```
RRRR    U   U  N   N  TTTTT   III   M   M  EEEEE        M   M   OOO    DDDD
R   R   U   U  N   N    T      I    MM MM  E            MM MM  O   O   D   D
R   R   U   U  NN  N    T      I    M M M  E            M M M  O   O   D   D
RRRR    U   U  N N N    T      I    M   M  EEEE  -----  M   M  O   O   D   D
R R     U   U  N  NN    T      I    M   M  E            M   M  O   O   D   D
R  R    U   U  N   N    T      I    M   M  E            M   M  O   O   D   D
R   R   UUUUU  N   N    T     III   M   M  EEEEE        M   M   OOO    DDDD



TTTTT  X   X  TTTTT          1    222   999
  T    X   X    T           11   2   2 9   9
  T     X X     T            1       2 9   9
  .      X      T            1      2   9999
  T     X X     T            1     2       9
  T    X   X    T      ..    1    2        9
  T    X   X    T      ..   111  22222  999


*START* Job RUNTIM Req #115 for EGB    Date 23-Jul-82 21:38:38 Monitor: Rational
File RM:<SIM.DOC.ARCHITECTURE>RUNTIME-MODEL.TXT.129, created: 14-Apr-82 10:30:33
        printed: 23-Jul-82 21:38:39
Job parameters: Request created:23-Jul-82 21:30:27   Page limit:54   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:A
```

R1000 RUNTIME MODEL

1. Introduction

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. This document is intended to introduce the basic concepts of the R1000 architecture and to document some of the higher level design decisions that are not easily extracted from the architecture simulation. At this point the architecture simulation is the primary reference document for detailed information about the architecture.

It should be noted that a thorough understanding of Ada semantics is a prerequisite to understanding the R1000 architecture. The reader is strongly encouraged to carefully study the Ada language reference manual and to write a couple of reasonably large programs before approaching the R1000 architecture.

We will first introduce the basic runtime structures in the R1000 and then look at specific architectural issues through a series of examples. Each example includes an Ada source program, the R1000 assembly code, and a script with explanatory narrative for running the example on the architecture simulation (the default directory for all filenames given in this document is <sim.doc.architecture>).

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 1 (see Daisy.Arch.Intro) 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. Formats 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.

3. Simple Type and Object Declarations.

This section presents a very simple example designed to introduce the basic principles for declaring types and objects on the R1000. The source code for this example can be found in EX1.ADA, the assembly code can be found in EX1.ASM, and EX1.PHO is a photo file recording a session executing this example on the simulation. Hardcopies of these three files are probably required to make it through the rest of this section. The photo file may be used to assist in following the script for this example and seeing the exact commands to enter.

The source code for this example is very simple. We have a package EXAMPLE_1 with no visible part (a main program on the simulation is usually a package with no visible part). In the body of EXAMPLE_1 we declare a new integer type and five variables of that type. We also declare a character variable used for input/output. We then perform a number of simple computations on the local variables, printing out the final result.

If we look at EX1.ASM (created by running the Ada compiler on EX1.ADA) we see a close correspondence between the source code and the object code. We will explain the operation of each instruction by stepping through the execution of this example on the simulation.

Before running the simulation at 9600 baud, insure that Xon/Xoff is enabled on your terminal. The simulation is invoked by typing "simulation" (for now use the version <sim.tests>simula.new instead).

It takes several seconds for the simulation to initialize all of its component packages, and then it will display a prompt character (}->).

The set of simulation commands can be displayed by typing "?" at this point. If we wish to record a transcript of the session we type "photo" (note that the simulation command interpretter will complete commands upon entry of a unique prefix; here we give full command names for readability).

Before executing EXAMPLE_1, we must first load the program segment into the simulation using the command "load <sim.doc.architecture>ex1". This resolves all jump addresses in the ASM file and produces a program segment address space in the simulation. On the R1000 there will be software tools for building a binary image of a program segment (as a data structure) and then special operations on the SEGMENT_VARs to convert the data structure into a program segment. We can see the results of this operation on the simulation by using the command "show program status" to see that a program segment has been created with a number given to identify that segment.

We can examine the loaded segment with the command "words program <segment-number> 0" followed by a space to see each additional word and a carraige return (CR) to stop. We can see that the format of the program segment is exactly as described above (in section 2.1).

To start the example program we now enter the "initiate <segment-number>" command. This elaborates the package EXAMPLE_1, building all required runtime structures. We can see that a control stack was created by using the "show control segments" command. Control stack zero is created by the simulation to act as an outermost main package. Control stack one is the package standard, also created by the simulation. Control stack 16 is the control stack created by the initiate command.

