R1000 Architecture

Preface

This is an overview of the R1000 architecture. It is a document that can be used by new employees and will be used as source material to be included in other documents such as the system overview manual (maybe others). At some point, this document may be cleaned up so that it can be given to customers. But for the moment, it is internal only.

The reader for this document is employees in the development, technical support areas, or manufacturing. The reader is assumed to be familiar with current mini-computers or microprocessors.

This is a company proprietary document! It will not be distributed outside of Rational!

Contents

Chapter 1: Introduction to the Rational Architecture

Rational took a fresh clean look at computer architectures when it began designing its products. Typical computer architectures, whose underlying characteristics were first attributed to John von Neumann, were first designed in the 1940's. While some advances have been made, very few current architectures are significantly different from those of 30 years ago.

Classic von Neumann architectures provide a simple homogenous address space and generalized addressing structure. They were fine when software was inexpensive and hardware was expensive before the advent of recent high-level languages. Classic architectures provide enormous flexibility and simplicity at the expense of reliability and ease of expressing complex data structures or algorithms.

Now it has become clear that the real expense in current computer systems is not the hardware but the software. Recent high-level languages have been built to improve the reliability, maintainability, ease of expression, and efficiency of the software written in those languages which reduces the cost of that software. Now it is time for architectures to reflect those same high-level language ideas.

In 1984, Rational began producing a series of machines based on a new architecture. The application of these machines is for system design using the Rational Design Environment. The RDE is an advanced computer-aided system design tool which vastly increases the productivity of programmers and other designers. The RDE requires a machine whose architecture optimizes the use of recent high-level languages.

Recent programming languages like Ada[1] have incorporated the best ideas for improving reliability, maintainability, ease of expression, and efficiency. The Rational

[1]Ada is a registered trademark of the United States Department of Defense.

architecture is designed for only Ada and Ada-like languages so many architectural features are derived from a corresponding Ada construct.

For instance, Ada has a very easy and efficient multitasking capability which encourages programmers to use multiple tasks. Yet few current von Neumann architectures provide for efficient tasking; some machines require tens of milliseconds to rendezvous with another task.

The Rational architecture provides for direct tasking support built into the machine. On the R1000, the first implementation of the Rational architecture, task rendezvous typically takes a few microseconds.

Ada and other recent languages also encourage the use of complex data types. For example, variant or discriminated records allow a program to treat a set of data as a unit while also allowing access to individual (and possibly quite interdependent) items within that unit. Fulfilling all of the type checking and other constraints that Ada imposes on an object of that complexity often requires a significant number of simple operations in most von Neumann architectures.

In the Rational architecture, manipulating variant records are single operations just like manipulating integers. The handling of complex data structures is inherent in the architecture. This is another example of how the Rational architecture provides simple, direct and efficient support for high-level language constructs.

An architecture built for the purpose of supporting high-level languages provides two key advantages:

- Efficiency – The architecture knows to do much more with each operation since the architecture knows much more about the program it's running. Also the operations can more directly match the language operations. As mentioned, tasking support is integral to the architecture, not system calls or subroutines.

- Reliability – The architecture knows not to do operations that don't make sense in the context of the program it is running. This is the simpliest and most reliable form of protection. This protection (e.g., type checking, privacy checking) is also integral to the architecture.

These advantages mean that the Rational architecture provides better facilities for building and running software written in languages like Ada than most other architectures.

Here are some of the more important features of the architecture that make these advantages so significant:

- Support for all complicated data structures.
- Stack based architecture.
- Lexical addressing structures.
- Multiprocessor support.
- High level tasking support.
- Highly segmented virtual memory.

The remainder of this chapter describes these features.

Complicated Data Structure Support

All values within the machine are ***tagged***. These tags not only specify what representation class the value is (e.g., integers, floating point, variant record), but also what ***type*** the value is as in the Ada sense of types (bounds, structure, etc.). No value exists in the architecture that does not have an explicit type. Therefore, no operation can occur on a value of an illegal or non-sensical type.

Stack Based Architecture

Many machines use stacks for activation records and evaluation of expressions but most make the use of the stack cumbersome and inefficient. Operations in the Rational architecture implicitly use a stack. Further, since Ada encourages the employment of modern software engineering techniques including high modulization, most programs will be highly modular with many tasks and subprograms that require efficient manipulation of the stacks. The Rational architecture is designed specifically for high stack usage.

Lexical Addressing Structure

The addressing structure used in the architecture knows that all programs written in Ada-like languages exhibit *locality* in its use of program variables. Locality suggests that, for the most part, only those variables declared in the local procedure block or enclosing blocks will be used. Thus, the addressing structure can be tailored to provide easy access to those variables that are declared on the current record of execution (the current stack). The addressing structure, which is called *lex-level delta*, uses the lexical level at which the addressed object was declared as the basis for addressing all objects in the architecture.

