INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Architectural support for, and parallel execution of, programs constructed from reusable software components

Welch, Lonnie Roy, Ph.D.
The Ohio State University, 1990

Copyright ©1990 by Welch, Lonnie Roy. All rights reserved.
NOTE TO USERS

THE ORIGINAL DOCUMENT RECEIVED BY U.M.I. CONTAINED PAGES WITH BLACK MARKS. PAGES WERE FILMED AS RECEIVED.

THIS REPRODUCTION IS THE BEST AVAILABLE COPY.
ARCHITECTURAL SUPPORT FOR, AND PARALLEL EXECUTION OF, PROGRAMS CONSTRUCTED FROM REUSABLE SOFTWARE COMPONENTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Lonnie R. Welch, B.S, M.S.

* * * *

The Ohio State University

1990

Dissertation Committee:

Prof. Bruce W. Weide
Prof. P. Sadayappan
Prof. D. Jayasimha

Approved by

Bruce W. Weide
Adviser
Department of Computer and Information Science
To my wife, Mary.
ACKNOWLEDGMENTS

I would like to thank my advisor, Bruce Weide, for the large amount of time he has spent reading drafts and meeting with me to discuss this research. The members of my reading committee—P. Sadayappan and D. Jayasimha—have offered many helpful suggestions for the dissertation and have also given generously of their time. I would like to thank the members of the Software Reusability group at Ohio State for helping me to refine my ideas.

To my wife, Mary, thanks for the encouragement and support. Your help has been invaluable. To my son, Joshua, thanks for giving me added incentive to complete the degree. To my parents, thanks for your many sacrifices, and for the good examples that you set.
VITA

January 9, 1962 ......................... Born - Akron, Ohio

1985 ........................................ B.S. Computer and Information Science
Ohio State University
Columbus, Ohio

1987 ........................................ M.S. Computer and Information Science
Ohio State University
Columbus, Ohio

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Software Engineering: Prof. B. W. Weide
- Parallel Computing: Prof. P. Sadayappan
- Computer Graphics: Prof. R. Parent
# Table of Contents

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>I  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Problem</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Features of RESOLVE</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 RESOLVE Design Guidelines</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 Objectives of the Implementation of RESOLVE</td>
<td>5</td>
</tr>
<tr>
<td>1.1.4 Potential Inefficiencies of Reusable Software</td>
<td>7</td>
</tr>
<tr>
<td>1.1.5 Assumptions and Limitations</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Research Approach</td>
<td>10</td>
</tr>
<tr>
<td>1.2.1 The Run-Time System</td>
<td>10</td>
</tr>
<tr>
<td>1.2.2 Architectural Support</td>
<td>11</td>
</tr>
<tr>
<td>1.2.3 Parallel Execution</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Prior Research on Topic</td>
<td>19</td>
</tr>
<tr>
<td>1.3.1 Hardware Support for High-Level Languages</td>
<td>19</td>
</tr>
<tr>
<td>1.3.2 RISC Processors</td>
<td>21</td>
</tr>
<tr>
<td>1.3.3 RISC-Based Parallel Processors</td>
<td>21</td>
</tr>
<tr>
<td>1.3.4 Models of Parallel Computation</td>
<td>23</td>
</tr>
<tr>
<td>1.3.5 Automatic Detection of Parallelism</td>
<td>24</td>
</tr>
<tr>
<td>1.4 Organization of the Dissertation</td>
<td>25</td>
</tr>
</tbody>
</table>
# THE RESOLVE VIRTUAL MACHINE

2.1 Major Components of the Architecture ........................................ 28
2.2 Instruction Set ....................................................................................... 36
  2.2.1 Data Movement Instructions ........................................ 36
  2.2.2 Arithmetic and Logic Instructions ........................................ 46
  2.2.3 Control Flow Instructions ................................................... 48
  2.2.4 Miscellaneous Instructions ................................................... 58
2.3 The Formal Specification .................................................................... 60

# RESOLVE PROGRAMS ON THE VIRTUAL MACHINE

3.1 The RESOLVE Run-Time System .................................................... 63
3.2 Code Generation .................................................................................... 71
  3.2.1 The Swap Statement ............................................................ 71
  3.2.2 The Procedure Operation ................................................... 75
  3.2.3 The Function Operation and The Assignment Statement .......... 84
  3.2.4 The Control Operation ......................................................... 87
  3.2.5 The If Statement ................................................................. 90
  3.2.6 The While Statement ......................................................... 93
  3.2.7 The Type Operations: Initialization and Finalization ............. 94
  3.2.8 The Module Initialization Operation ................................ 98
3.3 Primitive Modules................................................................................. 112

# PARALLEL EXECUTION OF RESOLVE PROGRAMS

4.1 Model of Parallel Execution ............................................................ 122
  4.1.1 The Model ............................................................................ 123
  4.1.2 Features of RESOLVE Exploited by the Model ..................... 128
  4.1.3 Evaluation ............................................................................ 130
4.2 Software Support for Parallel Execution ........................................... 133
  4.2.1 DAG Representation of Programs and Construction of the DAG . 134
  4.2.2 Deadlock-Free Mapping of Components onto Physical Processors . 139
  4.2.3 Run-Time System Issues ...................................................... 146
4.3 Parallel Implementation of the Virtual Machine ............................... 147
  4.3.1 Dynamic Extraction of Parallelism From Object Code ................... 147
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>37</td>
</tr>
<tr>
<td>7</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>43</td>
</tr>
<tr>
<td>12</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>47</td>
</tr>
<tr>
<td>14</td>
<td>49</td>
</tr>
<tr>
<td>15</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>52</td>
</tr>
</tbody>
</table>
17 Execution of the RETURN Instruction ...................................................... 53
18 Execution of the CALL_INIT Instruction .................................................. 55
19 Effects of the CALL_INIT Instruction on the Local Data Stack and Parameter Address Stack. ................................................................. 56
20 Execution of the RETURN_FIN Instruction. ............................................. 57
21 Execution of the NEW.FR Instruction ..................................................... 59
22 A Facility Data Record. ................................................................. 65
23 Realization of Stack By Array and Top Pointer. ..................................... 67
24 A Facility Data Record For a Stack Facility. ......................................... 68
25 The RFR for a Stack Facility. ................................................................. 70
26 The LDS During Initialization of Local Variables. ................................... 82
27 The Activation Record of a Module Initialization Operation ................. 102
28 The Construction of an FDR. ................................................................. 104
29 LDS for the Initialization Operation of the Stack Module ...................... 109
30 Execution of the PUSHI Instruction. ...................................................... 114
31 Execution of the PUSHLO Instruction. ................................................... 115
32 Execution of the DTS Instruction. ......................................................... 118
33 Execution of the ACCESS Instruction. ................................................... 120
34 The DAG for a Sample Program. ......................................................... 136
35 One Possible Logical Mapping for a Sample Program. ......................... 138
36 The Logical Call Graph for a Sample Program. ..................................... 140
37 A Physical Mapping for Client. ............................................................ 142
The Physical Call Graph for a Mapping of Client. ........................ 144
Deadlock-Free Physical Mapping of Client. .............................. 145
Data Distribution. ...................................................... 149
Contents of PEs. ......................................................... 150
LDS of PE1. ............................................................... 152
The Call to pop. ........................................................ 153
LDS of PE1 and LDS of PE2. ........................................... 154
RFRs and FCTs. ........................................................... 158
A Facility Clone Table (FCT). ......................................... 159
A Facility Clone Table for an Array Instance. .......................... 160
Case 1 of the CALL Instruction. ....................................... 162
Case 2 of the CALL Instruction. ....................................... 164
The Contents of a Call Packet. ........................................ 170
Construction of a Call Packet. ......................................... 171
Use of a Call Packet. .................................................... 173
Contents of a Return Packet. .......................................... 174
Construction of a Return Packet. ..................................... 175
Use of a Return Packet. ................................................ 177
CHAPTER I
INTRODUCTION

Reusable software is attractive when increased programmer productivity and decreased maintenance costs are the concerns. However, it may appear unattractive when compilation and execution efficiency are considered; techniques for addressing these issues are presented in this dissertation.

This chapter provides an introduction to the work. Section 1 presents the problems addressed in this dissertation by identifying potential inefficiencies of reusable software. The class of reusable software supported the techniques to be described is also defined. Section 2 outlines the approach taken in addressing these inefficiencies and Section 3 provides a review of related research. An overview of the remainder of the dissertation is provided in Section 4.

1.1 The Problem

General-purpose reusable software components have a well deserved reputation for being inefficient [BHI87]. The thesis of this work is that this problem is not inherent in generic software, i.e., reusable software can execute efficiently. Furthermore, it is claimed that architectural support and parallel processing can be used to achieve this goal.
For reusable software to become commonplace, many problems must be solved. The specific problems addressed by the dissertation are discussed below and the type of reusable software considered is defined. An introduction to RESOLVE, a language for writing verifiable and efficient reusable components, and a set of design guidelines for RESOLVE module designers, are presented. A list of objectives of implementations of the language is given, followed by an overview of the potential inefficiencies of reusable software. The section concludes with a statement of the assumptions and limitations of this research.

1.1.1 Features of RESOLVE

This work examines techniques for achieving efficient execution of reusable software components written in RESOLVE (REusable SOftware Language with Verifiability and Efficiency), a language currently under development at The Ohio State University [OWZ90]. RESOLVE programs are built from (reusable) program components, or modules, where a module has the following major characteristics that are relevant to this discussion:

- It typically provides a type and some operations to manipulate variables of that type; no types are built-in.

- It may be generic, i.e., parameterized by types, by operations, and even by other modules.

- It has a formal specification separate from its implementation. A formal specification of a module is called a concept or a template.
• It may have multiple implementations for the same concept. An implementation of a module is called a *realization*.

• It must be *instantiated* to be used, i.e., its parameters must be fixed; an instance of a module is called a *facility*.

• Its functionality can be extended by adding operations; the resulting module is called an *enhancement*.

Operations provided by RESOLVE modules are subtly different than the procedures and functions in other languages [HW89]. Below is a list of some features of RESOLVE operations that are relevant to the design of an efficient implementation and supporting virtual machine:

• Conceptually, parameters are passed “by swapping”: at operation invocation, the values of the formal parameters are swapped with the values of actual parameters; on operation return, they are swapped again. Any implementation of parameter passing that achieves this abstract effect is, of course, acceptable.

• The arguments to a call must be unique, i.e., the same variable may not appear twice in a particular argument list.

• Local variables are automatically initialized upon entry to a block where declared, and are automatically finalized upon exit from that block.

Other important features of RESOLVE for this work are:
• Assignment (copying) of one variable's value to another variable is not a part of
the language; instead, swapping the values of two variables is the only built-in
data movement primitive.

• There are no global variables. Instead, operations can access three kinds of
data:

1. operation parameters;

2. local variables; and

3. module variables (static variables associated with a particular module in-
stance that are shared among operations exported by that instance).

• Aliased variables cannot arise, i.e., the data structure representing a variable's
value can only be known by one name at any time.

• The types of variables is determined statically.

• Modules cannot be instantiated dynamically; i.e., instantiations are declarations
that occur outside the code of module operations, and all instantiations are
performed when a program begins execution.

• There are no constructs in the language for expressing parallelism.

1.1.2 RESOLVE Design Guidelines

Associated with RESOLVE is a set of guidelines [Wei86, Wei89] to be used by those
who design modules. This work assumes that RESOLVE reusable components are
designed following those guidelines, and that efficient implementation of such modules
and of clients using them is an important objective. Brief summaries of the design
guidelines are listed below:

- Try to encapsulate an ADT in each module.

- Do not pack as many concepts as possible—even apparently related ones—into
  one module; in particular, factor the structure of a conglomerate type from its
  contents by using type parameters.

- Whenever possible, define operations in such a way that they can be imple­
  mented to execute in constant time or in logarithmic time.

- Package only "primary" operations with the type; provide "secondary" opera­
  tions in enhancements.

- Be careful that the operations chosen to be provided with a new type span all
  interesting manipulations, but are "orthogonal".

- Design operations that insert a base type variable into a large structure so that
  they swap the value in, instead of copying it in.

1.1.3 Objectives of the Implementation of RESOLVE

Further differences between RESOLVE and other programming languages are found
in the useful objectives placed on the compiler, the linker, the run-time system, and
the architecture. These objectives (which arise from various reusability concerns)
include:
1. Modules must be separately compilable.

2. Object code of a client module should not require any changes if a module it uses is reimplemented.

3. Object code should be shareable among all instances of a (generic) realization.

4. Source code of modules may not be assumed to be available after compilation.

Some consequences of this constraint are:

- Usage-specific optimization of components, if performed, must be performed on object code.

- Parallelism, if used, must be extracted from object code; this may be done statically or at run-time.

Modules must be separately compilable to permit them to be developed individually. Otherwise, an entire application would need to be developed before compilation. Thus, code should be generated for a module (even a generic one) before the actual parameters to the module are known.

The object code of a client module should not require any changes if a module that it uses is reimplemented. This is not the case in some implementations of Ada, for example, resulting in time-consuming recompilation for even minute modifications localized in one module.

Object code should be shareable among all instances of a realization, because source code may not be available and therefore inlining or other instance-specific
code generation techniques may not be possible. This sharing should occur even for
instances of generic modules created with different arguments for generic parameters.

The source code of modules may not be available after compilation in a reusable
software components industry in which vendors make available only formal specifi-
cations and object code in order to protect trade secrets. With this restriction, the
only way to optimize a module’s code is to use formal specifications and object code.

Because there are no constructs in RESOLVE for specifying parallel execution,
parallelism must be extracted automatically. The compiler can only take limited steps
to parallelize modules (e.g., reordering statements to increase potential parallelism
within a single module), since it cannot know how or where they will be used (because
modules are compiled separately). Thus extraction of parallelism from programs will
largely be performed after compilation, using object code and formal specifications if
done statically, or using object code if done dynamically.

1.1.4 Potential Inefficiencies of Reusable Software

When software components are designed in a way that permits their reuse in as many
situations as possible, inefficiencies can arise at execution time. For example, client
programs tend to have many procedure and function calls. Every manipulation of a
variable whose type is provided by a reusable component is achieved by a call to a
module operation. Since RESOLVE has no built-in types, almost every statement is
a call. Each operation is usually small, too, because of extensive layering in module
composition.

In addition to frequent calls to small operations, programs built from reusable
components have another feature that can make them inefficient: they are not designed with a particular architecture in mind. If information hiding [Par72] is practiced, the author of a software component knows nothing about the context in which it will be used. The author does not know whether the component will be used on a parallel machine or on a sequential machine, and if used on a parallel machine what the granularity of parallelism will be. Programming statements that refer to architecture-specific features should therefore be avoided within reusable components [MW88, JKC89]. This results in simplified design, improved portability, and increased reuse, but means that parallelism must be detected automatically.

Another potential inefficiency of reusable software is that the source code of components may not be available. In a mature reusable components industry, it is likely that vendors will only provide formal specifications and object code of modules [OWZ89]. The absence of source code also suggests that parallelism should be extracted from object code or at run-time.

It would appear, then, that the large number of procedure calls would slow the execution of programs. The performance would appear to be further degraded by the inability to write software which is designed with a certain architecture in mind and by the lack of source code on which to perform optimizations. However, this need not be the case. Architectural support and a simple model of parallel execution can combat these potential problems.
1.1.5 Assumptions and Limitations

In this section the assumptions made in this research (and therefore not otherwise justified in the dissertation) are stated. Topics that are beyond the scope of the dissertation are also listed.

Certain properties of RESOLVE are given, as are several constraints on its implementation. No attempt is made to justify these features in the dissertation (for such a discussion see [HW89, Heg89]); efficient implementation of the language is the concern. The RESOLVE language is evolving, influenced by this work and by that of others, and this dissertation is only concerned with efficient implementation of “version 0” of the language, which was frozen in early 1990.

Due to the immensity of the task, some items discussed in the dissertation are only designed, not implemented. An implementation of the virtual machine is not produced in silicon; a chip is not laid out. However, the architectural design is completed at a high level and a simulator of it is used to execute RESOLVE programs. A compiler, a linker and hardware simulator are implemented for RESOLVE and for a uniprocessor implementation of the virtual machine, but not for a parallel implementation of the virtual machine. The feasibility of ideas related to parallel execution are shown by describing their implementations in terms of the existing compiler and the uniprocessor implementation of the virtual machine, not by measurements of actual parallel execution.

The mapping of components onto processors is briefly considered. The goals of the mapping are to illustrate the potential for parallel execution of RESOLVE programs
on the proposed architecture, to address the issue of deadlock, and to provide a framework in which more sophisticated mapping algorithms can be developed. While many possible mapping algorithms exist, the one to be designed here does not attempt to balance the load among processors, nor does it attempt to minimize communication delays between processors; these are areas for future research.

1.2 Research Approach

The efficient implementation of a language such as RESOLVE, with its combination of novel features, module design guidelines, severe implementation constraints, and intended position as a formulation for software reuse has not been attempted before. Due to the differences between RESOLVE and previous programming languages, the issues of hardware and software support for RESOLVE must be carefully explored. The following sections present issues associated with such an implementation and show the steps taken in addressing them. The first section discusses the run-time system, the second section deals with architectural support, and the final section considers parallel execution.

1.2.1 The Run-Time System

The run-time system, compiler and supporting architecture have been designed together, and a complete run-time representation of RESOLVE programs has been developed and implemented. Some problems that have influenced the design of the run-time system are:

- system functions:
- system startup
- storage allocation
- module initialization

• data management:
  - representation of variables
  - variable initialization and finalization
  - efficient implementation of swapping
  - access to facility data (module state)

• support for operations:
  - context of operations
  - efficient implementation of procedure calls
  - sharing of object code among all instances of a realization

1.2.2 Architectural Support

Wirth [Wir87] says, "Quite evidently, the language and the computer's architecture should be designed in harmony to make the translation from program text to code possible, effective, and reliable. The corollary is to regard computer, language, and compiler as a whole, and to design the three components as a harmonious unit." This approach has been taken in the design of a compiler and a virtual machine to support RESOLVE.
The virtual machine is called ARC (Architecture for Reusable Components). A formal specification of the virtual machine is written in RESOLVE itself. Many implementations of the specification are possible, including sequential and parallel implementations. Code is generated without knowledge of how the virtual machine is implemented. Furthermore, the specification is potentially useful in verification of the RESOLVE compiler.

The design of ARC is guided by certain goals related to utility and efficiency. The design goals are discussed in this section. The first goal is that it should be possible to implement the virtual machine efficiently. To achieve this, RISC principles [Pat85] are applied in the design of the virtual machine, and should be applied in its implementations. RISC concepts have been used to design processors to effectively support languages such as Pascal, C, Smalltalk [UBF+84], and Lisp [P+86, SH88]. RESOLVE is a new programming language and has many features not found in these other languages. Thus, it is interesting to apply RISC ideas in the design of an architecture to support RESOLVE.

RISC machines typically pipeline the execution of instructions. ARC is designed with the assumption that good implementations will have a two-stage instruction pipeline. The first stage fetches an instruction and the second stage executes it. In existing RISC machines it is usually necessary to determine the interactions of instructions in the pipeline and to prevent interference between them. For example, an instruction that writes a value to a register could interfere with the instruction following it if the latter reads the value of the same register. MIPS [PGH+84] relies on the
compiler to insert NO-OP instructions where necessary to avoid interference between instructions in the pipeline. RISC I uses hardware mechanisms to detect interference among instructions and to automatically forward the result of one instruction to another instruction that requires it. ARC does not require such mechanisms, since instructions in the pipeline cannot interfere with each other by design.

To facilitate the efficient pipelining of instructions, a set of requirements has been developed. First, delayed branch instructions are provided in ARC. A large percentage of the time a useful instruction (rather than a NO-OP instruction) can be placed after a branch instruction, so delayed branches actually provide opportunities for a RESOLVE compiler to exploit instruction pipelining.

A second requirement related to pipelining is that instructions should be of uniform length, so that fetch and decode of every instruction takes the same amount of time. Pipelining is more effective with this restriction. For the same reason, instructions should execute in a single cycle. Decoding of instructions is also simplified as a result of the uniform length of instructions.

Another requirement related to pipelining is that instructions should be simple (primitive) to avoid an unnecessarily long execution cycle. Since they must execute in a single cycle, it is desirable that all instructions have a short execution time. The execution cycle of the machine generally will equal the execution time of the most complex instruction, i.e., the one having the longest execution time. It is possible to avoid much of this penalty with a "virtual clock", however.

In addition to supporting efficient instruction pipelining, there are other goals
for the design of ARC. The instruction set of ARC should be useful to compiled RESOLVE programs. This is based on another idea borrowed from RISC computers: only provide instructions that are frequently used by compiled programs. An instruction set useful to compiled RESOLVE programs is developed here.

Another aim of the design is that an implementation of the processor should be able to fit on a single chip and should be inexpensive to produce. The efficient execution of programs on ARC requires that all the components of the processor reside on-chip. Otherwise, the execution cycle would be lengthened and throughput would decrease. To encourage large scale replication of the ARC processor as a node in a parallel machine, the chip should be inexpensive. We envision a parallel system as containing perhaps tens of thousands of processing elements (PEs).

Another design goal is that ARC should permit code to be shared among all instances of a realization in an efficient manner. This requirement is made to support a reusable software components industry in which vendors sell components in the form of formal specifications and object code (no source code). Such a practice would require code to be generated once for each realization, and to be used by all instances of the realization. Hardware support is used to make this technique efficient.

The instruction set is designed so that efficient code can be generated for all of RESOLVE. Specifically, code generation for assignment statements, swap statements, operation calls, return statements, if-then-else statements, and while statements is considered. Code generation for module initialization, and for variable initialization and finalization, is also considered.
Architectural support for primitive modules is also examined. These modules occur at the lowest level of abstraction, and must be directly implemented in the machine language of the virtual machine. Therefore, the virtual machine must provide instructions that permit their efficient implementation. The primitive modules for which hardware support is considered are those providing the types boolean, character, integer, pointer, array, record, and character string. Hardware support for a direct implementation of a module that performs storage allocation, which is shared among a number of other primitive modules, is also considered.

