NN        TT        RR    RR      OO      OO
  IIIIII      NN      NN        TT        RR      RR      OOOOOO
  IIIIII      NN      NN        TT        RR      RR      OOOOOO


TTTTTTTTTT    XX      XX    TTTTTTTTTT                    222222
TTTTTTTTTT    XX      XX    TTTTTTTTTT                    222222
    TT        XX      XX        TT                      22      22
    TT        XX      XX        TT                      22      22
    TT          XX  XX          TT                              22
    TT          XX  XX          TT                              22
    TT            XX            TT                            22
    TT            XX            TT                            22
    TT          XX  XX          TT                          22
    TT          XX  XX          TT                          22
    TT        XX      XX        TT           ....         22
    TT        XX      XX        TT           ....         22
    TT        XX      XX        TT           ....       2222222222
    TT        XX      XX        TT           ....       2222222222


*START* Job INTRO Req #805 for EGB    Date 25-Aug-82 12:08:10 Monitor: Rational
File RM:<SIM.DOC.PATENT>INTRO.TXT.2, created:  9-Aug-82  7:25:03
        printed: 25-Aug-82 12:08:15
Job parameters: Request created:25-Aug-82 12:08:09   Page limit:36   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:AS
```

INTRODUCTORY DESCRIPTION OF THE INVENTION

1. Runtime Model

While machine architectures are traditionally characterized by their instruction set, the architecture of the R1000 is best understood by examining the runtime representation of programs. There are several reasons for emphasizing the runtime representation rather than the instruction set. First, many of the frequently executed expression evaluation instructions can be understood in isolation, but the majority of the instructions can only be understood in terms of how they modify the runtime state and what runtime invariants they preserve.

A second major reason for emphasizing runtime representation of programs is that modern languages such as Ada are very declarative languages which place considerable emphasis on type and object declarations. Such languages require large amounts of information to be kept at runtime to represent complex objects and implement the dynamic semantics of type checking. A key feature of the R1000 architecture is its support for the declarative nature of modern languages. By including a complete representation of type declarations at runtime, the R1000 supports a very space efficient representation of objects and accelerates addressing objects and object components, eliminating many space/time tradeoffs required on other machines. The runtime representation of types and objects in the R1000 also allows direct hardware support for efficient dynamic type checking.

A third reason for emphasizing runtime representation of programs is that modern languages include very powerful control regimes, including simple procedures and functions, recursion, retentive control (module variables are not deallocated on leaving the module), coroutining, and true parallelism. While the design of modular and maintainable software is greatly enhanced by using such powerful control structures, the control and storage management overhead associated with such control structures can be prohibitively expensive. An effective runtime representation for supporting modern languages must address activation record construction, parameter transmission, maintenance of the storage structure in which the activation records reside, searching the storage structure for space for activation record allocation (in many runtime models) and the reclamation of unused storage within the structure. The R1000 design is based on a runtime representation which addresses these issues by closely modelling the semantics of modern languages. By exploiting the semantics of such languages, the runtime representation provides near optimal performance in terms of both space and time.

A fourth reason for focusing on runtime representation is the issue of protection and security. Modern languages place considerable emphasis on scope and visibility. Modern programming practices based upon encapsulation and abstraction depend upon these scope and visibility rules to enforce modularity and control interfaces. While security and protection have been important issues in machine architecture for several years, most recent efforts have focused on general purpose protection schemes that are unrelated to programming language semantics. The R1000 architecture gaurantees security (in the sense that a running programming cannot corrupt another program) through enforcement of the encapsulation and abstraction semantics of modern languages. This means that the protection facilities of the machine correspond exactly to the programmer conception of how objects are protected, allowing the programmer to write reliable and secure software. The R1000 runtime model is the key to enforcing this protection model with no performance penalty.

A final reason for emphasizing the runtime representation rather than the instruction set is that the role of the instruction set in the R1000 is much different from that of a conventional machine. On most machines the instruction set provides the interface between software and hardware. The instruction set defines the programmer's view of the machine and is the vehicle for insuring software compatibility between different machines in the same family. RMI intends that the Ada language serve this role, rather than the instruction set of any particular machine implementation. There will be no facilities for programming the R1000 in machine (or assembly) language. The Ada language provides the user view of the machine, and provides software compatibility. While other languages may be supported on the R1000, it is anticipated that the majority of those languages would be translated to an extended Diana representation and then compiled with the Ada compiler. This approach allows maximum freedom in tailoring the basic instruction set to a given hardware technology and target market, without sacrificing a consistent software interface.