Multiprocessor Support

The Rational architecture allows implementations to take advantage of multiprocessing techniques. The distribution of virtual memory addresses on distinct processors is an implementation issue not specified by the architecture. Memory references to locations contained in other processors is handled by the architecture, not the system software. Programs are not bound to specific processors but may be moved to an available processor when more than one processor exists.

High-Level Tasking Support

The Ada model of a task is implemented directly in the architecture. A task is conceptually a program unit that executes in parallel to other program units. Synchronization is provided for in Ada and implemented directly in the architecture. Task calls (rendezvouses) are handled in a similar fashion to procedure calls. Task calls to tasks on other processors are no different than task calls to tasks on the same processor.

Highly Segmented Virtual Memory

The Ada model of high modularity is supported by a highly segmented virtual memory. Memory segments are catagorized as one of seven segment types called *spaces*. These spaces, in turn, are divided into numerous segments. Each segment typically contains a specific type of information (e.g., code, data, etc.) for a specific program module. Cross-segment references are carefully protected.

Chapter 2: Run-time Model

A run-time model is an image of what a program looks like when it is running on the architecture. It shows where the various parts of a program are in memory and other architectural entities. It shows how the program is represented and partitioned.

The run-time model for the Rational architecture starts with a discussion of a principle programming language construct: all variables in the language are *typed*. This means that each variable has associated with it certain parameters like upper and lower bounds, bit length, and structure or representation.

Types and Objects

Talk about the separation of church and state (I mean the separation of types and objects). Show creating types, instantiating objects, elaborating objects, etc.

Show declaration, derivation, completion, etc.

Show bounds-with-type vs. bounds-with-object

Show how some things are referenced on control stack and others are placed on control stack (mumble-by-reference vs. mumble-by-value)

Show different representation classes: discrete, float, access, task, etc. Essentially introduce the following N chapters.

Segmented Virtual Memory

List 6 segments (control stack, type stack, data stack, import space, queue space, code segment). Define what is in each one. Refer to appropriate specs for bit patterns of segment contents.

Do we talk about the other 2 segment types???

Exceptions

Show how they are generated at Ada source level, how architecture handles them.

What exceptions can be raised from within instructions and what do they mean.

Utility subprograms

Show common use: enumeration types

Privacy and Visibility

Show difference at both architecture level and at Ada source level.

Stack Management

Show how stacks are used implicitly for most instructions.

Show how calling another task changes stacks and packages don't.

Control stack

Show uses.

Show (or refer to) specs of contents.

Type stack

Show uses.

Show (or refer to) specs of contents.

Data stack

Show uses.

Show (or refer to) specs of contents.

Displays

Explain use of display.

Show typical contents under subprogram calls, etc.

Addressing

Explain Various types of addresses generated in the machine.

Lex-level Delta

Show how a typical instruction addresses an object.

Show how imports are mapped into lex level 0.

Full Logical addresses

Show what a full logical address looks like.

Chapter 3: Discrete Types

Representation

Describe how discrete types and objects are represented on the Type, Data, and Control stacks. Show bit patterns or reference specs where bit patterns are found.

Declaring Types

List operations for declaring types. Show analogous Ada constructs where feasible. Point out any anomolies.

Declaring Objects

List operations for declaring objects. Show analogous Ada constructs where feasible. Point out any anomolies.

Operations

List operations for manipulating objects. Show analogous Ada constructs where feasible. Point out any anomolies.

Chapter 4: Floating Point Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 5: Array Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 6: Record Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 7: Variant Record Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 8: Access Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 9: Package Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 10: Task Types

Representation

Declaring Types

Declaring Objects

Operations

Includes: rendezvous, selects, entries, families, queue segments, etc.

Chapter 11: Segmented Heap Types

Representation

Declaring Types

Declaring Objects

Operations

DO we know what these are yet??????

Chapter 12: Generic Types

Representation

Declaring Types

Declaring Objects

Operations

Explain how generics are not macro-expanded. Why we need special instructions.

Chapter 13: Subprogram Types

Representation

Declaring Types

Declaring Objects

Operations

Includes: all program flow stuff, calls, returns, etc.

Chapter 14: Exception Types

Representation

Declaring Types

Declaring Objects

Operations

Chapter 15: Instruction Dictionary

Arranged by representation class.

See index for alphabetical listing.

Roughly 450 instructions in this chapter. At an average of 2 per page, that's a lot of pages in this chapter.

Draft Status

This draft outlines the entire book but only fills out only some of the chapters. The book is being developed even as you read this, though there are higher priority things I need to work on. As you read this book, think about whether it covers all that you think a new person here should know to a reasonable level of detail.

This draft is being used to test TEX macros as well. Please excuse TEX's (my) mistakes.

Please notice the words in the upper right hand corner of most pages like the one on this page. They are the words that appear in the index. If there is a word on any page that has been overlooked (or one that should have been) please mark such corrections on the list in the corner.

Please return any comments to RJB.

This draft was created on 1/25/1984.