In addition to providing a set of machine instructions useful to application programs, the virtual machine efficiently supports the run-time system of RESOLVE. Special-purpose instructions, registers, and memory units accommodate features of the run-time system. For example, the technique that permits code sharing among instances of a module is supported by the NEW.FR and RESTORE.FR instructions, by a "facility register", and by "facility memory".

Efficient implementation of the virtual machine is considered. It is designed in a way that allows an implementation to pipeline instruction execution. The virtual machine provides delayed branch instructions to avoid pipeline penalties due to instruction prefetching. Single cycle execution of instructions is assumed to be the approach an implementation of the virtual machine would employ to facilitate efficient pipelining. To avoid an unnecessarily long execution cycle, the virtual machine provides simple (primitive) instructions.
1.2.3 Parallel Execution

After discussing the design of the virtual machine, the use of parallel computation is explored. The items considered include:

- a model of parallel execution,
- a parallel implementation of the virtual machine,
- a distributed run-time system, and
- a deadlock-free software-to-hardware mapping algorithm

A model of parallel execution suited to the kind of reusable software components written in RESOLVE is developed. To hide the latency of a remote call, an operation is permitted to continue execution until it attempts to access a "locked" variable, i.e., one that was passed to a call whose return is pending. This model of execution is called asynchronous remote procedure call, or ARPC. A variable is automatically locked when it is passed as a parameter to a call. An operation attempting to access a locked variable waits for a remote call to return before it retries the access to the locked variable. The parallel execution model exploits such properties of the RESOLVE language as:

1. Aliased variables are not possible, i.e., the data structure representing a variable's value can only be known by one name at any time.

2. Parameters are passed (in the abstract) "by swapping": at operation invocation, the values of the formal parameters are considered to be swapped with the values
of the actual parameters; on operation return, they are swapped again.

3. The arguments to a call must be unique, i.e., the same variable may not appear twice in a particular argument list.

4. Operations can have no global variables. Instead, they can have three kinds of data:

   (a) operation parameters;

   (b) local variables; and

   (c) facility variables—static variables associated with a particular module instance. Facility variables are shared among operations exported by that instance. A facility variable cannot be passed as an argument to any operation of the facility in which it is facility variable.

5. There are frequent calls to operations.

The parallel model of execution is supported by a parallel (distributed memory) implementation of the virtual machine. The two major issues that potential parallelism raises in the design of the virtual machine are dynamic extraction of parallelism from object code, and communication between the processing elements (PEs) of the parallel computer.

In the implementation discussed here, dynamic extraction of parallelism is achieved by associating a “busy” bit with each word in data memory. The data movement instructions are implemented so that the busy bit of an item is turned on when its value
is pushed onto the parameter stack (there are no data registers on the virtual machine, so items can only be moved between data memory and the parameter stack); correspondingly, the busy bit of an item is turned off when a value is popped from the parameter stack into that item's memory location. Each data access instruction detects accesses to "busy" data words. There is also a busy bit associated with each PE's one-bit flag register, and conditional control flow instructions check the busy bit of that register before branching, to prevent race conditions.

Communication between the PEs is handled by a special component of each PE called the remote call processor (RCP). The RCP performs such tasks as detecting remote calls, constructing call packets, sending and receiving remote calls, and automatically removing parameters from the stack after calls return. Other issues addressed in designing the RCP include the contents of call packets, startup and completion of remote calls, protocol for interprocessor communication, and implementation of data movement instructions in a way that permits the RCP to automatically remove parameters from the stack after calls return.

The run-time system is also modified to accommodate parallel execution. The storage allocation module is distributed to every PE. In addition, special tables are needed to determine addresses of remote procedures at run-time.

The issue of mapping components onto processors is considered briefly. A directed acyclic graph (DAG) is constructed to represent important relationships between the components of a software system, and to model the potential parallelism of such a system. The DAG is used to map software components onto physical processors. A
deadlock free mapping technique is described.

1.3 Prior Research on Topic

This work is related to similar projects in the following areas:

- hardware support for high-level languages,
- RISC processors,
- RISC-based parallel processors,
- models of parallel computation, and
- automatic detection of parallelism.

1.3.1 Hardware Support for High-Level Languages

The dissertation describes research that involves the design of a compiler in conjunction with an architecture. Wirth [Wir86] points out two advantages to this approach: the compiler can be simplified, and efficient code can be generated. These claims are based on his experience with the Lilith computer, designed to support Modula-2. Earlier work on the efficacy of instruction sets can be seen in [Lun77, Tan78], where FORTRAN, ALGOL, BASIC, BLISS, and SAL programs are studied to determine the features that machines should provide.

Wulf [Wul81] discusses the relationships between compilers and computer architecture. He advocates instructions that provide primitives, not solutions. This is the same idea embraced by RISC hardware architects (and RESOLVE software module
designers). He comments that most machines do not support efficient access to data structures—a problem we must address in the support for RESOLVE.

The NS32000 architecture supports modular programming by providing special registers to manage tables containing information about a program's modules, and by offering instructions that use the information in those tables. A similar method works to support code sharing among instances of a RESOLVE module.

The STARLET computer [GG82] is engineered to efficiently execute programs that use ADT modules. While the goal of the STARLET system is similar to ours, its architecture is different. It employs a shared memory processor that achieves parallelism through the pipelining of instructions. A basic data type is provided by the architecture and all user-defined data types are mapped onto it. This enables the architecture to be tailored to pipeline the operations of the basic data type. Our architecture differs in several fundamental ways. First, it does not use shared memory. Second, it employs two types of parallelism: pipelining within each PE, and manipulation of more than one data structure at a time (where each of the data structures being manipulated is stored on a different PE).

Computer architectures designed to support high-level languages (HLLs) are surveyed in [SMMG86]. The authors define a taxonomy for HLL architectures, and classify several existing architectures. Our sequential implementation of the virtual machine falls under the category of an "indirect-execution reduced architecture", meaning that programs must be compiled before execution and that the processor is a RISC.
1.3.2 RISC Processors

It has been noticed by Patterson [Pat85] that many complex instructions (variable length, multiple cycle instructions) are infrequently used. Furthermore, the complex instruction set computers (CISCs) make compilers complicated by requiring analysis of a large number of cases during code generation. Variable length instructions also complicate instruction decoding. In contrast, RISC machines provide simple (primitive) instruction sets. RISC computers are designed by examining the way compiled code uses the instruction sets of computers and then providing instructions that will be used frequently.

Some early RISC architectures are the IBM 801 [Rad83], the RISC I [PS82] and the MIPS [PGH+84]. The RISC I was later extended to support Smalltalk [UBF+84] and Lisp [P+86], and the MIPS was extended to support Lisp [SH88]. Similar methods are followed in the design and implementation of ARC.

1.3.3 RISC-Based Parallel Processors

The usefulness of the RESOLVE virtual machine as a node in a multicomputer is investigated. This is the same approach taken by the RISC I and the MIPS groups, who developed the SPUR [P+86] and MIPS-X [CH87] processors, respectively. SPUR and MIPS-X are examples of shared-memory parallel computers that use RISC processing elements. A third example is the Reduced-Instruction Set Multi-Microcomputer System (RIMMS) [FEH84]. RIMMS divides the address space among the processors; a reference to a memory location on another processor causes execution to automa-
ically transfer to that processor. Similarly, call instructions on our parallel processor are implemented to automatically transfer control to a remote processor when a non-local operation is called. The main difference is that in ARC, nonlocal data are only referenced by address (the contents of nonlocal data are not accessed directly), thus control need not transfer to a remote machine whenever nonlocal data are accessed.

A similar approach is taken in the J-Machine [Dal88, DCC+87], a machine designed to support Concurrent Smalltalk (CST). Whenever a call is made to a method that is not local, or when an operand of a method is not local, a message is sent to the node where the external item is stored. A process is permitted to continue execution while awaiting return of an external message. The “futures” model of continued execution is used: an object whose value is being computed by a remote call is tagged as a future object. The future object can be passed to other methods, thus permitting execution to continue while a remote call executes. The ARC parallel processor allows a similar form of continued execution. However, the J-machine differs in its model of programming. It assumes an object-oriented paradigm, in which messages are sent to objects, whereas ARC supports a more traditional, procedural approach in which data objects (variables) are sent to procedures. The ARPC model also permits the PEs to be much simpler.

An example of a RISC-based distributed memory processor is the Transputer [INM84]. Both the Transputer and the J-Machine have on-chip RAM (as our machine will). Our processor also uses distributed memory and could have an interconnection topology similar to the Transputer, or could be a hypercube, for instance.
1.3.4 Models of Parallel Computation

ARC is designed to support a model of parallel execution developed for RESOLVE programs. In this section, existing models of parallel execution are compared to the one proposed here. In this research, the model of distributing components presented in [MW88] is refined by demonstrating an implementation, and is extended to increase the parallelism permitted.

Previous work includes a model of distributing data structures on a hypercube [SBB87]. The approach requires that the programmer specify how data structures are distributed. Our approach makes no such requirement.

Similar work has been done with Ada. APPL [JKC89] is a language for specifying mappings of programs to architectures. This is done to permit programs to be designed without taking hardware configurations into account—the same philosophy used in the parallelization of RESOLVE. Units of distribution of Ada programs are examined in [VMB+89]. Lundberg [Lun89] focuses on run-time system support for allocation and migration on a multiprocessor. The model of parallel execution in these papers is influenced by the task construct of Ada. In RESOLVE, a different model arises since there is no explicit notion of parallelism in the language's computational model.

Models of parallel execution have been developed for object-oriented languages, too. Dally [Dal88, DCC+87] presents a model and corresponding architecture for CST. Bensley et al. [B+88] develop a model that allows the run-time detection of data-dependent concurrency. Both of these approaches use the "futures" model. As
pointed out above, the object-oriented paradigm differs in that it assumes messages are sent to objects, whereas the RESOLVE approach assumes variables are passed to procedures that can manipulate their values. The ARPC model also permits the PEs to be much simpler.

Other recent work has concentrated on removing serialization constraints from some kinds of data structures [CD90]. This work is interesting, but only has limited applicability in the context of RESOLVE; an approach that works for arbitrary data types is required here.

1.3.5 Automatic Detection of Parallelism

As noticed in [vT89] the automatic parallelization of programs is an important problem to solve for exploiting parallel hardware. The extraction of parallelism from RESOLVE programs is a step in that direction, since it permits the object code of truly reusable software to execute on either sequential or parallel machines. ARC detects and extracts parallelism at run-time. Previous work on detection of parallelism includes:

- Data dependencies are used to schedule instructions on dataflow architectures in [Vee86, Pap88]. Dataflow is limited to a finite window of statements in a program in [HP86, Tom67]. To reduce the cost of PEs, we currently avoid either of these approaches. However, future plans include the exploration of these models for RESOLVE programs.
- Instructions of different streams are interleaved in [KS88]. Our approach is much simpler in order to keep the PEs as simple as possible—only one process per PE is permitted.

- "Futures" are used to parallelize object-oriented programs in [Dal88, DCC+87]. We have no notion of future objects—access to an in-use variable pauses a PE.

- Instructions are dynamically scheduled with hardware that resolves control and data dependencies in [Smi89]. The hardware reorders instructions to achieve parallel execution; we attempt no such reordering, in order to simplify the PEs.

- Compiler techniques for parallelizing FOR loops and array accesses are presented in [PW86, RS90, BFK+90]. While these techniques are useful for specific kinds of programs, automatic parallelization of RESOLVE requires a more general if less sophisticated approach. Parallelization of while loops with loop control variables of arbitrary types must be addressed, and parallelization of accesses to arbitrary data structures is also required. Some researchers have dealt with arbitrary data structures [CD90, Kha90, Gri90], but none has achieved the generality required for RESOLVE. Future work includes the exploration of parallelization of RESOLVE programs through the use of the formal specifications provided with them.

1.4 Organization of the Dissertation

The remainder of the dissertation is divided into four chapters. The design of the RESOLVE virtual machine is presented in Chapter 2 and the use of the virtual
machine by RESOLVE programs is discussed in Chapter 3. Chapter 4 considers the parallel execution of RESOLVE programs and describes a parallel implementation of the virtual machine. Conclusions, contributions, and future research are discussed in Chapter 5.
CHAPTER II
THE RESOLVE VIRTUAL MACHINE

The efficient implementation of a language such as RESOLVE, with its combination of novel features, module design guidelines, and severe implementation constraints, requires a synthesis of previously used techniques. Due to the differences between RESOLVE and existing programming languages, the issues of hardware and software support must be explored anew.

Part of the implementation of RESOLVE involves the design of a compiler and a supporting architecture in conjunction. Wirth [Wir87] says, "Quite evidently, the language and the computer's architecture should be designed in harmony to make the translation from program text to code possible, effective, and reliable. The corollary is to regard computer, language, and compiler as a whole, and to design the three components as a harmonious unit." This approach is taken in the design of a compiler and a virtual machine to support RESOLVE, which was itself influenced heavily by the kinds of implementation considerations addressed here. The virtual machine is called ARC (Architecture for Reusable Components).

This chapter presents the details of ARC. Section 1 describes the components of the architecture; section 2 presents the instruction set; and section 3 discusses the formal specification of the virtual machine.
2.1 Major Components of the Architecture

This section discusses the major components of the ARC processor. As shown in Figure 1, the principle elements of ARC are: registers, memory, an arithmetic and logic unit (ALU), a control unit, and an operation processor. There are no general purpose data registers in ARC. There are very few registers, and each has a specific purpose. Since there are no general purpose registers in ARC, operands to instructions are either on an evaluation stack, are encoded in the instructions themselves, or are in on-chip memory (or cache). The processor contains several types of memory, each for storing a particular kind of information. The ALU performs addition and subtraction of integers and tests a few conditions. The control unit is composed of an instruction register, a program counter and an instruction sequencer. ARC is designed to support the parallel execution of programs constructed from reusable RESOLVE components, with one or more components stored on each PE and with the components interacting via ARPC. The operation processor enables PEs to interact by sending and receiving remote calls.

A block diagram of a possible sequential implementation of ARC is shown in Figure 2. The virtual machine provides several kinds of memory—one for each different kind of item to be stored. This allows design decisions to be factored from unrelated concerns, resulting in ease of design, use, and modification. Instruction memory contains machine instructions—the code of the operations stored on the processor. The program counter (PC) contains the address of the next instruction in instruction memory to be fetched, and the instruction register (IR) holds the next instruction to
Figure 1: The Major Components of ARC.
Figure 2: Possible Organization of ARC.
execute. The instruction sequencer (IS) updates the PC during instruction execution. The IS contains a one-bit flag—the control flag—that can be set to true or false by RETURN instructions and by logical instructions, and that is testable by conditional branch instructions.

The data memory of ARC is divided into two components: the local data stack (LDS) and indexed memory (IM). The LDS is intended to be used for storing activation records for operation invocations (see Figure 3) and for holding the implicit operands of some machine instructions. An activation record for a RESOLVE operation contains entries for each local identifier of the operation. Each entry is typically the address of a variable's representation in indexed memory (see Figure 4). The exception to this occurs when variables' representations fit into a single data word. An example is an integer variable, the representation of which would consume the same amount of space as a pointer. The LDS has the word "stack" in its name since the activation records are stacked on the LDS and are created and destroyed in a last-in-first-out fashion, and since entries in the LDS are referenced relative to the top. The top-pointer of the LDS is increased (or decreased) when an item is pushed (or popped). The LDS is not a true stack, however, since entries can also be accessed randomly. Values on the LDS can be copied, swapped with values in IM, destructively read, and written.

Indexed memory contains static module data and the representations of local variables. For example, an operation with a local variable of type array has a pointer to the representation of the array stored in its activation record. The representation
Figure 3: Local Data Stack (LDS).
Figure 4: Representation of Variables.
of the array is stored in IM (see Figure 4). IM entries are addressed by specifying a base and an offset (similar to the indexed addressing mode found in many computers, hence the name "indexed memory"). Values can be copied, swapped with values on the LDS, destructively read, and written.

Facility memory (FM) holds tables used by CALL instructions. The tables are called run-time facility records, or RFRs and permit all instances of a realization to share code. An RFR for each module instance used by a program is created before run-time. RFRs contain two kinds of entries: addresses of other RFRs and addresses of operations in instruction memory (see Figure 5). The facility register (FR) is also used by CALL instructions and points to the RFR of the current facility (the one providing the operation that is executing). The facility data register (FDR) points to the location in IM of the current module's static data. Both the FR and the FDR are updated when an operation from a facility other than the current one is called. Their old values are saved for use when the called operation returns.

ARC contains a return stack for saving the context when an operation is invoked. The context consists of the PC, the FR and the FDR. The return stack is contained in the operation processor (OP). The OP automatically saves the PC before operation calls, but the FR and FDR must be explicitly saved by using machine instructions. The OP automatically restores parameters when a call returns (a feature exploited in parallel implementations of ARC). To automatically restore parameters, the OP maintains a parameter address stack (PAS) that contains a record of the addresses of all items pushed onto the LDS, since the LDS is used for passing parameters to
Figure 5: A Run-Time Facility Record (RFR).
operations. When an operation returns, the addresses of its actual parameters are obtained from the PAS and are used to restore the values of the actual parameters. In addition, the OP supports ARPC by constructing and sending call packets to operations on other PEs and by receiving and disassembling call packets from other PEs.

2.2 Instruction Set

This section discusses the basic instruction set of ARC. A few additional instructions will be presented when code generation is considered in chapter 3. The instructions can be grouped into four categories: data movement, arithmetic, control flow, and miscellaneous. A convention used in figures depicting effects of operations is that a dashed arrow indicates a value that existed prior to execution of an instruction and a solid arrow indicates a value computed by the instruction. Solid arrows are also used to represent values unchanged by instructions.

2.2.1 Data Movement Instructions

The data movement instructions can be divided into the following groups: PUSH, POP, ACCESS, and I/O.

PUSH Instructions

A PUSH instruction moves a data word from somewhere on the LDS or in IM to the top of the LDS. The top-pointer of the LDS is incremented before moving the item. A push instruction does not push a copy of a value. Instead, it destructively reads
Figure 6: Execution of the PUSH Instruction.
Figure 7: Execution of the PUSHL Instruction.
Figure 8: Execution of the PUSHFD Instruction.
a value: the value of a memory location is read, is placed on top of the LDS, and an unspecified value is left in that memory location. The address of a pushed value is remembered on the PAS so the OP can later restore a value there. The PUSH instructions of ARC are presented below:

- **PUSH n** — Push immediate. (See Figure 6.) Increment the top-pointer of the LDS and place the value “n” onto the top of the LDS.

- **PUSHL n** — Push LDS item. (See Figure 7.) Increment the top-pointer of the LDS. Move the value in the LDS at offset “n” from the old top onto the top of the LDS. Destroy the value in the LDS at offset “n” from the old top.

- **PUSHFD n** — Push facility data. (See Figure 8.) Increment the top-pointer of LDS. Move the value of IM[FDR + n] onto the top of the LDS. Destroy the value of IM[FDR + n].

In order for a return instruction to restore the parameters of a call, a record of the parameter addresses must be kept. This record is built incrementally: each time an item is pushed onto the LDS, its absolute address on the LDS or in IM is pushed onto the PAS. This is illustrated for the PUSHL and PUSHFD instructions in Figures 9 and 10. Notice that each entry in the PAS contains a tag indicating whether the stored address refers to a location on the LDS or in IM. Instructions that place a value onto the LDS from somewhere other than the LDS or IM cause a PAS entry to be created with the tag field set to NULL (see Figure 11). Such entries are discarded by return instructions.
Figure 9: Effects of the PUSHL Instruction on the Parameter Address Stack.
Figure 10: Effects of the PUSHFD Instruction on the Parameter Address Stack.
Figure 11: Effects of the PUSH Instruction on the Parameter Address Stack.
POP Instructions

A POP instruction removes the item from the top of the LDS and places it on the LDS or in IM. The top-pointer of the LDS is decremented following the move. There are three basic POP instructions:

- **POP n** — Pop "n" items. Subtract "n" from the LDS top-pointer.

- **POPL n** — Pop into LDS. (The reverse of PUSHL—see Figure 7.) Move the value on top of the LDS to offset "n" from the old top. Decrement the LDS top-pointer.

- **POPFD n** — Pop into facility data. (The reverse of PUSHFD—see Figure 8.) Move value on top of the LDS to IM[FDR + n]. Decrement the LDS top-pointer.

ACCESS Instructions

An ACCESS instruction swaps the item on top of the LDS with an item in IM. This instruction allows efficient implementations of many constructs found in RESOLVE programs, and is in line with the philosophy of RESOLVE programming (see [HW89]). The ACCESS_O instruction is presented here. Another version of the ACCESS instruction is presented in the discussion of RESOLVE programs on the virtual machine (see chapter 3).

- **ACCESS_O n** — Swap LDS entry with IM entry, offset of IM item is immediate.

(See Figure 12.) Swap item on top of the LDS with item in IM[a + n]. "a"
Figure 12: Execution of the ACCESS_O Instruction.
is second item on the LDS before the instruction executes. Retain "a" on the LDS.

I/O Instructions

An I/O instruction allows the movement of a data word from the input stream to the top of the LDS, or from the top of the LDS to the output stream. Two instructions are provided for these purposes:

- READ — Read item. Increment the top pointer of the LDS, remove the next value from the input stream and place it on top of the LDS.

- WRITE — Write item. Append a copy of the value on top of the LDS to the output stream.

2.2.2 Arithmetic and Logic Instructions

The arithmetic and logic instructions take their operands from the top of the LDS. They are the following:

- ADD — Add the two values on the top of the LDS. (See Figure 13.) Increment top-pointer of the LDS, perform the addition, and store the result on top of the LDS.