This section introduces the R1000 runtime model and describes the manner in which the runtime representation corresponds to Ada semantics. Only those features of Ada semantics which have substantial impact on the runtime model are addressed in this section.

2. Basic Runtime Structures

The basic component of Ada programs is the package. The package is the fundamental mechanism for modularizing programs, encapsulating types and data, and defining abstractions. The Ada task also provides modularity and encaspulation, in addition to implementing concurrency (including synchronization and communication). Correspondingly, the module (package or task) is the fundamental unit in the R1000 runtime model. Figure ? illustrates the basic runtime structure of a module.

Every module instance has a unique name and a set of address spaces that are accessed with that name. Address space creation and deletion is managed automatically in the R1000. The segment name generally consists of two parts, a virtual processor number and a segment identifier. The R1000 virtual address space is divided into 256 virtual processors for purposes of load leveling and crash recovery. A module is always associated with the same virtual processor, but the mapping between virtual and physical processors may be changed over time.

Each module instance includes a control stack (used for expression evaluation, parameter transmission, and activation records), a type stack (use for type descriptors of types declared in that package), and a data stack (used for data storage for all data structures declared in that module). A task also includes a queue address space which contains the message queues for the entries of that task. Each module instance makes use of an import segment associated with each instance of the module type (there may be many modules of the same type). For a single source definition of a module there may be several elaborated instances of the module type, but they all share a single program segment. Within the architecture virtual addresses consist of a memory sort (control, type, data, queue, import, or program), a segment (or address space) name and an offset within the segment. We will briefly describe the structure of each address space.

2.1 Program Segments.

The first words of the program segment are reserved for holding program segment names. The very first word contains a pointer to the module body start address and then the first three program segment names. Four program segment names may be stored in successive words. Program segment names must be included for every module type to be declared when executing the subject program segment.

The remainder of the code segment is organized into sections. The first section contains code for the module body, the other sections contain code for any subprograms declared in the module. Each section consists of some exception handling information followed by the instructions (eight per word) and literals for the declarative part, statement part, and exception handlers (in order).

The exception handling information includes the number of declared locals and the boundaries between declarative, statement and exception handling code. This information allows the machine to determine whether an exception should be propogated immediately (if the exception occured in the declarative or exception handling code) or whether it should be handled locally after cleaning up any partially evaluated expressions. This exception handling information effectively takes up the space of the first three instructions of each section. This leaves room for only five instructions in the first word of a section.

A program segment address includes a 24-bit segment number and a 15-bit instruction address (2**12 words). Relative program segment addresses are included in the jump and long literal instructions (since literals are stored in the program segment). An absolute start address is pushed on the stack when declaring a subprogram variable. Otherwise there are no instructions for addressing code segments.

2.2 Import Segment

In the Ada language each module is an open scope; that is, all objects declared in an outer package before the declaration of an inner package are visible within that inner package. For improved security and efficiency and to support languages other than Ada, the R1000 treats modules as closed scopes. From the point of view of the instruction set, the basic addressing mechanism involves refering to objects on the control stack by a static lexical level (0..15) and relative offset. Lexical level 1 corresponds to the outer frame of a module. Lexical level 0 refers to the import segment associated with the module type. Thus the import space can be viewed as a shared outermost control stack frame for all modules of the same type.

When a parent module elaborates the declaration of a module type visible part, it first pushes a list of objects (types and variables) on its control stack. These objects are imported by the child module, and are therefore placed in an import segment created for that module type. Further imports may be provided when elaborating the declaration of the module type body. This approach gives the parent explicit control over the portions of the current environment which are available to the child. Thus the import space can also be viewed as an access rights list.

An import segment address includes a 32-bit segment name and a 9-bit word offset. Import addresses are generated by instructions which refer to lexical level 0.

2.3 Control Stacks.

The control stack is the primary runtime structure for a module instance. All words on the control stack are tagged. A control stack consists of a task control block followed by a series of activation records. The task control block includes information about the state of the module, resource limits, scheduling, debugging, context information, microstate save areas, etc. This information is maintained automatically in the R1000.