We can now look at the state of the processor with the "show program registers" and "show control registers" commands. These two commands can be used at any time during the execution of the simulation to orient the user and see the current state of simulation.

The "show program registers" command displays the current value of the program counter and the current value of the instruction buffer.

The "show control registers" command displays the name of the control stack we are executing on, the top of that stack, and some statistics on the control stack accelerator register file. The statistics gathering stuff is not enabled in this version of the simulation.

We can look at the contents of the control stack with the "words control <current-stack> <word-offset>" instruction. Using a word offset of zero and then hitting space for each additional word, we can see all of the control information at the base of the stack. We will come back to the set of words at the bottom of the stack in later examples. All of the control words are described more completely in <sim.defs>stacks.ada.

We can execute the first instruction by entering the "continue" command. The first instruction was a load 0,0, which has loaded a package variable on top of the stack. We can see (using the word command to display the top word of the control stack) that this is a package variable refering to stack #1 (everything in the simulation is on processor 1), the package standard. The first 6 instructions of EXAMPLE_1 are concerned with getting local copies of standard types (string, character and integer). If we enter "continue", the simulation will execute the next instruction, a field read instruction which will read the nth object out of the visible part of the package identified by the package variable on top of the stack. If we now look at the control registers ("show control registers") we see that the top of the stack is still at 16. Looking at the top word on the stack ("word control 16 16") we see that the package variable has been consumed and replaced by the type string (an array variable with no value part).

Now we should "set instruction_trace foo" (to display each instruction as it executes) and "set state_display (to automatically display frequently useful state) and then we can execute the next two instructions with a single command ("step 2"). We can see that the top has moved up one, and we can see that a discrete variable has been placed on top of the stack to represent the type character. We can trace the type link by using the "word type <stack and offset out of the type link>" command. We see from the type descriptor on the type stack of package standard that character is a discrete type with bounds 0..127.

We can execute the next two instructions ("step 2"), again see that the control stack top has moved up one, and see that the item on top of the stack is a discrete with bounds INTEGER'FIRST..INTEGER'LAST.

The next four instructions declare the local type ANSWER. The first three instructions push parameters to the declaration, and the last consumes the parameters, replacing them with the new type. If we "step 3" and then look at "words control 16 19" (hit space bar twice to get second and third parameters) we see that the two bounds have been pushed on the stack and we have copied the type INTEGER to the top of the stack. If we then "continue" (or "step 1") the DECLARE_TYPE instruction checks that the two bounds are within the bounds of the parent type, builds a type descriptor on the type stack, and pushes the new type on top of the control stack (after removing the three parameters). The new type is on top of the stack, and we can see that its type link points to this stack (all of the previously examined type links pointed to types on the package standard). If we then look at the first three words of the type stack we see that a full type descriptor has been built for the new discrete type (see <sim.defs>dscrptr.ada), including the bounds and the exact size for objects of that type. The type utility is a subprogram associated with a type for use by the compiler to implement initialization, complex attributes, and other functions.

The next ten instructions declare five variables of type answer. It should be noted that there are more optimal code generation sequences which are more than twice as fast. If we step ten and then look at the control stack we can see that five variables have been created and left on top of the stack.

The next two instructions ("step 2") load the type character onto the top of stack and then declare a variable of that type.

The next two instructions load a literal on the stack and then store it into the variable A (performing constraint checking). We can step 2 and then look at A using the resolve reference command. A is at lex level 1, delta 7 (counting up from 0 and leaving two words for the frame mark). We can then either step 4 and see the value for B (lex level 1, delta 8), or step 1 four times to see the two operands loaded, the result computed and then stored. Similarly we can step 4 more each to see C and D (lex level 1, deltas of 9 and 10). Finally if we step 4 more we can see the final answer stored into E (lex level 1, delta 11).

The next several instructions are designed to output the answer (E). Input/output in the simulation is modelled as a special package in the package standard with visible operations for I/O. Rather than step some number of times, we can say "BREAK <program segment number> <word-offset> <instruction-index>" to set a break point right after the last print statement. We then say something like step 1000, and the simulation will run until it hits the break point. If we then say "exhibit", the output buffer is displayed. If we then step 100, the simulation runs until it reaches the input statement (which also forces the output buffer to print). If we enter some character, the example runs to completion. We will address the SIGNAL_ACTIVATED and SIGNAL_COMPLETION instructions in a later example.