- SUB — Subtract the value on the top of the LDS from the value immediately below it on the LDS. (Similar to ADD—see Figure 13.) Increment top-pointer of the LDS, perform the subtraction, and store the result on top of the LDS.
Figure 13: Execution of the ADD Instruction.
• CLRZ — Clear if zero. Set the control flag to false if the value on top of the LDS equals zero, otherwise set it to true.

• CLRN — Clear if negative. Set the control flag to false if the value on top of the LDS is less than zero, otherwise set it to true.

2.2.3 Control Flow Instructions

The control flow instructions provide the means to branch, to call operations, and to return from operations. They are presented here. In addition to unconditional and conditional branch instructions, different call and return instructions are provided for each variety of RESOLVE operation. Specifically, different instructions are provided for use with each of the following:

1. procedure and function operations
2. control operations
3. type initialize and finalize operations

Branch Instructions

The branch instructions of ARC are:

• BRANCH n — Unconditional branch. (See Figure 14.) Perform a delayed unconditional branch to the current PC plus “n”. (Note: the current PC is not the address of the branch instruction, but one greater than that.)
Figure 14: Execution of the BRANCH Instruction.
• BRFALSE n — Branch if false. If the value of the control flag is false, take a
delayed branch to the current PC plus “n”.

• BRTRUE n — Branch if negative. If the value of the control flag is true, take
a delayed branch to the current PC plus “n”.

Instructions for Use With Procedure and Function Operations

• CALL n — Call a procedure or function operation. (See Figures 15 and 16.)
Take a delayed branch to a procedure or function operation. Increment the
current PC and save the value on the return address stack. Load the address
of the operation into the PC (from FM[FR + n]). Place an entry on the PAS
(see Figure 16) telling the number of parameters to the called operation, and
set the tag of the entry equal to COUNT.

• RETURN — Return from a procedure or function operation. Return from a
procedure or function operation by restoring the PC and the parameters. Fig­
ure 17 shows the restoration of parameters for an operation with two parameters.
The first parameter, “y”, is restored to the LDS and the second parameter, “x”,
is restored to IM. The addresses of the parameters are taken from the PAS.

Instructions for Use With Control Operations

ARC also provides instructions for use with control operations:

• CALL_CTRL n — Call a control operation. (Similar to CALL—see Figures 15
and 16.) Take a delayed branch to a control operation. Increment the current
Figure 15: Execution of the CALL Instruction.
Figure 16: Effects of the CALL Instruction on the Parameter Address Stack.
Figure 17: Execution of the RETURN Instruction.
PC and save the value on the return address stack. Load the address of the operation into the PC (from FM[FR + n]).

- **RETURN_TRUE** — Return true from a control operation. (Similar to RETURN; see Figure 17). Return from a control operation by restoring the PC and the parameters, and by setting the value of the control flag to true.

- **RETURN_FALSE** — Return false from a control operation. (Similar to RETURN; see Figure 17). Return from a control operation by restoring the PC and the parameters, and by setting the value of the control flag to false.

**Instructions for Use With Type Operations**

Instructions are also provided for calling type initialize and finalize operations:

- **CALL_INIT n** — Call an initialize operation. (See Figures 18 and 19.) Take a delayed branch to an initialize operation. Increment the current PC and save the value on the return address stack. Load the address of the operation into the PC (from FM[FR + n]). A parameter is returned, but not passed in; Make this possible by incrementing the LDS top-pointer and placing an entry containing the address of the location on top of the LDS on the parameter address stack. The corresponding RETURN instruction causes the initialized value to be placed on top of the LDS (the location specified by the entry on top of the parameter address stack).
Figure 18: Execution of the CALL_INIT Instruction.
Figure 19: Effects of the CALL_INIT Instruction on the Local Data Stack and Parameter Address Stack.
Figure 20: Execution of the RETURN FIN Instruction.
• **RETURN_INIT** — Return from an initialize operation. Return from an initialize operation by restoring the PC and the parameters. A parameter is returned, but not passed in.

• **CALL_FIN n** — Call a finalize operation. Take a delayed branch to a finalize operation. Increment the current PC and save the value on the return address stack. Load the address of the operation into the PC (from FM[FR + n]). A parameter is passed in, but not returned. The value on top of the LDS is an implicit parameter (it was not explicitly pushed onto the LDS). Type finalize operations are assumed to return nothing; thus, no call record is created on the PAS.

• **RETURN_FIN** — Return from finalize operation. (See Figure 20.) Return from finalize operation by restoring the PC. A parameter is passed in, but not returned, thus the top-pointer of the LDS is decremented. Since there are no return parameters, do not pop any parameters automatically. The advantage of providing this instruction is that it eliminates the needless return of finalized variables (useful for reducing network traffic in a parallel implementation of ARC).

### 2.2.4 Miscellaneous Instructions

A few instructions fall into the miscellaneous category:

• **NO-OP** — Do nothing. Possibly needed because of delayed branching.
Figure 21: Execution of the NEW_FR Instruction.
• NEW\_FR n — Get new facility register. (See Figure 21.) Save the value of the facility register on the return stack and load the facility register with FM[FR + n].

• NEW\_FDR n — Get new facility data register. (Similar to NEW\_FR; see Figure 21.) Save the value of the FDR on the return stack and load the FDR with IM[FDR + n].

• RESTORE\_FR — Get old facility register. (The reverse of NEW\_FR; see Figure 21.) Load the FR with the value on top of the return stack.

• RESTORE\_FDR — Get old facility data register. Load the FDR with the value on top of the return stack.

• WAIT — Suspend the processor. Wait for an external call.

2.3 The Formal Specification

Fortunately, users of ARC need not rely on the informal description given above. A formal specification of ARC has been written using RESOLVE. (For additional information about RESOLVE specifications, see [Mur90, Har90].) One possible use of the specification is in verification of RESOLVE compilers; the specification permits a verifier to reason about the output of a code generator to determine if it is correct. The specification also permits implementors of ARC to provide many implementations (including sequential or parallel implementations); it permits users to change implementations, and still execute all software written for it. RESOLVE compiler
writers can refer to the specification when writing code generators. The specification is contained in Appendix A. Highlights of the specification are discussed in this section.

The specification (or concept) is parameterized by facilities describing the lengths of memory words and memory addresses. This permits a client of the template to select different sizes of memory and addresses independently from other concerns. Proofs of correctness will be valid for all possible values of the parameters.

Several theories from mathematics are used to reason about the structure and behavior of ARC. These include boolean algebra, string theory, tuple theory, and function theory. Most of these theories are parameterized and several different instances of them are used to model the components of ARC.

The processor itself is modeled by a tuple containing a field for each component. Instruction memory is modeled by a function from integers to instructions, where an instruction is a triple with three integer fields: opcode, base, and offset. The instruction register is represented by a triple of integers (opcode, base, and offset). Facility memory and indexed memory are represented by functions from integers to integers; the domains of the functions represent memory addresses and the ranges of the functions represent memory contents. Strings of integers model the return stack and the local data stack; pushing an item onto one of the stacks is described by concatenating an integer onto the string. The facility register, facility data register, program counter, and parameter count (the size of the parameter address stack) are modeled using integers; the control flag is modeled by a boolean. The parameter
address stack is modeled by a string of parameter addresses. A parameter address is modeled by a pair of integers—a tag specifying whether the address field refers to a location on the LDS or to a location in IM, and the address.

The definition of constants follows the declaration of math theories. The constants describe the number of machine instructions provided, the integer values of opcodes, the sizes of memories, and the largest integers permitted in a single data word.

The interface section of the specification begins with descriptions of the types provided by the template: “instruction” and “processor”. The mathematical models of the types are explained and their initial values are given.

Following the descriptions of the types, operations are defined. The concept provides the capabilities of loading and examining various memories. Each operation has an optional requires clause (pre-condition that must be satisfied for the operation to work as expected) and an ensures clause (post-condition that describes the effect of the operation when the requires clause is satisfied). These operations are used by loaders and user-interface programs built on top of a virtual machine simulator. The last operation of the concept is called “do-operation” and contains a description of the effect of each machine instruction provided by ARC.
CHAPTER III

RESOLVE PROGRAMS ON THE VIRTUAL MACHINE

This section explains how ARC is intended to be used by RESOLVE programs. It begins with a presentation of the RESOLVE run-time system and is followed by a discussion of code generation strategies. It concludes by considering hardware support for primitive RESOLVE modules—those that occupy the lowest level of the abstraction hierarchy and are coded in the assembly language of ARC.

3.1 The RESOLVE Run-Time System

The run-time management of ARC’s resources is considered in this section. Issues include: system startup, storage allocation, representation of variables, access to static module data, context of operations, and code sharing among instances of a realization.

Execution of a RESOLVE program begins with the initialize operation of the module at the highest level of the instantiation hierarchy (hereafter referred to as the “main” module). The operation is invoked by the initialize operation of the memory allocator module. The memory allocator manages the IM and provides operations to allocate a block of memory and to return a block of memory to free storage. It
also has a module initialize operation (as do all RESOLVE modules). This operation initializes tables in IM that are used for managing the free storage of IM and then calls the module initialization operation of main.

As discussed above, execution begins with the initialize operation of the memory allocator module and proceeds to the initialize operation of the main module. The initialization operation of main calls the module initialization operations of all facilities it instantiates, each of those calls the initialize operations of the facilities it instantiates, and so on. After initializing locally declared facilities, a module initialization operation allocates storage for the static variables (called facility variables) of the module and assigns them initial values. After that the user-supplied code for the initialization operation is executed. This code can invoke operations from any facilities known to the module.

Another issue considered in the design of the run-time system is the management of contexts for operations. The context of an operation consists of parameters, local variables, an optional function result, and facility data. The first three items are stored on the LDS (see Figure 3) and constitute an operation's activation record. An LDS entry is typically the address of a variable's representation. This representation is usually stored in IM (see Figure 4). Facility data records typically contain entries that contain addresses of the representations of facility variables; these representations are stored in IM. Facility variables are stored in IM. A particular facility variable is accessed by specifying its offset from the start of the FDR.

The FDR points to the location in IM of the facility data record for the current
Figure 22: A Facility Data Record.
facility. The structure of facility data records is shown in Figure 22. The facility data record for a module instance contains entries pointing to the facility data records of all facilities providing operations called from the code of the instance's operations. I.e., for each facility “f” that provides an operation called by facility “g”, there is a pointer to the facility data record of “f” stored in the facility data record of “g”. Facility data records also contain the static variables associated with a particular instance. These entries are usually pointers to the representations of the variables. The representations are stored in IM.

Before every operation call, the FDR must be changed to point to the facility data record of the facility providing the called operation. The FDR is loaded using the NEW_FDR instruction, and is restored after the return of every call with the RESTORE_FDR instruction. NEW_FDR obtains the new value of the FDR by adding the old value of the FDR to the immediate data encoded in the instruction, and retrieving the value in IM at that location. The RESTORE_FDR instruction replaces the FDR with the value on top of the return stack.

To illustrate the notion of facility data records, consider a generic stack module implemented using an array containing items of type “U T” (“T” is the type of the items in the stack) and an integer top-pointer (see Figure 23). The form of a facility data record for an instance of the stack is shown in Figure 24. Since the code for the stack module contains calls to the array and integer facilities and to the facility providing the type of items in the stack, the facility data record for a stack instance contains pointers to the facility data records of each of these facilities. Following these pointers
Figure 23: Realization of Stack By Array and Top Pointer.
Figure 24: A Facility Data Record For a Stack Facility.
is the static facility data associated with this instance of the stack module. A facility variable can be accessed by all operations of the stack facility by specifying its offset from the address contained in the FDR. For an operation to access its facility data, its caller must set the value of the FDR by using the NEW_FDR instruction. To enable a caller to do this, pointers to the facility data record of a module instance are placed in the facility data records of all clients of the instance.

Run-time facility records (RFRs) have a structure that is similar to facility data records (compare Figures 5 and 22). The RFR for a module instance contains entries pointing to the RFRs of all facilities providing operations called from the code of the instance's operations. It also contains the addresses of operations provided by the instance.

The purpose of RFRs is to permit code sharing among instances of a realization. This is achieved through the generation of instructions that make references to entries contained in RFRs. Before calling an operation, the NEW.FR instruction is used to load the FR with the address of the RFR of the facility providing the called operation. The offset field of NEW.FR specifies the position in the current RFR of the address of the new RFR. (The address of the current RFR is contained in the FR.) The CALL instruction contains an offset that indexes into the new RFR. It specifies the address of the code of the called operation. With this scheme, code can be generated once for each realization and used with all instances of it, since references are indexes into RFRs, and since the structure of the RFRs for all instances of a realization is the same.
Figure 25: The RFR for a Stack Facility.
Figure 25 shows the structure of an RFR for an instance of stack implemented with an array and a top-pointer. It contains pointers to the RFRs of the array and the integer facilities and of the facility providing the type of the items in the stack, and pointers to the code of the operations provided by the stack module.

### 3.2 Code Generation

This section considers code generation for RESOLVE programs. The purpose of this discussion is to illustrate how the features of ARC are suited to RESOLVE programs. RESOLVE has the following kinds of statements: swap, call, assignment, return, if, and while. Code generation for each is explained. Code generation is also discussed for operation definitions, for variable initialization and finalization, and for module initialization. For each construct, an example is shown and is followed by the code that is generated for the construct.

#### 3.2.1 The Swap Statement

The swap statement has the following form:

```plaintext
<swap>:
  x := y
```

The code generated for a swap statement depends on the locations of the variables to be swapped. The variables accessible to an operation are those contained in its activation record on the LDS (these are local variables and parameters), and
those stored in its facility data record in IM (these are static facility variables). The former are accessed by specifying offsets from the LDS top-pointer and the latter are referenced by specifying offsets from the FDR. Four different cases can arise: both “x” and “y” may be on the LDS, “x” may be on the LDS and “y” may be in IM, “y” may be on the LDS and “x” may be in IM, or both “x” and “y” may be in IM.

The first case occurs when “x” and “y” are local variables or are parameters to an operation, i.e., they are stored on the LDS. The code generated for the swap statement in such a case is shown below:

```
PUSHL x
PUSHL y + 1
POPL x + 2
POPL y + 1
```

The value of “x” is pushed onto the LDS and then the value of “y” is pushed onto the LDS. The values are then popped from the LDS into their new locations. The arguments to the PUSHL and POPL instructions represent offsets into the activation record for the operation in which “x” and “y” are declared. The values of these offsets are static, since the structure of activation records is static. However, the offsets of “x” and “y” from the top of the LDS change as values are pushed and popped. Constants are added (by the compiler) to the offsets into the activation records to reflect these changes.
The second case arises if “x” is a local variable or a parameter to an operation (i.e., it is stored on the LDS) and “y” is a facility variable (i.e., it is stored in IM). Examine the code for this case:

```
PUSHL x
PUSHFD y
POPL x + 2
POPFD y
```

As in the code for the first case of the swap statement, the values of “x” and “y” are pushed onto the LDS and then popped into their new locations. In this case, however, the value of “y” is referenced by specifying an offset from the FDR. Since the value in the FDR is not affected by pushes and pops, no offset needs to be added to “y”. Thus, the argument to the PUSH_FD and POP_FD instructions is the offset of “y” in a facility data record.

The third case arises if “x” is a facility variable (i.e., it is stored in IM) and “y” is a local variable or a parameter to an operation (i.e., it is stored on the LDS). The code for this case is:

```
PUSHL y
PUSHFD x
POPL y + 2
POPFD x
```
The locations of the values of "x" and "y" are in opposite locations of where they are in case 2. The values of "x" and "y" are pushed onto the LDS and then popped into their new locations. The value of "x" is referenced by specifying an offset from the FDR. Since the value in the FDR is not affected by pushes and pops, no offset needs to be added to "x". Thus, the argument to the PUSH.FD and POP.FD instructions is the offset of "x" in a facility data record.

The fourth case of the swap statement occurs when both "x" and "y" are facility variables, i.e., both "x" and "y" are stored in IM. The code for performing the swap in such a case is shown below:

```
PUSHFD x
PUSHFD y
POPFD x
POPFD y
```

As in the other cases, the values of "x" and "y" are pushed on to the LDS and then popped in reverse order to achieve the swap. Both "x" and "y" are accessed by specifying their offsets from the start of the current module's facility data.

The code generated for the fourth case of the swap statement could be shortened by one instruction if an ACCESS instruction were provided to swap the top item on the LDS with a value in IM. The code would look as follows:

```
PUSHFD x
```
The value of "x" is pushed onto the LDS and then swapped with the value of "y". The argument to ACCESS_FD specifies the offset of "y" in the current module's facility data record.

ACCESS_FD is not included in the instruction set of ARC because it would not be used very often. The above case is the only time it would be used in code generation, and very few module have facility variables if the RESOLVE design guidelines are followed.

### 3.2.2 The Procedure Operation

The next kind of statement for which code generation is explained is the statement for calling operations. RESOLVE operations can be procedures, functions, or controls. Procedures are similar to those in Pascal or Ada. RESOLVE functions are not permitted to change the values of their parameters and invocations can only occur on the right hand side of assignment statements. Control operations are similar to functions in that their parameters cannot be modified, but differ in that they can only return the values "yes" and "no". In addition, the values returned from control functions cannot be assigned to variables. They can only be tested in the conditional parts of if statements and while statements.

The code for all operation begins by initializing local variables, proceeds to the statements defining the behavior of the operation, and concludes with the finalization
of local variables. There are minor differences in the code generated for procedures, functions and controls. The code for calls to each kind of operation is shown as well as the code for definitions of each kind of operation.

The form of a procedure call is shown below:

<proccall>:

fac.opname(x, y)

A procedure name is qualified with the name of the facility, “fac”, that provides it. The procedure is called “opname” and has two parameters, “x” and “y”. If “x” is a local variable or a parameter to the calling operation (i.e., it is stored on the LDS), and “y” is a facility variable (i.e., it is stored in IM), the code that is generated is below:

PUSHL x
PUSHFD y
NEW_FR fac
NEW_FDR fac
CALL opname
NO-OP
RESTORE_FDR
RESTORE_FR
First, the parameters to the call are pushed onto the LDS. Since “x” is stored in an activation record on the LDS, its value is pushed as a parameter to opname using the PUSHL instruction. The value of “y” is stored in IM in the current facility data record, so it is pushed using the PUSHFD instruction. Creation of the activation record for “opname” is begun with these push instructions. It is completed by the code of “opname” when it initializes its local variables.

Following the placement of the parameters on the LDS, new values for the FR and the FDR are obtained. Recall that the argument passed to the NEW_FR instruction specifies the offset in the current RFR where the pointer to the RFR of the facility providing “opname” is stored. That value is loaded into the FR by the NEW_FR instruction, and is used by the CALL instruction. The old value of the FR is stored on the return stack. The argument to the NEW_FDR instruction specifies the offset in the current facility data record where the pointer to the facility record of the facility providing “opname” is stored. That address is loaded into the FDR and the old value of the FR is stored on the return stack.

The same argument is passed to both the NEW_FR and the NEW_FDR instructions, since the preambles of the RFR and the facility data record are isomorphic. A machine instruction, NEW_FACILITY, that exploits this characteristic could be added to the architecture. The NEW_FACILITY instruction would load new values into the FR and the FDR and save the old values. Efficient implementation of the instruction requires the loads to be performed in parallel and the saves to be performed in parallel.
With the RFR and the facility data record set up, the call to "opname" is made. The CALL instruction has an argument specifying the offset in the new RFR (the RFR for "fac") where the address of the code for opname is stored. That address is loaded into the PC. The instruction following it is executed before branching, since the CALL instruction is a delayed branch. A NO-OP instruction is inserted after the CALL instruction in the above code, but the NEW_FDR instruction can be placed there to reduce the number of instructions required. The resulting code is shown below:

```
PUSHL x
PUSHFD y
NEW_FR fac
CALL opname
NEW_FDR fac
RESTORE_FDR
RESTORE_FR
```

When "opname" returns, the values of the FR and the FDR are restored from the return stack using the RESTORE_FDR and RESTORE_FR instructions. In addition, the values of the parameters are automatically restored. "x" is restored to the LDS and "y" is restored to IM.

Having seen how a procedure is called, now consider code generation for procedures. The form of procedure definitions is given below:
<procdecl>:

PROCEDURE proc_name

[ PARAMETERS
  proc_frm1_parm*
END PARAMETERS ]

BEGIN

[ LOCAL VARIABLES
  var_decl*
END LOCAL VARIABLES ]

stmts

END proc_name

The parameter passing mechanism in RESOLVE is call-by-swapping [HW89]. This means that the values of the formal and actual parameters are swapped at the invocation of an operation and upon the return of the operation. The value of the formal before the swap occurs is an initial value of the appropriate type. Although the semantics of the parameter passing mechanism say that swapping of values occurs,
implementations of RESOLVE need not swap, as long as they produce the expected behavior. The implementation presented here places the address of the actual parameter into the memory location allocated for the formal parameter and performs the reverse on return.

A variable declaration (\texttt{var.decl}) has the following form:

\begin{verbatim}
var_name : fac.type_name
\end{verbatim}

Local variables are initialized at the start of a procedure by calling the appropriate type initialization operations. The compiler generates calls to the initialization operations, so it appears to a RESOLVE programmer that initialization is automatically performed. Local variables are finalized at the end of a procedure’s execution. Calls to variable finalize operations are also generated by the compiler.

Following variable initialization, the statements of a procedure are executed. The statements may be any of those discussed in this section (\texttt{swap}, \texttt{call}, \texttt{return}, \texttt{assignment}, \texttt{if}, and \texttt{while}), except for the \texttt{return yes} and \texttt{return no} statements (which can only be used to return from control operations).