Each activation record consists of a two word mark followed by one word for each object in the frame. The mark words include the static link and outer frame pointer (forming an abbreviated display), dynamic link, lex level, return address, and other information. Activation records are built automatically in the R1000. The call and exit instructions save and restore frame state and implement Ada semantics for subprogram call and exit. As mentioned above, all addressing is relative to the currently visible activation records. The R1000 automatically maintains the abbreviated display and performs address computations (converting lexical level and delta into a control stack address) invisibly. Parameters are pushed on the stack before invoking a subprogram and are addressed with negative offsets from the mark words.

All types and statically named variables are allocated one word on the control stack. Most objects on the control stack consist of a 64-bit value part and a 64-bit type part. For a type object, the value part is meaningless. The type part of the word includes a tag indicating the class (package, task, discrete, float, array, record, etc.), information about visibility and protection, and a link to a full type descriptor on a type stack. If the type was declared in the same module as the object, then the type link points to the type stack with the same segment name as the control stack. The contents of the value part of a control word depends upon the class of object. For discrete, float, access, package and task objects, the value is represented directly in the control word. For records and arrays, the value part is a data stack reference, pointing to the location of the data structure on the data stack.

A control stack address includes a 32-bit segment name and a 20-bit word offset. Control stack addresses are generated by instructions which include a lex level and delta. All expression evaluation and parameter transmission implicitly uses the top of the control stack. All objects are loaded on the top of the control stack before being manipulated. Content specifications for all of the words on the control stack can be found in <sim.def>stacks.ada. Precise bit formats can be found in <sim.doc>control-word.format.

2.4 Type Stacks.

The type stack contains descriptors for the types declared in the corresponding module. A single type descriptor is shared by all instances of an object. Type descriptors include information such as the bounds of a scalar type, index bounds and addressing constants for an array type, field type and location for a record field, etc. Type descriptors provide support for address computations, dynamic constraint checking, protection, type conversions, and implementing various attributes.

Like the control stack, the type stack is organized into frames corresponding to the outer package and any subprogram activations. There are no mark words on the type stack (top and frame information for the type stack are in the mark words on the control stack).

In addition to holding type descriptors, the type stack holds the list of the names of the children created by a frame. Children include any modules, collections, or import spaces declared in that frame. Basically this is the list of address spaces which must be reclaimed when exiting a frame. For the outer frame of a package, one of the module control block words points to the first child word in that frame. For subprogram activations, the first word of the type stack frame is reserved for the first child word. Since the number of children being declared is not known until complete elaboration of the frame, the child words are scatter between the type descriptors, and are connected in a linked list. Thus each child word includes a link, a count of the number of children in this word (0..3) and the names of up to three children.

Formats for all of the words on the type stack can be found in the package stacks (<sim.def>stacks.ada). The package descriptors (<sim.def>dscrpt.ada) gives an Ada description for most of the type descriptors on the type stack.

A type stack address includes a 32-bit segment name and a 20-bit word offset. The type stack is fixed format and word aligned to allow direct hardware support. Type stack addresses never appear explicitly in instructions, but are generated implictly as required.

2.5 Data Stacks.

The data stack is organized as a string of zero to 2**32 bits and is completely uncontainerized. Data structures (records and arrays) are stored on the data stack and are accessed through variables and expression temporaries on the control stack. All data structures are interpretted in accordance with the appropriate type descriptor. Sizes and intermediate addresses are computed when the type descriptor is elaborated, based on any dynamic constraints. The sizes represent the minimum number of bits required to represent the object, and the object is stored in this minimal representation on the data stack.

A data stack address includes a 32-bit segment name and a 32-bit bit offset. Data stack addresses never appear explicitly in instructions, but are generated implictly as required.

2.6 Queue Segments.

A queue segment is associated with every task instance. The queue segment is organized into a series of linked lists of messages and free space. Each entry variable on the control stack (or on the data stack if it is a member of an entry family) contains a pointer to the head of the corresponding free list and pointers to the head and tail of the corresponding message list. The first word in the queue address space is reserved for resource management information (current extent of the address space, maximum extent permissible, etc.).

A queue address includes a 32-bit segment name (corresponding to the name of the associated task) and a 20-bit word offset. Queue addresses never appear explicitly in instructions, but are generated implictly as required.