Initialization of local variables is the first thing done when a procedure begins execution. That code is a sequence of initializations, one for each local variable. The (optimized) code to initialize a local variable is:

\begin{verbatim}
NEW_FR fac
\end{verbatim}
CALL_INIT type_name.initialize
NEW_FDR fac
RESTORE_FDR
RESTORE_FR

The FR and the FDR are set up to point to the RFR and the facility data record, respectively, of the facility providing the type of the variable to be initialized. The CALL_INIT instruction is used here since a type initialize operation takes no parameters, but returns one parameter¹. Following the call, the values of the FR and the FDR are restored from the return stack.

To illustrate variable initialization, consider an operation that declares three local variables, "x", "y", and "z". The LDS at various times during execution of the operation is shown in Figure 26. The first picture shows the LDS before any of the variables are initialized. The second shows the LDS after the call to, but before the return of, the operation that initializes "x". A place has been created on the LDS for the value of "x", but that value has not yet been computed. The third picture shows the LDS after the call to, but before the return of, the initialize operation for "y". The value for "x" has been computed and the place for "y" on the LDS has been allocated. The fourth picture shows the LDS after the call to initialize "z".

Finalization of local variables is performed following execution of the statements of a procedure. The finalization code for the local variable declaration section is a

¹At this point in the discussion it may not be obvious why the ordinary CALL instruction is not used here, but when parallel implementations of ARC are considered in Chapter 4, the reason will become clear.
Figure 26: The LDS During Initialization of Local Variables.
sequence of calls to the type finalization operations for the types of the local variables. The code to finalize a local variable is shown below:

```
NEW_FR fac
CALL_FIN type_name.finalize
NEW_FDR fac
RESTORE_FDR
RESTORE_FR
```

The finalize operation for the appropriate type is called, after the FR and FDR are set to point to the RFR and facility data record of the facility providing the type. The variable to be finalized is an implicit parameter to the finalize operation—it need not be pushed onto the LDS prior to the call. The parameter is passed to the operation, but is not returned. Thus, following the return of the finalize operation, the top-pointer of the LDS is decremented. The LDS shrinks as variables are finalized and grows as they are initialized, creating and destroying the local variable section of activation records. The CALL_FIN instruction accommodates the unusual interface of variable finalize operations.

An alternate way of finalizing variables is shown below:

```
PUSHL x ;push var. to be finalized
NEW_FR fac
```
CALL type_name.finalize ; call type finalize opr.
NEW_FDR fac
RESTORE_FDR
RESTORE_FR
POP 1

This technique does not require the CALL_FIN and RETURN_FIN machine instructions. The variable to be finalized is pushed onto the parameter stack and the appropriate operation is called. Following the return from the call, the storage for the variable on the LDS is deallocated by performing a "POP 1" instruction. This method for finalizing variables was not chosen for reasons related to parallel execution (see Chapter 4).

3.2.3 The Function Operation and The Assignment Statement

Code generation for function operations is also necessary. In RESOLVE, function results can only be assigned to variables; they cannot be passed as arguments to other operations. Thus, it is appropriate to also discuss code generation for assignment statements at this point.

The form of an assignment statement is as follows:

<assignment>:

\[ x := \text{fac.opname}(y, z) \]
Assume "x", "y", and "z" are on the LDS, i.e., each is a local variable or a parameter to an operation. The code generated for an assignment statement of the above form looks like:

```
PUSHL y
PUSHL z + 1
PUSHL x + 2
NEW_FR fac
CALL fac.opname
NEW_FDR fac
RESTORE_FDR
RESTORE_FR
```

The parameters to the function are pushed onto the LDS. In addition, the old value of the variable to which the function result is assigned must be pushed. There are two reasons this old value is pushed. The first is to permit ARC to automatically pop a function result into the correct location when a function returns. (Recall that ARC automatically restores parameters when operations return.) To achieve this, it must know the addresses of the parameters. The addresses are remembered when items are pushed onto the LDS.

Another reason for pushing the variable to which the function result is assigned is that it must be finalized before the function result is stored into it. The finalization
takes place in the code generated for a function definition. The form of such a
definition appears below:

<provfunc>:

```
FUNCTION func_name RETURNS var_name : fac.type_name

[ PARAMETERS
  func_frm1_parm*
END PARAMETERS ]

BEGIN

[ LOCAL VARIABLES
  var_decl*
END LOCAL VARIABLES ]

stmts

END func_name
```

With functions a variable is supplied for returning a value. The parameters cannot
have their values changed, thus they can only have "preserves" mode.
The variable used to return a function result is first finalized and then initialized. It seems that these operations cancel each other, but that is not the case. Since a function result is assigned to a variable, the old value of the variable must first be finalized. Thus, the old value of the variable is passed as a parameter to the function and is finalized by the function. The initialize operation is necessary because the return variable is treated like a local variable by the code of the function, and all local variables are assumed to have initial values. Following the finalization and initialization of the result variable, the local variables are initialized, the statements execute, and then the local variables are finalized. The result variable is not finalized, but its value is returned to the caller.

3.2.4 The Control Operation

Control operations were incorporated into RESOLVE for the purpose of making branch decisions in if-then-else statements and in while loops. They can be thought of as functions returning either the value "yes" or the value "no". Control operations are not permitted to change the values of their parameters. They can only be called in two locations in programs: as the condition of an if-then-else statement and as the condition in a while statement. The form of a call to a control operation is identical to that of a call to a procedure and is shown below:

```
<ctrlcall> :

fac.opname(x, y)
```
Assume "x" and "y" are on the local data stack, i.e., each is a local variable or a parameter to an operation. The code generated for the above control call is shown below:

```assembly
PUSHL x
PUSHL y + 1
NEW_FR fac
CALL_CTRL opname
NEW_FDR fac
RESTORE_FDR
RESTORE_FR
```

The code generated for a control call differs slightly from that generated for a procedure call. The CALL_CTRL instruction is used here to convey the notion that the called operation will affect the control flag of the processor. The usefulness of this instruction is discussed further in the chapter on parallel execution.

The definition of a control operation has the form shown below:

```assembly
<provctrl> :

CONTROL ctrl_name

[ PARAMETERS
```
func_frm_l_parm*
END PARAMETERS ]

BEGIN

[ LOCAL VARIABLES
var_decl*
END LOCAL VARIABLES ]

stmts

END ctrl_name

For a control definition, code is generated to initialize local variables, code for each of the statements of the operation is produced, and variable finalization code is generated. Control operations cannot change the values of their parameters, thus their formal parameters can only have "preserves" mode. The form of a control operation parameter, a func_frm_l_parm, is:

<proc_frm_l_parm>:

PRESERVES var_name : fac.type_name

Another difference between control operations and procedures is that control op-
erations must contain a "return yes" or a "return no" statement, since it is through the use of these statements that control operations return their results.

3.2.5 The If Statement

The use of control operations in RESOLVE is illustrated by examining if-then-else statements and while statements. There are four different forms of the if-then-else statement. The first form does not have an "else" portion and is shown below:

```<if>:

IF ctrlcall THEN
    stmts
END IF
```

The value of a control operation must be tested before executing or skipping the statements in the if-then statement. Code generation for the if-then is straightforward and is shown below:

```Code(ctrlcall)
BRFALSE next
NO-OP
Code(stmts)
next: ...
```
The control operation is called and it sets the value of the control flag either to true or false. The value of the control flag is tested by the BRFALSE instruction and if it is false the “then” portion is skipped. Otherwise the “then” portion is executed. Optimization can be performed on the generated code by removing the NO-OP following the BRFALSE instruction and replacing it with the last instruction of the code of the call to the control operation (Code(ctrlcall)). The optimized code is shown below, with the code of the call to the control operation expanded inline.

\[
\text{PUSHL } x \hfill \\
\text{PUSHL } y + 1 \hfill \\
\text{NEW_FR fac} \hfill \\
\text{CALL_CTRL opname} \hfill \\
\text{NEW_FDR fac} \hfill \\
\text{RESTORE_FDR} \hfill \\
\text{BRFALSE next} \hfill \\
\text{RESTORE_FR} \hfill \\
\text{Code(stmts)} \hfill \\
\text{next: ...}
\]

Another form of the if-then statement negates the value returned by the control. Its syntax is shown below along with the code generated for it.

\[
\text{IF NOT ctrlcall THEN}
\]
The final two forms of the if-then-else statement contain an "else" portion. One of these negates the value returned by the control operation and the other does not. The latter form is shown below, followed by the code generated for it. The code for the former differs only in the BRANCH instruction used; it uses BRTRUE instead of BRFALSE.

IF ctrlcall THEN
  stmts1
ELSE
  stmts2
END IF
3.2.6 The While Statement

While statements test the value of a control operation before each iteration of a loop. The control operation may have the word "NOT" in front of it (just as it could in if-then-else statements). The first form of the while statement is shown below, and is followed by the code. For the case where the result of the control operation is negated, the difference in the generated code is that BRTRUE is used in place of BRFALSE.

<while> : 

WHILE ctrlcall DO
    stmts
END WHILE
The code begins by evaluating a control operation. Based on the value returned by the operation, a conditional branch instruction is used to decide whether to execute the body of the loop. After executing the body of the loop, an unconditional branch is made back to the start of the loop. Optimizations can be made to the code by removing the NO-OP following the BRFALSE instruction and replacing it with the last instruction of Code(ctrlcall), and by replacing the NO-OP following the unconditional branch instruction with the last instruction of Code(stmts) (the code of the loop body).

3.2.7 The Type Operations: Initialization and Finalization

The code to call initialize and finalize operations for types has been shown. However, the format and the code for such operations has not been given, nor has the format for defining types been defined. These items are presented here.

<type decl>:
TYPE type_name IS REPRESENTED BY fac.type

EXEMPLAR exemplar

[ conventions ]
[ correspondence ]
[ model ]
[ type_init ]
[ type_fin ]

END type_name

A type definition gives a name (type_name) to the type being defined, and specifies the name (fac.type) of the type used to represent it. The EXEMPLAR is representative of all variables of the type in the remainder of the definition. It is used to define such things as the code to produce an initial value of variables of the type.

The conventions section explains basic assumptions made in the implementation of the type. The correspondence section describes how the representation of the type can be mapped to the specification of the type, and the model section defines an operation used for displaying variables of the type during testing. These sections are not used in code generation and will not be discussed further in this dissertation (for additional details see [Har90, Mur90]).

A type_init operation defines the code that is executed when a variable of the type comes into existence. The type_fin operation defines the code that is executed when a variable becomes no longer known. The form and code generation rules for these operations are presented below.
As shown above, a type initialization operation has no parameters. However, it does return a result (an initialized variable). The result is referred to in the code of the operation by using the exemplar from the type definition. The exemplar is initialized at the beginning of the execution of the operation by calling the initialize operation of the representation type. Local variables may be declared and the statements section may modify the initial value given to the exemplar. The code generated for a type initialization operation is given below:
Type finalization works by receiving an implicit parameter (the exemplar), but not returning one. The code generated initializes local variables, executes the statements, and calls the finalize operation for the representation type to finalize the exemplar. The format of the operation and the code for the operation follow.

\[
\text{<type_fin>:

FINALIZATION

BEGIN

[ LOCAL VARIABLES

   var_decl*

END LOCAL VARIABLES ]

stmts}
\]
END INITIALIZATION

InitCode(LocVarDeclSec)
Code(stmts)
FinCode(LocVarDeclSec)
FinCode(exemplar_name)
RETURN_FIN

3.2.8 The Module Initialization Operation

Creation of every instance of a module also involves an initialize operation. This operation is invoked when the instance is created. It initializes the static variables associated with the facility, calls the initialize operations of modules that it instantiates, and records pointers to the static data of those facilities. A facility data record is constructed to store the facility data pointers and the static data. The operation is defined within the realization auxiliary section (shown below):

<real_aux_sec>:

REALIZATION AUXILIARY

[ FACILITIES
fac_decl*
END FACILITIES ]

[ CONSTANTS
const_decl*
END CONSTANTS ]

[ VARIABLES
var_decl*
END VARIABLES ]

[ conventions ]

[ correspondence ]

opr_decl*

[ model ]

[ real_aux_init ]

END REALIZATION AUXILIARY
The realization auxiliary section of a module may contain facility declarations. The declarations cause the compiler to generate calls to initialize operations of the declared facilities. These calls are placed in the module initialization code. The realization auxiliary section may also contain declarations of variables and constants—static facility data created by the initialization operation when the module is instantiated. The module initialization operation (real.aux.init) is defined in the realization auxiliary section. Facility declaration and module initialization will be discussed below.

The form of a facility declaration is as follows:

```
<fac.decl>:

fac_name IS conc_mod_name [(fac_conc_arg+)]

REALIZED BY real_mod_name [(fac_real_arg+)]

[ fac_ren_sec ]
```

A facility declaration, or instantiation, creates an instance of a module. The instance is given a name, fac.name, that can be used to access the types and operations that it provides. Instantiation names the concept, conc_mod_name, and fixes the conceptual parameters. It also selects a realization, real_mod_name, and fixes the parameters to the realization. Types and operations provided by the instantiated module may be renamed in the renaming section, fac_ren_sec.

The compiler generates a call to the initialization operation of each facility instantiated. The call is placed in the initialization operation of the client module. The
from of a module initialization operation is shown below:

<real_aux_init>:

INITIALIZATION

BEGIN

[ LOCAL VARIABLES
  loc_var_decl*
END LOCAL VARIABLES ]

stmts

END INITIALIZEATION

The code generated for a module initialization operation is responsible for initializing all facilities declared in the module, for initializing facility constants and variables, for initializing the local variables of the module initialization operation, for executing the statements of that operation, and for finalizing its local variables.

The activation record of a module initialization operation is stored on the LDS and has the form shown in Figure 27. It contains (beginning at the bottom) the facility data pointers (FDPs) of the facilities providing the conceptual and realization
Figure 27: The Activation Record of a Module Initialization Operation.
parameters to the module, its own FDP, the FDPs of locally declared facilities, the facility constants and variables, and the local variables. An FDP points to the location in IM of the facility data for a module.

The activation record (see Figure 27) is built as the module initialize operation executes. The FDPs of the facilities providing the parameters are passed as parameters to the operation. The FDP of the module being initialized is created by calling the mem.alloc operation of the memory manager facility. This FDP is also the value returned by the facility initialize operation. The FDPs of locally declared facilities are obtained by calling the initialization operations of those facilities. The pointers to the facility constants and variables and the pointers to the local variables are obtained by calling the appropriate type initialization operations. When all the pointers have been filled in and the code of the operation has executed, all the pointers except for the pointers to the local variables and this facility's FDP are copied to this facility's data record (see Figure 28).

The code for a module initialization operation is shown below. The following constants are used in the code:

1. cp—the number of parameters to the module's concept.

2. rp—the number of parameters to the module's realization

3. If—the number of facilities declared within the module.

4. fc—the number of facility constants declared within the module.

5. fv—the number of facility variables declared within the module.
Local Variables

Facility Variables & Constants

FDPs of local facilities

Result (FDP)

FDPs of parameters

pointers to facility data records

facility variables

LDS

Figure 28: The Construction of an FDR.
PUSH cp + rp + lf + fc + fv  ; size of facility data record
NEW_FR mem_mgr
CALL mem_alloc  ; alloc. fac. data rec.
NEW_FDR mem_mgr
RESTORE_FDR
RESTORE_FR
POPL 2  ; save fac. data rec. pointer
POP 1
Code(fac_decl_sec)
Code(const_decl_sec)
Code(var_decl_sec)
Code(real_aux_init)

; copy FDPs of fac_parms and local facilities to FDR (in IM)

; copy FDP of 1st conc. parm.
PUSHL lv + fv + fc + lf + rp + cp
ACCESS_0 0
POP 1

; copy FDP of 2nd conc. parm.
PUSHL lv + fv + fc + lf + rp + cp - 1
ACCESS_0 1
POP 1

...

;copy FDP of 1st realiz. parm.
PUSHL lv + fv + fc + lf + rp
ACCESS_0 cp
POP 1

;copy FDP of 2nd realiz. parm.
PUSHL lv + fv + fc + lf + rp - 1
ACCESS_0 cp + 1
POP 1

...

;copy FDP of 1st local facility
PUSHL lv + fv + fc + lf - 1
ACCESS_0 cp + rp
POP 1
;copy FDP of 2nd local facility
PUSHL lv + fv + fc + lf - 2
ACCESS_0 cp + rp + 1
POP 1

...

;copy module consts & vars to FDR
;copy FDP of 1st module constant
PUSHL lv + fv + fc - 1
ACCESS_0 cp + rp + lf
POP 1

;copy FDP of 2nd module constant
PUSHL lv + fv + fc - 2
ACCESS_0 cp + rp + lf + 1
POP 1

...

RETURN_INIT
NO-OP
A module initialization operation first allocates storage for its facility data record. The pointer to the storage is stored on the LDS.

After allocating the storage, the initialization operations of the locally instantiated facilities are called. For each parameter to a facility, the FDP of the facility providing it is passed to its initialize operation, so that the initialize operation of the instantiated facility can record the FDP in its facility data record. A facility may provide more than one parameter to another facility, e.g., an operation and a type provided by the same facility may be passed as parameters to another facility. In such a case, an FDP would appear more than once in the list of parameters passed to a particular module initialization operation.

To permit this, ARC provides an instruction for pushing a copy of a value (remember that the PUSH instructions destroy the value stored in the source word). The instruction is only used in the code of facility initialization operations, and serves the single purpose of pushing FDPs. Therefore, the instruction is called PUSHFDP. The instruction has one argument, "n". It copies the value at offset "n" from the old top of LDS to LDS[top].

Following the code for initializing the declared facilities, the facility constants and variables are initialized. This is exactly like the initialization of local variables of other operations. The code of real_aux.init is generated next. It contains code to initialize the local variables, to execute the statements of the operation, and to finalize the local variables. Lastly, the relevant items are copied from the LDS to the facility data record in IM.
Figure 29: LDS for the Initialization Operation of the Stack Module.
As an example, consider the module initialization operation for the stack module. Figure 29 shows the LDS contents for successive stages in the execution of the operation. The contents of the LDS when the operation begins execution is shown first. The LDS has one entry, corresponding to the type parameter to the stack module. The pointer to the FDR of the facility providing the type (T) is placed on the LDS prior to calling the initialization operation. The state of the LDS after the initialization operation has allocated storage for the FDR of the stack instance is shown next: the pointer to the stack facility's FDR is on top of the LDS. The state of the LDS after the locally declared facilities (record, array, and integer) have been initialized is shown last. Pointers to the FDRs of the local facilities are stored in the activation record of the module initialization operation. The pointers must be retained after the operation returns, so their values are copied into the FDR of stack (located in IM).

The code generated for the module initialization operation of stack is shown below:

```
PUSH 4 ; size of facility data record
NEW.FR mem_mgr
CALL mem_alloc ; alloc. fac. data rec.
NEW_FDR mem_mgr
RESTORE_FDR
RESTORE.FR
POPL 2 ; save fac. data rec. pointer
POP 1
```
Code(fac_decl_sec)
Code(const_decl_sec)
Code(var_decl_sec)
Code(real_aux_init)

; copy FDPs of fac_parms and local facilities to FDR (in IM)

; copy FDP of 1st conc. parm. (T)
PUSHL 4
ACCESS_0 0
POP 1

; copy FDP of 1st local facility (Integer)
PUSHL 2
ACCESS_0 1
POP 1

; copy FDP of 2nd local facility (Array)
PUSHL 1
ACCESS_0 2
POP 1
3.3 Primitive Modules

ARC must make it possible to write primitive RESOLVE modules. These modules occur at the lowest level of the abstraction hierarchy of programs. I.e., most modules are constructed using some subset of them. Primitive modules are coded directly in the assembly language of the virtual machine, so instructions must be provided to produce efficient implementations of these modules, especially since the operations of these modules are called frequently. RESOLVE programs require the following primitive modules: a memory manager, and modules providing the ADTs boolean, integer, pointer, array, record, and character string. An overview of each module is provided. Several new instructions are introduced where it is shown how implementations of certain primitive modules benefit from their existence.

The memory manager module is responsible for managing the free storage of IM. It is linked with every program. Its initialization operation, the first operation to execute when a program is run, initializes tables in IM and then calls the initialize
operation of the main module. The memory manager also provides operations to allocate blocks of memory and to return blocks of memory to free storage.

The allocate operation is called by initialization operations of most types to create storage for variable representations, and the free operation is called in finalization to release storage. Due their high frequency of use, these operations must execute as quickly as possible.

Efficient implementation of the memory manager module is facilitated by the following machine instructions: PUSHI, PUSHI.O n, POPI, and POPI.O. In the descriptions of these instructions, a "#" is used to refer to the processor state before the instruction executes.

- **PUSHI** — Push indexed memory item. (See Figure 30.) Increment the top-pointer of LDS. Move value of IM[a + i] onto the top of the LDS, where "a" and "i" are on top of the LDS before the instruction executes. Destroy the value of IM[a + i]. Retain "a" and "i" on the LDS.

- **PUSHI.O n** — Push indexed memory item, offset is immediate. (See Figure 31.) Increment the top-pointer of LDS. Move the value of IM[a + n] to the top of the LDS, where "a" is on top of the LDS before the instruction executes. Destroy the value of IM[a + n]. Retain "a" on the LDS.

- **POPI** — Pop item to indexed memory. (The reverse of PUSHI; see Figure 30.) Move value on top of LDS to IM[a + i]. Subtract 1 from the LDS top-pointer, where "a" and "i" are on top of the LDS before the instruction executes. Retain "a" and "i" on LDS.
Figure 30: Execution of the PUSHI Instruction.
Figure 31: Execution of the PUSHI_O Instruction.
• POPI.O n — Pop item to indexed memory, offset is immediate. (The reverse of PUSHI.O; see Figure 31.) Move value on top of LDS to IM[a + n]. Subtract 1 from the LDS top-pointer, where “a” is on top of the LDS before the instruction executes. Retain “a” on LDS.

The PUSHI and POPI instructions are useful for reading and writing array entries, since they allow a base and an offset to be specified on top of the LDS. The base can be the starting address of an array in IM and the offset can refer to the distance of an entry in the array from the base of the array. The memory manager benefits from these instructions for organizing free storage.

The memory manager also uses records, so it benefits from the existence of the PUSHI.O and POPI.O instructions. These instructions can be used to access a record by placing the base address of the record on top of the LDS and the offset of a field of the record in the instruction itself.

Another fundamental module is the boolean template. It provides the type boolean and operations for setting and testing the values of boolean variables. This template benefits from the logical instructions (CLRZ and CLRN), the conditional branch instructions, and the RETURN_TRUE and RETURN_FALSE instructions (which have been explained previously).

The integer template provides the type integer and a basic set of operations for manipulating integers. This module uses the SUB, ADD, CLRZ, and CLRN machine instructions. In addition, a couple more instructions are required:

• MIN_ALLOWED — Get minimum value allowed. Increment top-pointer of
LDS. Place \texttt{min\_value} on top of the LDS, where \texttt{min\_value} is the smallest integer value that can be stored in a single word in a particular implementation of the virtual machine.

- \textbf{MAX\_ALLOWED} — Get maximum value allowed. Increment top-pointer of LDS. Place \texttt{max\_value} on top of the LDS, where \texttt{max\_value} is the smallest integer value that can be stored in a single word in a particular implementation of the virtual machine.

These instructions are required for implementing operations that return the largest and smallest integer values allowed by the integer template. They allow the built-in integer type to provide integers represented using the number of bits contained in data words of any implementation of ARC. The code for these operations is portable to any implementation of ARC, regardless of the word size.

The pointer template provides a way for the pointer type to be safely encapsulated in a module. It can be used to provide implementations of linked structures (such as lists) without the problems typically associated with pointers [Pit90].

One new machine instruction is needed to implement the module:

- \textbf{DTS} — Duplicate item on top of the LDS. (See Figure 32.) Increment top-pointer of LDS. Place onto the top of the LDS a copy of the value immediately beneath the new top.

The DTS instruction permits an item on the LDS to be copied. The pointer module requires this capability to copy the value of a pointer. Note that the PUSHFDP
Figure 32: Execution of the DTS Instruction.
instruction is inadequate for this, since it only copies an item on the LDS. The pointer module also needs the ability to copy pointers stored in IM.

The array template provides the ability to manage vectors of items of arbitrary type. Its implementation is simplified by the existence an ACCESS instruction:

- **ACCESS** — Swap LDS entry with IM entry. (See Figure 33.) Swap the item on top of LDS with item in IM[a + i]. “a” and “i” are the second and third items on the LDS before the instruction executes. Retain “a” and “i” on LDS.

The operation for obtaining the value of an array element swaps the desired array element with the value of the parameter to the operation. For example, to place the value of the nth item of an array “a” and into the variable “x”, the following procedure call is written in RESOLVE:

```plaintext
access(a, n, x)
```

This sets a[n] equal to the value “x” had before the call and sets “x” equal to the value a[n] had before the call. The contents of the LDS at the time the array access operation begins execution is precisely what is required by the ACCESS machine instruction. Thus, the code to implement the array access operation will look like:

```plaintext
RETURN
ACCESS
```

In RESOLVE, a different template is provided for each different size of record. Record templates provide operations for accessing each of the fields of a record. For
Figure 33: Execution of the ACCESS Instruction.
example, a record with three fields provides the operations "access1", "access2", and "access3" for accessing the first, second, and third fields of a record. An operation on a record swaps the value of one of its parameters with the value of a particular field of the record, e.g., the procedure call access3(r, x) swaps the value of "x" with the value in field3 of record "r". To efficiently implement record accesses, the ACCESS.0 machine is provided (defined previously).

The parameters to record access operations are in the order required by the ACCESS.0 instruction. Thus, the code for the operations is very short. The code for access3 is:

```
RETURN
ACCESS.0 3
```

The character string template provides the ability to convert integers to strings of characters, and to perform the opposite conversion. It also has operations for performing input and output. The latter two operations require machine instructions to read and write a character (see Chapter 2).
CHAPTER IV
PARALLEL EXECUTION OF RESOLVE PROGRAMS

The use of parallel execution as a means of improving the execution speed of RESOLVE programs is explored in this chapter. A model of parallel execution appropriate for RESOLVE programs is developed. The virtual machine is treated both as the underlying architecture for execution of RESOLVE programs, as well as a node in a parallel processor supporting parallel execution.

Software support for executing RESOLVE programs in parallel using ARPC is also considered. The run-time system presented in chapter 3 is extended to permit the use of multiple processors. The mapping of program components onto PEs is briefly discussed. Several improvements are made to the architecture presented in chapter 2, and implementation details of the processor are examined.

4.1 Model of Parallel Execution

In this section, a model of parallel execution suited to the kind of reusable software components characterizing RESOLVE programs is developed. The model is illustrated using a sample program. Features of RESOLVE that make the model attractive are discussed and limitations of the model are mentioned.
4.1.1 The Model

RESOLVE programs can be executed in parallel on a distributed memory machine as follows. Each PE contains the operations of one or more facilities and (possibly) the representations of one or more variables. The code of a facility may be replicated on multiple PEs if necessary to increase parallelism. A variable's representation is placed on a PE where the operations of the facility providing its type are located. To hide the latency of a remote call, an operation is permitted to continue execution until it attempts to access a "locked" variable (an asynchronous remote procedure call or, ARPC). A variable is automatically locked when it is passed as a parameter to a call and is unlocked upon return of the call. An operation attempting to access a locked variable must wait for a remote call to return before retrying the access.

The grain-size of parallelism might be called "variable-level", for two reasons. First, the parallelism arises from performing operations on different variables at the same time. Second, variables may denote values with small (fine-grain) or large (large-grain) representations. An example of the former is a 16-bit integer, while the latter might be an associative search abstraction represented by a B-tree. The language and the parallel execution method treat these situations uniformly.

To illustrate the model, consider a procedure (call it "client") that declares variables of types "real" and stack (of real). Assume that client is a procedure within a module called "main". A segment of client's code is shown below.

```plaintext
procedure client
```
local variables

s1, s2: stack
x, y, z: real

end local variables

begin

    ...
    pop(s1, x)
    pop(s2, y)
    pop(s1, z)
    ...

end client

Client has two variables "s1" and "s2" that contain stacks of real numbers. It also has three real number variables "x", "y", and "z". The code of client contains a statement that pops a value from s1 into x, followed by a statement that pops a value from s2 into y. Since there are no dependencies between these two statements, they could execute in parallel using ARPC. Some of the parallelism exploitable by this approach can be visualized by examining the annotated program below. The parallel lines that appear next to some statements in the figure indicate that statements could execute in parallel. The calls (generated by the compiler) to initialize and finalize the variables have been included in the program. The initialization (and finalization) of all variables can proceed in parallel.
procedure client

local variables

s1, s2: stack
x, y, z: real

end local variables

begin

// s1 := stack_init()
// s2 := stack_init()
// x := real_init()
// y := real_init()
// z := real_init()

// pop(s1, x)
// pop(s2, y)
// pop(s1, z)

// stack_finalize(s1)
// stack_finalize(s2)
// real_finalize(x)
// real_finalize(y)
ARPC can achieve parallel execution at multiple levels in the abstraction hierarchy. For example, in the program above, not only could the pop operations execute in parallel, but operations on the constituent objects of the stacks' representations could also execute in parallel. To see how this would work, assume that both $s_1$ and $s_2$ are represented as arrays (of reals) with integer top pointers.

The code for the pop operation (shown here for reals, but in reality a generic procedure) is shown below:

```
procedure pop

parameters
    alters s: stack
    produces x: real
end parameters

local variables
    contents: array
    top: integer
end local variables

begin
```
The array and the top pointer of the stack are stored in a record with two fields. Operations on records, arrays, and integers are called to achieve a pop. Some of these operations can be performed in parallel as seen in the annotated program below:

```plaintext
procedure pop

parameters
    alters s: stack
    produces x: real
end parameters

local variables
    contents: array
    top: integer
end local variables
```

rec_access(s, 0, contents)  (* get contents field *)
rec_access(s, 1, top)  (* get top index field *)
array_access(contents, top, x)  (* swap out top element *)
decrement(top)
rec_access(s, 0, contents)  (* put back contents field *)
rec_access(s, 1, top)  (* put back top index field *)
end pop
```
begin
  // contents = array_init
  // // top = integer_init
  // rec_access(s, 0, contents)
  rec_access(s, 1, top)
  array_access(contents, top, x)
  // decrement(top)
  // rec_access(s, 0, contents)
  // rec_access(s, 1, top)
  // // array_finalze(contents)
  // // integer_finalze(top)
end pop

4.1.2 Features of RESOLVE Exploited by the Model

The ARPC model is appropriate for RESOLVE programs due to certain features of the language. These features also permit a simple hardware implementation of the model.

The absence of aliased variables simplifies detection of data dependencies. Each variable is represented as an address referring to the location of the representation of its value. These addresses are passed as parameters in the code generated for an operation call, even though RESOLVE programmers are not aware that only the address is actually passed. (They think that swapping occurs, but a simpler implementation that still yields the correct results is employed.) There is only one
accessible copy of the address of the data structure representing a variable's value at any time. When a copy of a parameter's address is pushed onto the LDS as a parameter, the source word becomes inaccessible. If aliasing were permitted in RESOLVE, a more complicated pointer analysis would be required.

The requirement that the arguments to a call must be unique allows the hardware mechanism that locks variables to be uncomplicated. When a variable is pushed as a parameter it is locked and only the copy pushed onto the parameter stack is accessible. With this simple technique, duplicate arguments in a call would result in deadlock. An attempt to push the second copy of an argument would cause the processor to wait for it to be unlocked. However, it would not be unlocked until the return of the operation to which it was passed, and the call to that operation depends on the unlocking of the parameter. Thus, if duplicate arguments are permitted, a more complicated detection mechanism or the removal of these duplicates by the compiler is required.

Since operations cannot have global variables, accesses to variables are easy to synchronize. Access to facility variables is controlled simply by not replicating the code of a facility's operations when the facility contains static data. In practice, most RESOLVE modules do not have static data.

The frequent occurrence of calls to operations is a result of using highly cohesive modules when building a software system (a desirable approach from the perspective of reuse). An effect on a data structure will frequently require calls to operations of modules at levels lower in the hierarchy of abstractions than the module providing
the type of the data structure. The model of parallelism proposed encourages the
development of layered software, since the source of parallelism is operation calls and
the more layers there are in the system, the greater the potential for parallelism.

4.1.3 Evaluation

The model of execution encourages the development of layered software by increasing
parallelism in correspondence to increases in layering. However, it would seem that
the execution model applies only to sequences of statements that occur statically in a
program’s code. Can it be applied to sequences of statements that occur dynamically;
specifically, can it permit iterations of loops to execute in parallel? ARPC can be
used to execute loop iterations concurrently, but pre-run-time assistance by software
is required.

The problem is in serializing accesses to data structures. For example, there must
be a way to permit the elements of an array to be accessed simultaneously. A method
for allowing this has been proposed in the literature [CD90] and is called concurrent
aggregates (CAs). CAs are data structures that can be accessed concurrently. They
have multiple “representatives” on different PEs that manage portions of a data
structure. When a call is made to access a CA, the run-time system decides which
representative is to handle the call and forwards the call to it.

A similar technique can be used in the parallel execution of RESOLVE programs.
The number of representatives can be chosen when a module is instantiated or when
a variable is declared. In the first case, the number indicates how many distributed
components all variables declared using a type from the module will have. In the
second case, the number of representatives for each variable is chosen at declaration. To initialize a variable, several memory-allocate operations need to be called—one for each representative; each ideally is called on a different PE. A variable initialize operation returns a list of pointers to the segments of the data structure managed by the representatives; the variable is stored as a vector of pointers. A module needs to provide an operation for deciding which representative to call when an operation on an aggregate is invoked; this operation resides on every PE where operations on the aggregate can be called. Thus, accesses to data structures need not be serialized. Therefore, a loop that accesses a data structure such as an array can be concurrent.

However, using the model of execution there are problems concurrently executing an algorithm written using recursion in place of a loop. The easiest way to handle this is to not write programs in that way, but if this limitation is too severe, it is possible to transform recursive algorithms into iterations.

The use of CAs increases the amount of parallelism extractable from programs. However, CAs may not be compatible with the parameter passing mechanism call-by-swap. It would seem that the semantics of call-by-swap prevents multiple simultaneous accesses to a data structure; the first call to an operation that accesses the structure locks it, and subsequent calls to operations to access it block. This difficulty is easily overcome—the implementation of calls to operations of an aggregate need not lock the entire data structure. The problem then becomes one of permitting RESOLVE programmers to be cognizant of this detail, so that code can be written
in a way that permits this type of parallelism to be extracted; if this raises concerns that code will be architecture specific when written in this way, then interactive compilation (like that used in vectorizing compilers [PW86]) can be used to extract this kind of parallelism from non-architecture specific programs.

There are two main reasons not to rely on CAs for parallel execution of RESOLVE programs. First, their use requires modifications to RESOLVE and to the run-time system. Second, recursion can be used as the basis for writing programs to be executed in parallel. Before performing an operation on the elements of an abstraction containing homogeneous elements (such as an array or a sequence), the data structure is first split into disjoint segments. The operation is then called once for each segment. Since the segments are disjoint, operations on them can proceed in parallel.

As an example, consider a data structure containing sequences of integers. To compute the sum of all integers in the sequence “s1”, one might write the following:

```plaintext
function add returns sum: integer

parameters
   alters s1: sequence
end parameters

local variables
   alters s2, s3: sequence
end local variables
```
begin
  if size(s1) > 1
    then
      split(s1, s2, s3)
      sum := add(s2) + add(s3)
    else if sequence.size(s1) = 1
      sum := remove(s1, 0)
    else sum := 0
  end add

Add splits the sequence s1 into subsequences s2 and s3. It then calls itself once with s2 as a parameter and once with s3 as a parameter, unless the size of the sequence is less than two, when the result is computed directly. The two recursive calls are executed in parallel under the model of ARPC.

4.2 Software Support for Parallel Execution

The parallel execution of RESOLVE programs requires some assistance from system software. This section considers the mapping of program components onto processors and examines the issues of memory allocation and system startup.

Mapping Strategy

The mapping of the components of a program onto ARC in a way that exploits its features is performed in two phases (see [WW90c]). The first phase, called logical
mapping, determines the maximum amount of parallelism attainable from a program in the ARPC model of parallel execution. The second phase, called physical mapping, uses the results of the first phase to map a program's components onto a physical realization of ARC. Logical mapping is performed once for each program, but physical mapping is performed every time that program is used on a different realization of ARC.

Logical mapping is performed by mapping program components onto logical processors (also called logical PEs). A distributable component is defined to be the data structure representing a variable's value and the code to access it. Each distributable component can reside on a processor by itself. It is assumed that there is an infinite number of logical PEs, and that the communication cost between two logical PEs is uniform for any pair of logical PEs. Logical PEs and their interconnections are modeled using a directed acyclic graph (DAG).

Physical mapping maps the logical processors onto physical processors (physical PEs). Physical constraints such as the number of PEs available and communication costs are considered at this point.

### 4.2.1 DAG Representation of Programs and Construction of the DAG

In this section it is shown how to construct a DAG to model the potential parallelism in a software system. The DAG for a particular program shows the relationships among its distributable components, and the maximum amount of parallelism attainable with the model of parallelism used. The graph can be used to map the
components onto ARC.

A program is modeled by a DAG, $G = (V, E)$, where:

- $v \in V$ denotes the operations of a facility, $f(v)$, and the representations of one or more variables whose types are provided by $f(v)$;

- $(x, y) \in E$ indicates that the code of facility $f(x)$ calls some operation(s) provided by facility $f(y)$; and

- There exists exactly one vertex in $G$ with indegree 0, representing the facility at the highest level of the abstraction hierarchy.

The DAG representing a particular program can be constructed as follows:

1. Place a vertex in the graph for each facility used in the program.

2. Place an edge in the graph for each call dependency in the program. Only calls between operations of different facilities are represented in the graph.

Each vertex in the graph can be thought of as a logical processor. We assume there are an infinite number of logical processors and that these processors can be connected in an arbitrary fashion.

The DAG for a stack client is shown in figure 34. Main chooses a stack realization that uses a record with two fields. The first field is an array and the second field is an integer. The program also uses a real number module. The DAG contains a node for each module instance used. Edges between nodes denote calls to operations of one facility by operations of another facility; e.g., the client operation of the main module calls operations of the stack facility and the real number facility.
Figure 34: The DAG for a Sample Program.
The amount of parallelism attainable from the graph can be increased by replicating some of the facilities. The goal is to have one vertex in the graph for each (static) distributable component (a variable plus the code to access it). This requires the code of a facility to be replicated. A facility's code can be replicated safely iff (1) it has no static data and (2) it provides no operations with two or more parameters of a type that it exports.

Replication of a facility is reflected in the graph by adding a new vertex. Edges are also added to the graph so the new vertex is adjacent to the same vertices as the vertex representing the facility being replicated; i.e., when a vertex is replicated, so are all edges connected to it.

All edges adjacent to a vertex must be replicated when the vertex is replicated due to the effects caused by swapping the values of two variables. When the values of two variables are swapped, their representations may change PEs. The DAG must represent all possible situations that could arise during execution, so it must contain edges showing all potential calls between components. Specifically, if facility "x" uses a type provided by facility "y", the DAG must contain edges from all vertices representing copies of "x" to all vertices representing copies of "y".

For example, the DAG shown in figure 34 can be expanded to increase potential parallelism. One possible logical mapping is shown in figure 35. The squares represent logical processors. Since there are two stack variables, the code of the stack facility is replicated twice; the nodes containing representations of the stacks are also replicated so the representation of a single variable can be placed on each node. The code for
Figure 35: One Possible Logical Mapping for a Sample Program.
the real number facility is replicated once for each real number variable declared in main. Replication is performed once for node in the original DAG by determining the number of local variables of each type declared in each module.

The logical call graph for a program shows all potential interprocessor calls. The logical call graph for the sample program can be seen in figure 36. To simplify the drawing, logical processors containing clones of the same module are clustered. Edges incident to a cluster are incident to all logical processors in the cluster.

The logical mapping for a program shows the parallelism possible if the number of PEs is unlimited and if the PEs can be connected in an arbitrary fashion. The next section discusses issues that arise when the logical processor graph is mapped onto a physical machine.

**4.2.2 Deadlock-Free Mapping of Components onto Physical Processors**

After the DAG for a program has been constructed, it is used to map the components of a program onto physical PEs. The same logical mapping is reused each time a program is physically mapped onto a different physical realization of ARC. If the number of vertices in the graph (the number of logical PEs) is less than the number of physical PEs in an implementation of ARC, the physical mapping is trivial if maximum parallelism is the only concern; place the content of each logical PE onto its own physical PE. This assumes that the interconnection of the logical PEs can be modeled in the physical machine. If the number of logical PEs is greater than the number of physical PEs, existing techniques for embedding graphs are employed
Figure 36: The Logical Call Graph for a Sample Program.
The application of such techniques is not discussed here, since the purpose of this work is to show how ARC's model of parallel execution is intended to be used.

The tradeoffs between static and dynamic mapping of local variables should be weighed. Static mapping decides where (on which PEs) to place the local variables of an operation before execution of the program, whereas dynamic mapping decides each time an operation executes. The overhead of dynamically mapping the local variables of an operation should be substantially smaller than the cost of executing the code of the operation. An efficient hardware assisted dynamic mapping scheme is described later in this chapter.

There are other important issues to consider when mapping components onto physical processors. A mapping incorporating these considerations has not yet been developed, so these issues are only mentioned for completeness. The communication distance between the PE containing the code of an operation and the PEs where its local variables are stored can affect performance. Granularity of operations and frequency of use of operations should be weighed when deciding how many copies of operations to make available. Load balancing and utilization of PEs are factors to be considered in conjunction with the desired level of parallelism.

A physical mapping for the stack client example is shown in figure 37. Boxes represent physical processors. The code for the real number facility is placed on processors with the code of other facilities to improve processor utilization.

Deadlock can result on ARC if care is not taken in placing components onto physical PEs (see [WW90b]). It can occur if a cycle exists in the "physical call graph"—a
Figure 37: A Physical Mapping for Client.
graph in which vertices represent physical PEs and edges represent potential calls between operations of components on different PEs. The reason it can occur is that once a processor has accepted a call, it will not accept any others until the called operation has completed.

A physical call graph for client is shown in figure 38. As in the logical call graph, PEs containing identical code are clustered. The mapping represented in the figure has the potential for deadlock because of the existence of cycles in the physical call graph. For example, client can call an operation of stack; a stack operation can call an operation of array; and an operation of array can call an operation of real. A cycle also exists involving main, stack, record, array, and real. These cycles can be removed by moving the code for real from the same physical PE as main to its own PE (see figure 39).

To avoid deadlock, the mapping is constrained to prevent cycles in the physical call graph. This can be achieved by using the relationships represented in the logical call graph. Another way to avoid deadlock is to allow multiple threads of execution per PE; this solution was not chosen in ARC because it would significantly complicate the PEs.

Recursion was identified as the programming technique to be used in place of loops for performing operations on large data structures. Special consideration is required when mapping facilities containing recursive operations onto logical PEs. A single copy of the code for such a facility in the logical graph represents a binary tree of copies of the code; the number of copies of the code actually used is determined by
Figure 38: The Physical Call Graph for a Mapping of Client.
Figure 39: Deadlock-Free Physical Mapping of Client.
the physical mapping algorithm by taking into account factors such as the number of processors. An attribute associated with such facilities in the logical call graph identifies them.

An attribute is also used to identify another kind of entry in the logical call graph—facilities providing types whose representations fit into a single data word, e.g., the integer module. The identity of such modules is important in constructing efficient physical mappings. It is inefficient to generate a remote call to access an integer when the representation of the integer is stored locally. Thus, the code for such facilities is replicated on each physical PE calling it. In this way inter-PE calls to integer are not generated, hence cycles are not introduced as a result of this optimization.

4.2.3 Run-Time System Issues

The memory allocator module presented in Chapter 3 is altered slightly to accommodate efficient parallel execution. It is distributed to every physical PE to avoid a bottleneck due to contention for its services.

The initialize operation of the memory allocator described in chapter 3 calls the module initialize operation of the main module. Since there are many copies of the allocator in parallel execution, a startup module is introduced for the purposes of calling the initialize operations of the memory allocators and calling the initialize operation of the main module.
4.3 Parallel Implementation of the Virtual Machine

One reason the ARPC model is attractive is that it can be implemented in hardware without complex logic (see [WW90a]). To substantiate this claim, the architectural support necessary for a distributed memory implementation of the virtual machine that supports ARPC is outlined. Details of the interface between PEs are also given.

4.3.1 Dynamic Extraction of Parallelism From Object Code

Programming statements that refer to architecture-specific features (such as multiple processors) should be avoided within reusable components [MW88, JKC89] to promote simplified design, improved portability, and increased reuse. For these reasons, RESOLVE provides no constructs for specifying parallel execution. Automatic extraction of parallelism is one means of achieving parallel execution for RESOLVE programs.

A compiler will generally not be able to parallelize programs since they are composed of separately compiled modules. In a mature reusable components industry, it is likely that vendors will only provide formal specifications and object code of modules [OWZ89]. Although the compiler can perform some local optimizations for individual modules, it cannot perform global (interprocedural) optimizations. Thus, extraction of parallelism from programs will have to be performed, at least in part, using object code and formal specifications of the separate components.

A method of extracting parallelism from object code at run-time has been devised. A "busy" bit is associated with each data word; i.e., each entry on the LDS and in IM
has a busy bit. (The term busy bit was first used by Tomasulo [Tom67] in describing a mechanism used to synchronize accesses to registers on the IBM 360 by arithmetic units.) The data movement instructions of ARC turn on the busy bits of the source words and turn off the busy bits of the destination words. When an instruction attempts to access a word having its busy bit on, the processor waits idly until a remote operation returns. At that point it retries the access.

The client procedure is used to illustrate the use of busy bits. The contents of PEs after the initialization of client’s local variables is shown in figure 40. A pointer to a variable’s representation is shown as a white square. A black square indicates a variable’s representation. Each stack is represented as a record with two fields, thus the representations of $s_1$ and $s_2$ are on PEs where the code for records is located. The items in the records are themselves pointers to the contents arrays ($c_1$ and $c_2$) and the top pointers ($top_1$ and $top_2$).

Pointers to variables are stored on the LDS, representations of variables in IM and operations in instruction memory. This is illustrated in figure 41.

The ARC code that a compiler generates for client is shown below:

```
pushl s1
call pop
pushl x
pushl s2
```
Figure 40: Data Distribution.
Figure 41: Contents of PEs.
call pop
pushl y

pushl s1
call pop
pushl z

The parameters to calls are placed on top of the LDS using the pushl instruction. The instruction to push the last parameter to a procedure is placed after the call instruction, since delayed branching is used. Instructions to restore the parameters after calls return are not required—this is done automatically.

Figure 42 shows snapshots of the LDS of PE1 at various stages of preparation for the call to pop. The first snapshot shows the LDS before any parameters are pushed. The second indicates that “s1” has been pushed—an entry containing “s1” appears on top of the LDS. The asterisk next to the original entry for “s1” shows that the busy bit of “s1” is turned on. As seen in the last snapshot, analogous actions are taken when “x” is pushed.

Figure 43 shows the call packet for pop being sent between PEs. Logically, the call packet consists of the address of op2 (denoted as PC), and the parameters, “s1” and “x”.

The LDS of PE1 after the call packet is sent to PE2 can be seen in Figure 44. The parameters pushed onto the LDS of PE1 prior to the call have been removed.
Figure 42: LDS of PE1.
Figure 43: The Call to Pop.
Figure 44: LDS of PE1 and LDS of PE2.
and "s1" and "x" are still busy; they will remain busy until pop returns. The LDS of PE2 after the call is also shown in Figure 44. It contains two entries—"s1" and "x"; their busy bits are not set. As illustrated in this example, the busy bit scheme allows only one copy of a variable's pointer to be accessed at any moment.

The second call to pop will set the busy bits of "s2" and "y" on PE1 and will send a call packet to the PE holding the second copy of the stack code.

The third call to pop will cause PE1 to block until the first call returns. The block occurs when the pushl instruction attempts to push s1. When the first call returns s1, its busy bit is turned off and the the blocked call is retried, and succeeds.

In addition to a busy bit being associated with each data word, there is also one for the control flag. It is needed to prevent race conditions that would occur if a processor permitted two control functions to be outstanding at the same time. Recall that control functions are used in the conditional sections of if statements and while statements and that control functions return the value yes or the value no. Control functions are implemented on ARC by using the control flag to return their results. If a call to one control function is outstanding, then another call to a second control function should not be permitted to proceed; otherwise one could not predict which function would return first and incorrect results may result. To prevent this situation from occurring, the busy bit of the result flag is turned on when a call to a control function is made; it is turned off when the control function returns. A call to a control function—made by using the CALL.CTRL machine instruction—is not permitted to execute until the busy bit of the control flag is turned off.
Conditional branch instructions also depend on the busy bit of the result flag being turned off in order to complete execution; this is to ensure that branches depending on the value returned by a particular control function do not execute until that function has returned.

4.3.2 Asynchronous Remote Procedure Call

In this section the requirements of ARPC for both the virtual machine and its implementation are discussed. Tables, registers, and machine instructions needed for calling operations on remote processors are described. The construction and contents of call packets and return packets is discussed. The efficiency of calls is improved by the presence of the operation processor (OP), which dispatches calls in parallel with the execution of instructions.

Tables Needed for Calling Operations

Run-time facility records (RFRs) are introduced in chapter 2 as a means for all instances of a realization to share code. This capability allows code to be generated once for each realization. When distributing RESOLVE programs, it is desirable to duplicate code of facilities and place copies (clones) on more than one PE. Code may still be generated once for each realization. Furthermore, the benefit of parallel execution is achieved by allowing operations provided by a particular instance to execute in parallel.

All instances of a realization that are mapped to the same PE share code. To permit this, a unique RFR is created for each clone of each instance of each facility.
An RFR is placed on each PE where a clone of a facility is located. RFRs contain pointers to the code of the operations provided by the facility. The RFR for a facility also contains entries for each module used by the facility.

In addition to RFRs, facility clone tables (FCTs) are used to call operations, because it is necessary to determine which copy (clone) of an instance is being called. An FCT is placed on each PE where an operation of a facility could be called. The RFR entries for facilities providing called operations are addresses of FCTs. This is illustrated in Figure 45.

The contents of an FCT is shown in Figure 46. An FCT contains an entry for each clone of a facility's code; an entry contains two fields: PE-id and RFR-addr. The PE-id identifies a physical PE containing a copy of the code to be called. The RFR-addr field is the memory address of the RFR for the clone of the instance being called.

An example of the FCT for an instance of the array template is shown in Figure 47. The code for the instance of array has been replicated three times and placed on PEi, PEj and PEk. PEi contains the RFR for a stack implemented using a contents array, and thus contains a pointer to the FCT of the array facility. The FCT of the array facility contains entries for the clones of the code of the array facility; the entries are addresses of the RFRs of the clones. The RFRs contain pointers to the code of the operations provided by the array facility.

Call instructions can be divided in two types according to how the decision of which clone's code to call is made. Case 1 occurs when calling an operation having a
Figure 45: RFRs and FCTs.
Figure 46: A Facility Clone Table (FCT).
Figure 47: A Facility Clone Table for an Array Instance.
parameter of a type exported by the same module providing the operation; in addition, the operation must be provided by a primitive module (one whose operations are implemented directly in the machine language of ARC). In such a case, the clone on the same PE as the representation of the actual parameter is called, since the operation directly accesses the representation. The address of the parameter is used to determine which copy of the operation to call. Thus, data words are tagged with a clone identifier—an offset into the facility clone table.

Figure 48 shows the hardware implementation of the case 1 CALL instruction. The FR contains the address of the RFR for the facility providing the operation currently executing. The instruction has four fields: opcode, fac, parm, and opr. The fac field of the instruction indicates the offset in the caller’s RFR of the entry that points to the callee’s FCT. The parm field identifies the parameter whose type is provided by the same module as the one providing the called operation; it indicates the offset from the top of the LDS where that parameter is located. The opr field of the instruction is the offset into the callee’s RFR where the address of the called operation is stored.

The following steps are taken to call an operation. The value of the FR is added to fac to obtain an address in FM. The value in the specified location of FM is added to the clone# tag of the LDS entry indicated by parm to obtain another FM address. The value at that location of FM is added to opr to obtain the address in FM of the pointer to the code of the called operation.

Case 2 of the call instruction is used when calling operations that have no pa-
Figure 48: Case 1 of the CALL Instruction.
rameters, when calling initialize operations, and when calling operations provided by synthesized modules (those whose operations are not implemented directly in the machine language of ARC, but are implemented in RESOLVE). In such cases, any copy of the operation may be called.

Figure 49 shows the implementation of the case 2 call instruction works. It is the same as case 1 of the CALL instruction with the following exception. The decision of which clone to call is made dynamically by using the first two fields of an FCT—"number of clones" and "last clone used". The number of the clone called is determined by the expression:

$$\left\lceil \frac{(\text{last clone used} + 1) \mod (\text{number of clones})}{2} \right\rceil$$

The value of last clone used in the FCT is incremented (mod number of clones) following the read of its value, so that the next call to an operation of the facility is sent to a different clone.

A ramification of having two kinds of call instruction is that the code generator must decide which kind of call instruction to use. To do this it must know whether modules providing called operations are primitive or synthesized. Since source code of modules may not be available, this information must be contained in the intermediate representation of modules. One intermediate form is called the compiled realization module (CRM) and is described in [Heg89]. The CRM representation could be enhanced with an additional field providing this information.

Clone selection is performed dynamically to help balance the load among the PEs containing the clones. This is not a global load balancing scheme. To broaden the
Figure 49: Case 2 of the CALL Instruction.
scope of the load balancing, two steps can be taken. The first step has to do with the values initially placed in the last clone used fields of the FCTs for a facility; these values should vary as widely as possible between FCTs. That is, if there are five clones of a facility, and the FCT for the facility is placed on ten machines, then the values of the last clone used fields in the ten FCTs should contain an even distribution of the values 0, 1, 2, 3, and 4; two FCTs should initially contain the value 0 for last clone used, and two should initially contain each of the values 1, 2, 3, and 4. In this way, different clones are called initially.

The second step to achieving better load balancing adds an additional field to the FCT, and subdivides case 2 call instructions into calls to initialize operations and other calls. A last clone used field for each type of call is stored in FCTs. In this way, calls to variable initialize operations are more evenly distributed; their distribution is not disturbed by calls to other operations. This helps to achieve distribution of the local variables of an operation. These two optimizations will not be used in the remainder of the dissertation to simplify the discussion.

FCTs may be modified to restrict the clones that can be used. This is necessary when calling recursive operations. The physical call graph showing calls among clones of a facility providing a recursive operation must be a tree, with each leaf calling only its own copy of code. The tree structure avoids cycles that might occur if any copy of the code could be called. To restrict the clones that can be used from a PE the number of clones field of the FCT on that PE is set to be less than the true number of clones. For example, if only two clones of the sequence facility are to be called
from PE4 then the number of clones field of the FCT on PE4 for sequence is set to 2. The first two clone entries in the FCT will have to correspond to the clones to be used. An important assumption is that case 2 calls are used for calling all operations of facilities exporting recursive operations; thus, primitive modules can not provide recursive operations.

Instructions Used in Calling Operations

There are several advantages to dividing the CALL into more primitive instructions. Cycle time is reduced by shortening the execution time of most complex instruction. The need for pipeline stalls when CALL executes is eliminated. The number of bits available for each field of the CALL instruction is increased. If the CALL instruction is not broken into simpler instructions, it is necessary to have additional logic on the chip to quickly perform the multitude of tasks it requires. The tradeoffs involved between the opposing approaches are deferred until a later date when a chip is designed. In this section a set of primitive instructions that implement CALL are presented.

The NEW_FCR instruction loads a value into the facility clone register (FCR). The value loaded is the address of the FCT of the facility providing the called operation. The instruction has one argument, fac; it is added to the value in the FR to obtain the address in FM of the value to be loaded into the FCR.

The FCR points to FCT of a facility providing an operation about to be called. The FCR is only used in setting up calls, and need not be saved on calls or restored on returns.
The NEW_CLONE instruction loads a value into the clone register (CR) and saves the old value of the CR. It loads the number of the clone providing an operation about to be called. Case 1 calls provide an operand to the NEW_CLONE instruction. The operand specifies the offset from the top of the LDS of the parameter that is used to determine the clone to utilize. Case 2 calls use the first two fields of the FCT to obtain the clone number.

The CR contains the number of the clone that provides an operation about to be called. Its value is used by push and arithmetic instructions to set the clone number of data words, thus it is saved and restored on calls and returns.

The NEW_FR instruction loads the FR with the address of the RFR for the facility providing an operation about to be called. It also saves the old value of the FR. The new value of the FR is read from FM at the address computed by adding the FCR and the CR.

The CALL operation loads the PC with the address of the called operation. That address is taken from an RFR in FM (possibly on another PE) at the location specified by the sum of the value in FR and the operand of the instruction.

Following the return of a call, the RESTORE_FR and RESTORE_CR instructions are used to restore the old values of the FR and the CR, respectively.

To summarize, the code generated for a call is:

```
NEW_FCR fac
NEW_CLONE [param] ;param appears in case 1 calls
NEW_FR
```
Implementation of Data Movement Instructions

Data words, the entries on the LDS and in IM, are tagged with clone numbers. These numbers are used to decide which clone of an operation to call. Instructions that affect clone numbers of data words are discussed in this section.

Clone numbers of data words are set when variables are initialized. For this purpose, ARC provides the SET_CLONE_NUM instruction, which sets the clone number of the item on top of LDS to the value contained in the CR. This instruction is needed because when storage is allocated for a variable, its clone number is (incorrectly) set to be that of the memory allocator module. To repair this, the SET_CLONE_NUM instruction is used.

Clone numbers of data are also altered by data movement and arithmetic instructions. The push, pop and DTS instructions set the clone number of the destination word to the clone number of the source word. The push immediate, ADD, and SUB instructions set the clone number of the pushed item to the value of the CR.

Data movement instructions also affect the contents of the parameter address stack (PAS). Recall from chapter 2 that the PAS is used in the automatic restoration of parameters after calls return to permit continued execution after remote calls are
made. The PAS contains the source addresses of items pushed onto the LDS.

Every push instruction causes the address of the pushed item to be placed on the PAS. Execution of a pop instruction causes the removal of the top item on the PAS. Push immediate, ADD, SUB and DTS instructions cause a null address to be pushed onto the PAS, so that the automatic removal of such an item from the LDS simply discards its value.

Actions Taken When a Call Is Made

When a call is made to a remote operation (one that is stored in the instruction memory of another PE) a call packet is constructed and sent to the other PE. Following construction of the packet, the context of the caller is restored and execution is continued.

Call packets contain all information required for another PE to execute an operation. The format of a call packet is shown in figure 50. The number of the PE providing the called operation is in the first field of the packet; this value is taken from the high-order bits of the FR (see figure 51). The second, fourth, and fifth fields contain the values to be loaded into the registers of the called PE; they are copied from the registers of the caller. The third field specifies the offset in the callee’s RFR of the code pointer for the called operation; its value is obtained from the operand field of the instruction register. The sixth field of a packet specifies the number of parameters contained in the packet; this value is taken from the top entry on the PAS. Following this are the parameters. Each parameter entry consists of an address
<table>
<thead>
<tr>
<th>called PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
</tr>
<tr>
<td>opr</td>
</tr>
<tr>
<td>FDR</td>
</tr>
<tr>
<td>CR</td>
</tr>
<tr>
<td>#params</td>
</tr>
<tr>
<td>param-0</td>
</tr>
<tr>
<td>param-1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>param-n</td>
</tr>
<tr>
<td>calling PE</td>
</tr>
</tbody>
</table>

Figure 50: The Contents of a Call Packet.
Figure 51: Construction of a Call Packet.
followed by a value; the address is taken from the PAS and the value is taken from
the LDS. The last field of a call packet is the number of the PE making the call; it
is loaded from a register dedicated to holding that value. (The value stored in this
register is also used for recognizing incoming calls.) The PE number of the caller is
used to return the result of a call.

Following the construction and transmission of a call packet, a PE restores the
context of the caller and continues execution until a busy datum is accessed.

Actions Taken When a Remote Call Is Received

Upon accepting a call packet a PE disassembles it and places its contents into the
correct registers and memory locations. This is illustrated in figure 52. The FR,
FDR and CR are loaded with values from the packet. The PC is loaded with the
value in FM at the address obtained by adding the value of FR to the opr field of
the packet. The values of the parameters are placed on the LDS and the addresses of
the parameters are placed on the PAS. The parameter count is also placed on top of
the parameter addresses on the PAS. The number of the calling PE is saved on top
of the PAS.

Actions Taken When a Call Returns

When a remote operation returns, a return packet must be constructed and trans­
mitted to the calling PE. The format of a return packet is shown in figure 53. The
number of the calling PE is contained in the packet so that the intended recipient
will know that it is a packet addressed to itself; this value is taken from the PAS (see
Figure 52: Use of a Call Packet.
Figure 53: Contents of a Return Packet.

calling PE
#params
param-0
param-1
...
param-n
flag use bit
control flag
Figure 54: Construction of a Return Packet.
The parameter count and the addresses of the parameters are taken from the PAS. The values of the parameters are taken from the LDS. A bit in the return instruction determines whether the value of the control flag is to be used or ignored by the calling PE; the value of this bit is placed in the packet. The control flag's value is also placed in the return packet.

**Actions Taken When a Return Packet Is Received**

Upon receipt of a return packet, a PE disassembles it and distributes its contents to different parts of the processor. Figure 55 shows the dissemination of the contents of a call packet. The values of parameters are placed in the appropriate locations on the LDS and in IM; the addresses of the parameters are contained in the packet. Depending on the value of the call type field of the packet, the value of the control flag is or is not updated with the value contained in the packet.

**The Operation Processor**

The operation processor (OP) constructs, disassembles, and transmits packets. To maximize performance, the OP operates in parallel with the instruction execution unit.

Upon detection of a remote call from its PE's execution unit, the OP begins construction of a call packet and the instruction execution unit continues as if the call had already returned. A call is detected to be remote when the PE-id field of the FR is found to differ from the value of the register MYPE.

Access to the components of the PE used in the construction of the call packet
Figure 55: Use of a Return Packet.
are synchronized because the OP and the instruction execution unit can both access them. This is achieved by locking them until the OP has constructed the call packet. When a component is locked, only the OP can access it. The components containing lock bits are the LDS, CR, FDR, FR and PAS. Each is unlocked when its contents are placed in the call packet. The FR and CR are unlocked first, since the two instructions following CALL instructions require them. A lock may be associated with each word in the LDS and the PAS or with each unit as a whole. The former case permits more parallelism between the instruction processor and the OP, but complicates the PE; an implementor should consider these tradeoffs.

4.4 Summary

The parallel execution of RESOLVE programs was considered in this chapter. A model of parallel execution was developed. Software support for parallel execution was addressed with a mapping algorithm and a run-time system. Hardware support was achieved with a distributed memory implementation of ARC.
CHAPTER V

CONCLUSIONS

The dissertation has considered many aspects of the efficient implementation of RESOLVE. The contributions are summarized here. Areas requiring future research are also identified.

5.1 Contributions

Software reuse increases the productivity of software engineers. However, the benefits of reuse can be offset by poor execution-time performance of the software products. To avoid this pitfall the potential inefficiencies of a class of reusable software components have been identified and addressed. Architectural support for efficient execution of RESOLVE programs has been considered. In addition, it has been shown how the efficiency of RESOLVE programs can be improved through parallel execution.

A virtual machine architecture (ARC) suited to all of RESOLVE was designed using RISC principles. Previously, RISC architectures have been designed to accommodate all programs written in a particular language. A different approach has been taken here—an architecture suited to programs that are not only written in a particular language, but that are also designed according to a specific set of guidelines. Novel instructions have been developed to support code sharing among instances,
control functions, and variable initialization and finalization.

The virtual machine has been formally specified in RESOLVE. The specification does not describe a particular implementation. Using the specification, code can be generated without knowledge of how the architecture is realized, and formal verification of a RESOLVE compiler is (in principle) possible. To my knowledge, no other virtual machine has been specified in this way, i.e., as a software component.

A model of parallel execution suitable for RESOLVE reusable components also has been developed. The model not only supports faster execution of RESOLVE programs, but encourages the development of layered software by increasing parallelism in correspondence to increases in layering. A distributed memory implementation of the virtual machine has been designed to support this model of parallel execution. The implementation supports automatic parameter restoration and uses a simple technique for the automatic synchronization of data accesses. While some of the ideas here have been proposed for other architectures (e.g., dataflow machines), they have not been combined in this way before.

Several other new features have been used in the distributed memory implementation. For example, the simple hardware-supported techniques for achieving dynamic scheduling and dynamic data distribution, and for associating variables with facility clones, are new.

This research is potentially useful with programming languages other than RESOLVE, especially if programmers adhere to the style of programming described here. Programs written in Ada and C++, for example, can benefit from the implementation
techniques described here.

5.2 Future Research

This research has illuminated an abundance of interesting problems that remain to be investigated. A sampling of them is presented here.

A large set of applications should be developed and profiled on sequential and parallel implementations of ARC. The knowledge acquired will guide fine tuning of the instruction set of the virtual machine and the design of the parallel architecture. The profiling can be performed by implementing a simulator, perhaps on an existing parallel processor.

Chips for sequential and parallel implementations of the virtual machine eventually should be designed and built. Such implementations will permit more detailed assessment of the processors, since the limits of on-chip resources will become manifest.

A related task is the exploration of a memory hierarchy. Software and hardware support for exploiting limited on-chip memory, on-chip and off-chip cache, and off-chip main memory need to be considered.

Frequent calls to small operations will result in heavy use of an interprocessor network. An appropriate network needs to be designed and analyzed. Related work in the J-machine project [DCC+87] can be utilized.

The use of formal specifications of modules to automatically parallelize programs needs to be explored. Assertions about the behavior of operations and loops have been shown to be useful to vectorizing compilers [PW86], but little has been done
with assertions in the context of a language like RESOLVE.

A mapping algorithm that increases execution speed of programs is also needed. Factors to be taken into account by such a mapping algorithm include computation time and memory requirements of software components, and interprocessor communication costs.

To facilitate the development of reusable software, it would be useful to create a development environment for RESOLVE programmers. This would include a symbolic debugger, monitoring capabilities, a parallel debugger, a visually interactive program for mapping components to PEs, and a testing assistant.
Appendix A

Virtual machine specification

conceptualization Virtual_Machine_Template

parameters

facility Integer_Facility is Integer_Template
@renaming
  Integer_Facility.int as int
  Integer_Facility.integer as integer
end renaming

facility Fac_Mem_Contents_Facility is Integer_Template
renaming
  Fac_Mem_Contents_Facility.int as fac_mem_contents
  Fac_Mem_Contents_Facility.int as fac_mem_addr
  Fac_Mem_Contents_Facility.int as instr_mem_addr
end renaming

facility Indxd_Mem_Contents_Facility is Integer_Template
renaming
  Indxd_Mem_Contents_Facility.int as
    indxd_mem_contents
  Indxd_Mem_Contents_Facility.int as indxd_mem_addr
  Indxd_Mem_Contents_Facility.int as
    local_data_contents
end renaming

facility Opcode_Facility is Integer_Template
renaming
  Opcode_Facility.int as opcode_type
end renaming

facility Boolean_Facility is Boolean_Template
renaming
  Boolean_Facility.bool as bool
end renaming
end parameters

auxiliary

math facilities

Booleans is Two_Valued_Boolean_Algebra_Template
  renaming
    Booleans.boolean as boolean
  end renaming

Integer.Strings is String_Theory_Template (integer)
  renaming
    Integer.Strings.string as integer_string
  end renaming

tuple_2_theory is
tuple_2_theory_template(integer, integer)
  renaming
    tuple_2_theory.tuple as param_addr
    tuple_2_theory.Projection1 as addr_type
    tuple_2_theory.Projection2 as addr
  end renaming

Param.Strings is String_Theory_Template (param_addr)
  renaming
    Param.Strings.string as param_addr_string
  end renaming

Integer_To_Integer_Function is
Function_Theory_Template (integer, integer)
  renaming
    Integer_To_Integer_Function.function as integer_to_integer
  end renaming

tuple_3_theory is
tuple_3_theory_template(integer, integer, integer)
  renaming
    tuple_3_theory.tuple as instruction_model
    tuple_3_theory.Projection1 as opcode
    tuple_3_theory.Projection2 as base
    tuple_3_theory.Projection3 as offset
  end renaming

Integer_To_Instruction_Function is
Function_Theory_Template (integer, instruction_model)
  renaming
    Integer_To_Instruction_Function.function as integer_to_instruction
  end renaming
tuple_12_theory is
tuple_12_theory_template{
    integer_to_instruction, /* instr. mem */
    integer_to_integer, /* fac. mem */
    integer_string, /* return stack */
    integer_string, /* local stack */
    integer_to_integer, /* indexed mem */
    integer, /* facility reg */
    integer, /* fac. data reg */
    integer, /* PC */
    instruction_model, /* instr. reg */
    boolean, /* control flag */
    param_addr_string, /* param addr stack */
    integer) /* param count */

renaming
    tuple_12_theory.tuple as processor_model
    tuple_12_theory.Projection1 as instruction_memory
    tuple_12_theory.Projection2 as facility_memory
    tuple_12_theory.Projection3 as return_stack
    tuple_12_theory.Projection4 as local_stack
    tuple_12_theory.Projection5 as indexed_memory
    tuple_12_theory.Projection6 as facility_register
    tuple_12_theory.Projection7 as facility_data_register
    tuple_12_theory.Projection8 as program_counter
    tuple_12_theory.Projection9 as instruction_register
    tuple_12_theory.Projection10 as control_flag
    tuple_12_theory.Projection11 as param_addr_stack
end math facilities

math constants

num_instructions : integer
NOP : integer
WAIT : integer
BRANCH : integer
BRFALSE : integer
CALL : integer
RETURN : integer
RETURN_TRUE : integer
RETURN_FALSE : integer
PUSH : integer
PUSHL : integer
POP : integer
POPL : integer
MAX_ALLOWED : integer
MIN_ALLOWED : integer
ACCESS : integer
ADD : integer
SUB : integer
CLRZ : integer
CLR : integer
NEW_FR : integer
RESTORE_FR : integer
NEW_FDR : integer
RESTORE_FDR : integer
CALL_TI : integer
RETURN_TI : integer
CALL_TF : integer
RETURN_TF : integer
BRTRUE : integer
PUSHFD : integer
POPF : integer
CALL_CTRL : integer
DTS : integer
PUSHFDP : integer
PUSHI : integer
PUSHI_O : integer
POPI : integer
POPI_O : integer
indexed_mem_word_size : integer
max_int : integer
min_int : integer
indexed_mem_size : integer
parameter_stack_size : integer
local_stack_size : integer
instruction_mem_size : integer
return_stack_size : integer
facility_mem_size : integer
LDS : integer
IM : integer
NULL : integer
COUNTER : integer

end math constants

constraint
"min_int \leq 0 < max_int and
min_int \geq -(\text{indexed_mem_word_size}) and
max_int \leq \text{indexed_mem_word_size}"

end auxiliary
interface

type instruction is modeled by instruction_model
exemplar instr
Initially "opcode(instr) = 0 and base(instr) = 0 and
   offset(instr) = 0"
end instruction

type processor is modeled by processor_model
exemplar p
Initially
   "∀ addr: integer,
     (instruction_model.init(instruction_memory(p)(addr)) and
      facility_memory(p)(addr) = 0 and
      indexed_memory(p)(addr) = 0) and
      return_stack(p) = ∅ and
      local_stack(p) = ∅ and
      facility_register(p) = 0 and
      facility_data_register(p) = 0 and
      program_counter(p) = 0 and
      instruction_model.init(instruction_register(p)) and
      control_flag(p) and
      param_addr_stack(p) = ∅ and
      param_count(p) = 0 and
end processor

procedure instr_mem_load
parameters
   alters p: processor
   preserves addr: instr_mem_addr
   preserves instr: instruction
end parameters
ensures "§(p, {instruction_memory}) and
   Δ(instruction_memory(p), (addr)) and
   instruction_memory(p)(addr) = instr"
end instr_mem_load
procedure facility_mem_load
parameters
    alters p: processor
    preserves addr: fac_mem_addr
    preserves base_or_proc: fac_mem_contents
end parameters
ensures "δ(p, {facility_memory}) and
    Δ(facility_memory(p), {addr}) and
    facility_memory(p)(addr) = base_or_proc"
end facility_mem_load

procedure indexed_mem_load
parameters
    alters p: processor
    preserves addr: indxd_mem_addr
    preserves value: indxd_mem_contents
end parameters
ensures "δ(p, {indexed_memory}) and
    Δ(indexed_memory(p), {addr}) and
    indexed_memory(p)(addr) = value"
end indexed_mem_load

procedure load_registers
parameters
    alters p: processor
    preserves facility: fac_mem_addr
    preserves facility_data: fac_mem_addr
    preserves next_instr_addr: instr_mem_addr
    preserves this_instr: instruction
    preserves control: bool
end parameters
ensures "δ(p, {facility_register, program_counter,
    instruction_register, control_flag}) and
    facility_register(p) = facility and
    facility_data_register(p) = facility_data and
    program_counter(p) = next_instr_addr and
    instruction_register(p) = this_instr and
    control_flag(p) = control"
end load_registers
function instruction_mem_examine returns instr: instruction
  parameters
    preserves p: processor
    preserves addr: instr_mem_addr
  end parameters
  ensures "instr = instruction_memory(p)(addr)"
end instruction_mem_examine

function facility_mem_examine returns base_or_proc: fac_mem_contents
  parameters
    preserves p: processor
    preserves addr: fac_mem_addr
  end parameters
  ensures "base_or_proc = facility_memory(p)(addr)"
end facility_mem_examine

function indexed_mem_examine returns contents: indxd_mem_contents
  parameters
    preserves p: processor
    preserves addr: indxd_mem_addr
  end parameters
  ensures "contents = facility_memory(p)(addr)"
end indexed_mem_examine

procedure examine_registers
  parameters
    preserves p: processor
    produces facility: fac_mem_addr
    produces facility_data: fac_mem_addr
    produces next_instr_addr: instr_mem_addr
    produces this_instr: instruction
    produces control: bool
  end parameters
  ensures "facility = facility_register(p) and facility_data = facility_data_register(p) and next_instr_addr = program_counter(p) and this_instr = instruction_register(p) and control = control_flag(p)"
end examine_registers
function local_stack_remaining returns size: int
  parameters
    preserves p: processor
  end parameters
  ensures "size = local_stack_size - |local_data_stack(p)|"
end local_stack_remaining

control valid_opcode
  parameters
    preserves opcode: opcode_type
  end parameters
  ensures valid_opcode iff "opcode≥0 and opcode<num_instructions"
end valid_opcode

procedure reset
  parameters
    alters p: processor
  end parameters
  ensures "δ(p, [indexed_memory, return_stack, local_stack, 
    facility_register, program_counter, 
    instruction_register, control_flag, param_addr_stack, 
    param_count]) and 
  ∆(indexed_memory(p), [addr | addr integer]) and 
  ∀addr: integer (indexed_memory(p)(addr) = 0) and 
  return_stack(p) = Λ and 
  local_stack(p) = Λ and 
  facility_register(p) = 0 and 
  facility_data_register(p) = 0 and 
  program_counter(p) = 0 and 
  instruction_model.init(instruction_register(p)) and 
  control_flag(p) and 
  param_addr_stack(p) = Λ and 
  param_count(p) = 0"
end reset
procedure do_instruction
params
    alters p: processor
end params
requires &

(\text{operation}(\text{instruction\_register}(p) = \text{BRANCH}) \Rightarrow 
    0 \leq \text{program\_counter}(p) + 
    \text{offset}(\text{instruction\_register}(p)) 
    < \text{instruction\_mem\_size} \text{ and }
    \text{program\_counter}(p) = \text{instruction\_mem\_size - 1} \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{BRFALSE}) \Rightarrow 
    \text{not control\_flag}(p) \Rightarrow 
    0 \leq \text{program\_counter}(p) + 
    \text{offset}(\text{instruction\_register}(p)) 
    < \text{instruction\_mem\_size} \text{ and }
    \text{program\_counter}(p) = \text{instruction\_mem\_size - 1} \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{CALL}) \Rightarrow 
    |\text{return\_stack}(p)| < \text{return\_stack\_size} \text{ and }
    \text{program\_counter}(p) = \text{instruction\_mem\_size - 1} \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{RETURN\_TRUE}) \Rightarrow 
    |\text{return\_stack}(p)| \geq 2 \text{ and }
    \text{program\_counter}(p) = \text{instruction\_mem\_size - 1} \text{ and }
    \text{return\_stack}(p) \neq \Lambda \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{RETURN\_FALSE}) \Rightarrow 
    |\text{return\_stack}(p)| \geq 2 \text{ and }
    \text{program\_counter}(p) = \text{instruction\_mem\_size - 1} \text{ and }
    \text{return\_stack}(p) \neq \Lambda \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{PUSH}) \Rightarrow 
    |\text{local\_stack}(p)| < \text{local\_stack\_size} \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{PUSHL}) \Rightarrow 
    0 \leq \text{offset}(\text{instruction\_register}(p)) < |\text{local\_stack}(p)| \text{ and }
    |\text{local\_stack}(p)| < \text{local\_stack\_size} \text{ and}

(\text{operation}(\text{instruction\_register}(p)) = \text{POP}) \Rightarrow 
    |\text{local\_stack}(p)| > 0 \text{ and}
((operation(instruction_register(p)) = POPL) ⇒
offset(instruction_register(p)) < \|local_stack(p)\| and
\|local_stack(p)\| ≠ \Lambda \) and

((operation(instruction_register(p)) = MAX_ALLOWED) ⇒
\|local_stack(p)\| < local_stack_size and

((operation(instruction_register(p)) = MIN_ALLOWED) ⇒
\|local_stack(p)\| < local_stack_size and

((operation(instruction_register(p)) = ACCESS) ⇒
\|local_stack(p)\| ≥ 3 ) and

((operation(instruction_register(p)) = ADD) ⇒
\exists \ i, j: integer, \ 3 s: string of integer
\|local_stack(p)\| = s • i • j and
\|local_stack(p)\| < local_stack_size ) and

((operation(instruction_register(p)) = SUB) ⇒
\exists \ i, j: integer, \ 3 s: string of integer
\|local_stack(p)\| = s • i • j and
\|local_stack(p)\| < local_stack_size ) and

((operation(instruction_register(p)) = CLRZ) ⇒
\|local_stack(p)\| > 0 ) and

((operation(instruction_register(p)) = CLRN) ⇒
\|local_stack(p)\| > 0 )
ensures

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{NO-OP} \; \text{or} \\
& (\text{operation}(\text{instruction_register}(#p)) = \text{WAIT}) \Rightarrow \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
& \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
& \text{instruction_register}(p) = \\
& \text{instruction_memory}(p)(\text{program_counter}(#p)) ) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{BRANCH}) \Rightarrow \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
& \text{program_counter}(p) = \text{program_counter}(#p) + \\
& \text{offset}(\text{instruction_register}(#p)) \text{ and} \\
& \text{instruction_register}(p) = \\
& \text{instruction_memory}(p)(\text{program_counter}(#p)) ) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{BRANCH_FALSE} \text{ and} \\
& \text{not control_flag(#p)}) \Rightarrow /* (\text{BRANCH TAKEN}) */ \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
& \text{program_counter}(p) = \text{program_counter}(#p) + \\
& \text{offset}(\text{instruction_register}(#p)) \text{ and} \\
& \text{instruction_register}(p) = \\
& \text{instruction_memory}(p)(\text{program_counter}(#p)) ) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{BRANCH_FALSE} \text{ and} \\
& \text{control_flag(#p)}) \Rightarrow /* (\text{BRANCH NOT TAKEN}) */ \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
& \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
& \text{instruction_register}(p) = \\
& \text{instruction_memory}(p)(\text{program_counter}(#p)) ) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{BRANCH_TRUE} \text{ and} \\
& \text{control_flag(#p)}) \Rightarrow /* (\text{BRANCH TAKEN}) */ \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
& \text{program_counter}(p) = \text{program_counter}(#p) + \\
& \text{offset}(\text{instruction_register}(#p)) \text{ and} \\
& \text{instruction_register}(p) = \\
& \text{instruction_memory}(p)(\text{program_counter}(#p)) ) \text{ and} \\
\end{align*}
\]

\[
\begin{align*}
( & (\text{operation}(\text{instruction_register}(#p)) = \text{BRANCH_TRUE} \text{ and} \\
& \text{not control_flag(#p)}) \Rightarrow /* (\text{BRANCH NOT TAKEN}) */ \\
& \delta(p, \{\text{program_counter}, \text{instruction_register}\}) \text{ and} \\
\end{align*}
\]
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) and

{operation(instruction_register(#p)) = DTS} ⇒
δ(p, {local_stack, program_counter, instruction_register,
    param_count, param_addr_stack}) and
∃ i:integer, ∃ s1: integer_string
    (local_stack(#p) = s1 • i and
    local_stack(p) = s1 • i • i and
    param_addr_stack(p) =
        param_addr_stack(#p) • (NULL, 0) ) and
param_count(p) = param_count(#p) + 1 and
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) ) and

{operation(instruction_register(#p)) = PUSH} ⇒
δ(p, {local_stack, program_counter, instruction_register,
    param_count, param_addr_stack}) and
local_stack(p) = local_stack(#p) •
    offset(instruction_register(#p))
and program_counter(p) = program_counter(#p) + 1 and
param_count(p) = param_count(#p) + 1 and
param_addr_stack(p) = param_addr_stack(#p) • (NULL, 0)
and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) ) and

{operation(instruction_register(#p)) = PUSHL} ⇒
δ(p, {local_stack, program_counter, instruction_register,
    param_count, param_addr_stack}) and
∃ i,j:integer, ∃ s1,s2: integer_string
    (local_stack(#p) = s1 • i • s2 and
    [s2] = offset(instruction_register(#p)) and
    local_stack(p) = s1 • j • s2 • i and
    param_addr_stack(p) =
        param_addr_stack(#p) • (LDS, [s1]) ) and
param_count(p) = param_count(#p) + 1 and
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) ) and
\((\text{operation}(\text{instruction}\_\text{register}(\#p)) = \text{PUSHFD}) \Rightarrow\)
\[\delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_count, param\_addr\_stack, indexed\_memory}\}) \text{ and} \]
\[\text{local\_stack}(p) = \text{local\_stack}(\#p) \times \]
\[\begin{array}{c}
\text{indexed\_memory}(\#p)(\text{facility\_data\_register}(p) + \\
\text{offset}(\text{instruction}\_\text{register}(\#p)) \text{ and} \\
\Delta(\text{indexed\_memory}(p), \{\text{facility\_data\_register}(p) + \\
\text{offset}(\text{instruction}\_\text{register}(\#p))\}) \text{ and} \\
\text{indexed\_memory}(p)(\text{facility\_data\_register}(p) + \\
\text{offset}(\text{instruction}\_\text{register}(\#p)) = 0 \text{ and} \\
\text{param\_count}(p) = \text{param\_count}(\#p) + 1 \text{ and} \\
\text{param\_addr\_stack}(p) = \text{param\_addr\_stack}(\#p) \times \\
(\text{IM, facility\_data\_register}(p) + \\
\text{offset}(\text{instruction}\_\text{register}(\#p)) \text{ and} \\
\text{program\_counter}(p) = \text{program\_counter}(\#p) + 1 \text{ and} \\
\text{instruction\_register}(p) = \\
\text{instruction\_memory}(p)(\text{program\_counter}(\#p)) \text{ and} \\
\end{array}\]

\((\text{operation}(\text{instruction}\_\text{register}(\#p)) = \text{PUSHI}) \Rightarrow\)
\[\delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_count, param\_addr\_stack, indexed\_memory}\}) \text{ and} \]
\[\exists i, j : \text{integer}, \exists s1 : \text{integer\_string} \times \]
\[\begin{array}{c}
(\text{local\_stack}(\#p) = s1 \times i \times j \text{ and} \\
\text{local\_stack}(p) = \\
\text{local\_stack}(\#p) \times \text{indexed\_memory}(\#p)(i + j) \text{ and} \\
\Delta(\text{indexed\_memory}(p), \{i + j\}) \text{ and} \\
\text{indexed\_memory}(p)(i + j) = 0 \text{ and} \\
\text{param\_addr\_stack}(p) = \\
\text{param\_addr\_stack}(\#p) \times (\text{IM, } i + j) \text{ and} \\
\text{param\_count}(p) = \text{param\_count}(\#p) + 1 \text{ and} \\
\text{program\_counter}(p) = \text{program\_counter}(\#p) + 1 \text{ and} \\
\text{instruction\_register}(p) = \\
\text{instruction\_memory}(p)(\text{program\_counter}(\#p)) \text{ and} \\
\end{array}\]

\((\text{operation}(\text{instruction}\_\text{register}(\#p)) = \text{PUSHI}_0) \Rightarrow\)
\[\delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_count, param\_addr\_stack, indexed\_memory}\}) \text{ and} \]
\[\exists i, j : \text{integer}, \exists s1 : \text{integer\_string} \times \]
\[\begin{array}{c}
(\text{local\_stack}(\#p) = s1 \times i \times j \text{ and} \\
j = \text{offset}(\text{instruction}\_\text{register}(\#p)) \text{ and} \\
\text{local\_stack}(p) = \\
\text{local\_stack}(\#p) \times \text{indexed\_memory}(\#p)(i + j) \text{ and} \\
\Delta(\text{indexed\_memory}(p), \{i + j\}) \text{ and} \\
\end{array}\]
indexed_memory(p)(i + j) = 0 and
param_addr_stack(p) =
param_addr_stack(#p) • (IM, i + j) and
param_count(p) = param_count(#p) + 1 and
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
instruction_memory(p)(program_counter(#p)) and

((operation(instruction_register(#p)) = PUSHFDP) \Rightarrow
\delta(p, \{local_stack, program_counter, instruction_register, param_count, param_addr_stack\}) and
local_stack(p) = local_stack(#p) •
indexed_memory(p)(facility_data_register(p) +
offset(instruction_register(#p)) and
param_count(p) = param_count(#p) + 1 and
param_addr_stack(p) = param_addr_stack(#p) •
(IM, facility_data_register(p) +
offset(instruction_register(#p)) and
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
instruction_memory(p)(program_counter(#p)) and

((operation(instruction_register(#p)) = POP) \Rightarrow
\delta(p, \{local_stack, program_counter, instruction_register, param_count, param_addr_stack\}) and
\exists s: integer \rightarrow
(local_stack(p) = local_stack(#p) • s and
offset(instruction_register(p)) = |s| and

((param_count(#p) - |s|) \geq 0) \Rightarrow
param_count(p) = param_count(#p) - |s| and
\exists t: param_addr_string
(param_addr_stack(#p) =
(param_addr_stack(p) • t) and

((param_count(#p) - |s|) < 0) \Rightarrow
param_count(p) = 0 and
param_addr_stack(p) = \Lambda \) and

program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
instruction_memory(p)(program_counter(#p)) and
((\text{operation}(\text{instruction}\_\text{register}(#p)) = \text{POPL}) \Rightarrow
\delta(p, \{local\_stack, program\_counter, instruction\_register, 
param\_count, param\_addr\_stack\}) \text{ and }
\exists i,j: \text{integer, } \exists s1,s2: \text{integer}\_\text{string}
\begin{align*}
&(local\_stack(#p) = s1 \cdot i \cdot s2 \cdot j \text{ and } \\
&local\_stack(p) = s1 \cdot j \cdot s2 \text{ and } \\
&[s2] + 1 = \text{offset}(instruction\_register(#p)) \text{ and } \\
&(s2) > 0) \Rightarrow
\exists i: \text{param}\_addr \text{ (param\_addr\_stack(#p) = } \\
\text{param\_addr\_stack(p) \cdot i) \text{ and } } \\
\text{program\_counter(p) = program\_counter(#p) + 1 and } \\
\text{instruction\_register(p) = } \\
\text{instruction\_memory(p)(program\_counter(#p)) } \text{ and }
\end{align*}

((\text{operation}(\text{instruction}\_\text{register}(#p)) = \text{POPFD}) \Rightarrow
\delta(p, \{local\_stack, program\_counter, instruction\_register, 
param\_count, param\_addr\_stack, indexed\_memory\}) \text{ and }
\exists i: \text{integer, } \exists s1: \text{integer}\_\text{string}
\begin{align*}
&(local\_stack(#p) = s1 \cdot i \text{ and } \\
&local\_stack(p) = s1 \text{ and } \\
&A(indexed\_memory(p), \{facility\_data\_register(p) + 
\text{offset}(instruction\_register(#p))\} ) \text{ and } \\
&indexed\_memory(p)(facility\_data\_register(p) + 
\text{offset}(instruction\_register(#p)) = i) \text{ and } \\
&(s2) > 0) \Rightarrow
\exists i: \text{param}\_addr \text{ (param\_addr\_stack(#p) = } \\
\text{param\_addr\_stack(p) \cdot i) \text{ and } } \\
\text{program\_counter(p) = program\_counter(#p) + 1 and } \\
\text{instruction\_register(p) = } \\
\text{instruction\_memory(p)(program\_counter(#p)) } \text{ and }
\end{align*}

((\text{operation}(\text{instruction}\_\text{register}(#p)) = \text{POPI}) \Rightarrow
\delta(p, \{local\_stack, program\_counter, instruction\_register, 
param\_count, param\_addr\_stack, indexed\_memory\}) \text{ and }
\exists i, j, k: \text{integer, } \exists s1: \text{integer}\_\text{string}
\begin{align*}
&(local\_stack(#p) = s1 \cdot i \cdot j \cdot k \text{ and } \\
&local\_stack(p) = s1 \cdot i \cdot j \text{ and }
\end{align*}
\[ \Delta(\text{indexed\_memory}(p), \{i + j\}) \text{ and } \]
\[\text{indexed\_memory}(p)(i + j) = k \text{ and } \]

\[(\text{param\_count}(#p) > 0) \Rightarrow \]
\[\text{param\_count}(p) = \text{param\_count}(#p) - 1 \text{ and } \]
\[\exists i: \text{param\_addr} (\text{param\_addr\_stack}(#p) = \]
\[\text{param\_addr\_stack}(p) + i) \text{ and } \]

\[\text{program\_counter}(p) = \text{program\_counter}(#p) + 1 \text{ and } \]
\[\text{instruction\_register}(p) = \]
\[\text{instruction\_memory}(p)(\text{program\_counter}(#p))) \text{ and } \]

\[(\text{operation}(\text{instruction\_register}(#p)) = \text{POPI} \Rightarrow \]
\[\delta(p, \{\text{local\_stack}, \text{program\_counter}, \text{instruction\_register}, \]
\[\text{param\_count}, \text{param\_addr\_stack}, \text{indexed\_memory}\}) \text{ and } \]
\[\exists i, j, k: \text{integer}, \exists s_1: \text{integer\_string} \]
\[\text{local\_stack}(#p) = s_1 \cdot i \cdot j \text{ and } \]
\[k = \text{offset}(\text{instruction\_register}(#p)) \text{ and } \]
\[\text{local\_stack}(p) = s_1 \cdot i \text{ and } \]
\[\Delta(\text{indexed\_memory}(p), \{i + j\}) \text{ and } \]
\[\text{indexed\_memory}(p)(i + j) = j \text{ and } \]

\[(\text{param\_count}(#p) > 0) \Rightarrow \]
\[\text{param\_count}(p) = \text{param\_count}(#p) - 1 \text{ and } \]
\[\exists i: \text{param\_addr} (\text{param\_addr\_stack}(#p) = \]
\[\text{param\_addr\_stack}(p) + i) \text{ and } \]

\[\text{program\_counter}(p) = \text{program\_counter}(#p) + 1 \text{ and } \]
\[\text{instruction\_register}(p) = \]
\[\text{instruction\_memory}(p)(\text{program\_counter}(#p))) \text{ and } \]

\[(\text{operation}(\text{instruction\_register}(#p)) = \text{MAX\_ALLOWED} \Rightarrow \]
\[\delta(p, \{\text{local\_stack}, \text{program\_counter}, \text{instruction\_register}, \]
\[\text{param\_count}, \text{param\_addr}\}) \text{ and } \]
\[\text{local\_stack}(p) = \]
\[\text{local\_stack}(#p) \cdot \text{max\_int} \text{ and } \]
\[\text{param\_count}(p) = \text{param\_count}(#p) + 1 \]
\[\text{param\_addr\_stack}(p) = \]
\[\text{param\_addr\_stack}(#p) \cdot (\text{NULL}, 0) \]
\[\text{program\_counter}(p) = \text{program\_counter}(#p) + 1 \text{ and } \]
\[\text{instruction\_register}(p) = \]
\[\text{instruction\_memory}(p)(\text{program\_counter}(#p))) \text{ and } \]
\[(\text{operation}(\text{instruction\_register}(\#p)) = \text{MIN\_ALLOWED}) \Rightarrow \delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_count, param\_addr}\}) \text{ and} \]
\[
\begin{align*}
\text{local\_stack}(p) &= \text{local\_stack}(\#p) \cdot \min\_int \text{ and} \\
\text{param\_count}(p) &= \text{param\_count}(\#p) + 1 \\
\text{param\_addr\_stack}(p) &= \text{param\_addr\_stack}(\#p) \cdot (\text{NULL, 0}) \\
\text{program\_counter}(p) &= \text{program\_counter}(\#p) + 1 \text{ and} \\
\text{instruction\_register}(p) &= \\
&\text{instruction\_memory}(p)(\text{program\_counter}(\#p)) \text{ and} \\
\end{align*}
\]

\[(\text{operation}(\text{instruction\_register}(\#p)) = \text{ACCESS}) \Rightarrow \delta(p, \{\text{local\_stack, indexed\_memory, program\_counter, instruction\_register}\}) \text{ and} \]
\[
\exists a, i, x: \text{integer}, \exists s: \text{integer\_string} \\
\begin{align*}
\text{local\_stack}(\#p) &= s \cdot a \cdot i \cdot x \text{ and} \\
\text{indexed\_memory}(p, \{a+i\}) &= \text{indexed\_memory}(p)(a+i) = x \text{ and} \\
\text{local\_stack}(p) &= s \cdot a \cdot i \cdot \text{indexed\_memory}(\#p)(a+i) \\
\end{align*}
\]
\[
\text{and program\_counter}(p) = \text{program\_counter}(\#p) + 1 \text{ and} \\
\text{instruction\_register}(p) = \\
\text{instruction\_memory}(p)(\text{program\_counter}(\#p)) \text{ and} \\
\]

\[(\text{operation}(\text{instruction\_register}(\#p)) = \text{ADD}) \Rightarrow \delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_addr\_stack, param\_count}\}) \text{ and} \]
\[
\exists i, j: \text{integer}, \exists s: \text{integer\_string} \\
\begin{align*}
\text{local\_stack}(\#p) &= s \cdot i \cdot j \text{ and} \\
\text{local\_stack}(p) &= s \cdot i \cdot j \cdot i+j \text{ and} \\
\text{param\_count}(p) &= \text{param\_count}(\#p) + 1 \\
\text{param\_addr\_stack}(p) &= \text{param\_addr\_stack}(\#p) \cdot (\text{NULL, 0}) \\
\text{program\_counter}(p) &= \text{program\_counter}(\#p) + 1 \text{ and} \\
\text{instruction\_register}(p) &= \\
&\text{instruction\_memory}(p)(\text{program\_counter}(\#p)) \text{ and} \\
\end{align*}
\]

\[(\text{operation}(\text{instruction\_register}(\#p)) = \text{SUB}) \Rightarrow \delta(p, \{\text{local\_stack, program\_counter, instruction\_register, param\_addr\_stack, param\_count}\}) \text{ and} \]
\[ \exists i,j: \text{integer}, \exists s: \text{integer_string} \\
\quad (\text{local_stack}(#p) = s \cdot i \cdot j \text{ and} \\
\quad \text{local_stack}(p) = \\
\quad s \cdot i \cdot j \cdot i-j \text{ and} \\
\quad \text{param_count}(p) = \text{param_count}(#p) + 1 \\
\quad \text{param_addr_stack}(p) = \\
\quad \text{param_addr_stack}(#p) = (\text{NULL}, 0) \\
\quad \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
\quad \text{instruction_register}(p) = \\
\quad \text{instruction_memory}(p)(\text{program_counter}(#p)) \text{ and} \\
\quad ((\text{operation}(\text{instruction_register}(#p)) = \text{CLRZ}) \Rightarrow \\
\quad \delta(p, \{\text{control_flag, program_counter,} \\
\quad \text{instruction_register}\}) \text{ and} \\
\quad \exists i: \text{integer}, \exists s: \text{integer_string} \\
\quad (\text{local_stack}(#p) = s \cdot i \text{ and} \\
\quad (i = 0 \text{ and not control_flag}(p)) \text{ or} \\
\quad (i = 0 \text{ and control_flag}(p))) \text{ and} \\
\quad \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
\quad \text{instruction_register}(p) = \\
\quad \text{instruction_memory}(p)(\text{program_counter}(#p)) \text{ and} \\
\quad ((\text{operation}(\text{instruction_register}(#p)) = \text{CLR}) \Rightarrow \\
\quad \delta(p, \{\text{control_flag, program_counter,} \\
\quad \text{instruction_register}\}) \text{ and} \\
\quad \exists i: \text{integer}, \exists s: \text{integer_string} \\
\quad (\text{local_stack}(#p) = s \cdot i \text{ and} \\
\quad (i < 0 \text{ and not control_flag}(p)) \text{ or} \\
\quad (i \geq 0 \text{ and control_flag}(p))) \text{ and} \\
\quad \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
\quad \text{instruction_register}(p) = \\
\quad \text{instruction_memory}(p)(\text{program_counter}(#p)) \text{ and} \\
\quad ((\text{operation}(\text{instruction_register}(#p)) = \text{READ}) \Rightarrow \\
\quad \delta(p, \{\text{local_stack, program_counter,} \\
\quad \text{instruction_register}\}) \text{ and} \\
\quad \exists i: \text{integer} (\text{local_stack}(p) = \text{local_stack}(#p) \cdot i) \text{ and} \\
\quad \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \\
\quad \text{instruction_register}(p) = \\
\quad \text{instruction_memory}(p)(\text{program_counter}(#p)) \text{ and} \\
\quad ((\text{operation}(\text{instruction_register}(#p)) = \text{WRITE}) \Rightarrow \\
\quad \delta(p, \{\text{program_counter, instruction_register}\}) \text{ and} \]
\[ \exists i: \text{integer}, \exists s: \text{integer_string} \]
\[ \quad (\text{local_stack}(p) = s \cdot i \text{ and} \]
\[ \quad \text{output}(i)) \]
\[ \quad \text{program_counter}(p) = \text{program_counter}(#p) + 1 \text{ and} \]
\[ \quad \text{instruction_register}(p) = \]
\[ \quad \mathcal{E} \quad \text{instruction_memory}(p)(\text{program_counter}(#p)) \]
((operation(instruction_register(#p)) = CALL) ⇒
δ(p, {return_stack, program_counter,
    instruction_register, param_addr_stack,
    param_count}) and

return_stack(p) = return_stack(#p) •
                 (program_counter(#p) + 1) and
param_addr_stack(p) = param_addr_stack(#p) •
                         (COUNTER, param_count(#p)) and
param_count(p) = 0 and
program_counter(p) =
    facility_memory(p)((facility_register(#p)) +
                       offset(instruction_register(#p))) and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) ) and

((operation(instruction_register(#p)) = CALL_CTRL) ⇒
δ(p, {return_stack, program_counter,
    instruction_register, param_addr_stack,
    param_count}) and

return_stack(p) = return_stack(#p) •
                 (program_counter(#p) + 1) and
param_addr_stack(p) = param_addr_stack(#p) •
                         (COUNTER, param_count(#p)) and
param_count(p) = 0 and
program_counter(p) =
    facility_memory(p)((facility_register(#p)) +
                       offset(instruction_register(#p))) and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) ) and

((operation(instruction_register(#p)) = CALL_INIT) ⇒
δ(p, {return_stack, program_counter, local_stack,
    instruction_register, param_addr_stack,
    param_count}) and

return_stack(p) = return_stack(#p) •
                 (program_counter(#p) + 1) and
local_stack(p) = local_stack(#p) • 0 and
param_addr_stack(p) = param_addr_stack(#p) •
                       (LDS, |local_stack(p)|) •
                       (COUNTER, param_count(#p) + 1) and
param_count(p) = 0 and
program_counter(p) =  
    facility_memory(p)((facility_register(#p)) +  
    offset(instruction_register(#p))) and

instruction_register(p) =  
    instruction_memory(p)(program_counter(#p)) and

\[ ((\text{operation}(\text{instruction_register(#p)}) = \text{CALL_FIN}) \Rightarrow  \\
\delta(p, \{\text{return_stack, program_counter,}  \\
\text{instruction_register, param_count}\}) \text{ and} \]

return_stack(p) = return_stack(#p) •  
    (program_counter(#p) + 1) and

param_count(p) = 0 and

program_counter(p) =  
    facility_memory(p)((facility_register(#p)) +  
    offset(instruction_register(#p))) and

instruction_register(p) =  
    instruction_memory(p)(program_counter(#p)) and

\[ ((\text{operation}(\text{instruction_register(#p)}) = \text{RETURN_TRUE}) \Rightarrow  \\
\delta(p, \{\text{return_stack, program_counter,}  \\
\text{control_flag, instruction_register, local_stack,}  \\
\text{indexed_memory, param_addr_stack, param_count}\}) \text{ and} \]

and

return_stack(p) = return_stack(#p) • program_counter(p)  
and param_count(p) = 0 and

\[ \exists s_1, s_2: \text{param_addr_string} \exists i: \text{param_addr}  \\
(\text{param_addr_stack(#p)} =  \\
\text{param_addr_stack(p)} • s_1 • i • s_2 \text{ and} \]

|s_2| = param_count(p) and

|s_1| = addr(i) and

\[ \exists s_3, s_4: \text{integer_string}  \\
(\text{local_stack(#p)} = s_3 • s_4 \text{ and} \]

|s_4| = |s_1| and

|local_stack(p)| = |s_3|) and

\[ \forall j: \text{param_addr (member(s_1, j))} \Rightarrow  \\
\exists s_3, s_4: \text{param_addr_string}  \\
(s_1 = s_3 • j • s_4 \text{ and} \]

(addr_type(j) = LDS) \Rightarrow  \\
\exists s_5, s_6, s_7, s_8, s_9: \text{integer_string}  \\
\exists k, l: \text{integer}  \\
(\text{local_stack(#p)} = s_5 • k • s_6 • l • s_7  \\
\text{and} |s_5| = addr(j) \text{ and} \]

|s_7| = |s_4| and
\begin{align*}
\text{local_stack}(p) &= s8 \cdot 1 \cdot s9 \quad \text{and} \\
|s8| &= |s5| \quad \text{and} \\
\text{(addr_type}(j) &= \text{LM}) \Rightarrow \\
\exists s5, s6: \text{integer_string} \\
\exists k: \text{integer} \\
\delta(p) &= s5 \cdot k \cdot s6 \quad \text{and} \\
|s6| &= |s4| \quad \text{and} \\
\Delta(\text{indexed_memory}(p), \{\text{addr}(j)\}) \quad \text{and} \\
\text{indexed_memory}(p)(\text{addr}(j)) &= k \}
\end{align*}

\text{and}

\text{control_flag}(p) \quad \text{and} \\
\text{instruction_register}(p) = \\
\text{instruction_memory}(p)(\text{program_counter}(p)) \quad \text{and} \\
\left(\text{operation}(\text{instruction_register}(p)) = \text{RETURN_FALSE}\right) \Rightarrow \\
\delta(p, \{\text{return_stack}, \text{program_counter}, \\
\text{control_flag}, \text{instruction_register}, \text{local_stack}, \\
\text{indexed_memory}, \text{param_addr_stack}, \text{param_count}\}) \quad \text{and} \\
\text{return_stack}(p) = \text{return_stack}(p) \cdot \text{program_counter}(p) \\
\text{and} \quad \text{param_count}(p) = 0 \quad \text{and} \\
\exists s1, s2: \text{param_addr_string} \quad \exists i: \text{param_addr} \\
\text{(param_addr_stack}(p) = \\
\quad \text{param_addr_stack}(p) \cdot s1 \cdot i \cdot s2 \quad \text{and} \\
|s2| = \text{param_count}(p) \quad \text{and} \\
|s1| = \text{addr}(i) \quad \text{and} \\
\exists s3, s4: \text{integer_string} \\
\text{(local_stack}(p) = s3 \cdot s4 \quad \text{and} \\
|s4| = |s1| \quad \text{and} \\
|\text{local_stack}(p)| = |s3| \quad \text{and} \\
\forall j: \text{param_addr}(\text{member}(s1, j)) \Rightarrow \\
\exists s3, s4: \text{param_addr_string} \\
(s1 = s3 \cdot j \cdot s4 \quad \text{and} \\
\text{(addr_type}(j) = \text{LDS}) \Rightarrow \\
\exists s5, s6, s7, s8, s9: \text{integer_string} \\
\exists k, l: \text{integer} \\
\text{(local_stack}(p) = s5 \cdot k \cdot s6 \cdot l \cdot s7 \\
\quad \text{and} \quad |s5| = |s3| \quad \text{and} \\
|s7| = |s4| \quad \text{and} \\
\text{local_stack}(p) = s8 \cdot l \cdot s9 \quad \text{and} \\
|s8| = |s5| \quad \text{and}
(addr_type(j) = IM) ⇒
3 s5, s6: integer_string
3 k: integer
    (local_stack(#p) = s5 • k • s6 and
    |s6| = |s4| and
    \[\Delta(\text{indexed_memory}(p), \{\text{addr}(j)\}) \land \text{indexed_memory}(p)(\text{addr}(j)) = k)\]
and
not control_flag(p) and
instruction_register(p) =
    instruction_memory(p)(program_counter(#p)) and

((operation(instruction_register(#p)) = RETURN) ⇒
δ(p, \{return_stack, program_counter,
    instruction_register, local_stack,
    indexed_memory, param_addr_stack, param_count\})
and
return_stack(p) = return_stack(#p) • program_counter(p)
and param_count(p) = 0 and
∃ s1, s2: param_addr_string 3 i: param_addr
    (param_addr_stack(#p) =
        param_addr_stack(p) • s1 • i • s2 and
    |s2| = param_count(#p) and
    |s1| = addr(i) and
∃ s3, s4: integer_string
    (local_stack(#p) = s3 • s4 and
    |s4| = |s1| and
    |local_stack(p)| = |s3|) and
∀ j: param_addr (member(s1, j) ⇒
∃ s3, s4: param_addr_string
    (s1 = s3 • j • s4 and
    (addr_type(j) = LDS) ⇒
∃ s5, s6,s7,s8,s9: integer_string
3 k, l: integer
    (local_stack(#p) = s5 • k • s6 • l • s7
    and |s5| = addr(j) and
    |s7| = |s4| and
    local_stack(p) = s8 • l • s9 and
    |s8| = |s5|) and
    (addr_type(j) = IM) ⇒
∃ s5, s6: integer_string
3 k: integer
(local_stack(#p) = s5 • k • s6 and
|s6| = |s4| and
Δ(indexed_memory(p), (addr(j))) and
indexed_memory(p)(addr(j)) = k))

and
instruction_register(p) =
instruction_memory(p)(program_counter(#p)) and

((operation(instruction_register(#p)) = RETURN_INIT) ⇒
δ(p, (return_stack, program_counter,
  instruction_register, local_stack,
  indexed_memory, param_addr_stack, param_count))

and
return_stack(p) = return_stack(#p) • program_counter(p)
and param_count(p) = 0 and
3 s1, s2: param_addr_string 3 i: param_addr
(param_addr_stack(#p) =
  param_addr_stack(p) • s1 • i • s2 and
|s2| = param_count(#p) and
|s1| = addr(i) and
3 s3, s4: integer_string
  (local_stack(#p) = s3 • s4 and
  |s4| = |s1| and
  |local_stack(p)| = |s3| and
∀ j: param_addr (member(s1, j)) ⇒
3 s3, s4: param_addr_string
  (s1 = s3 • i • s4 and
  (addr_type(i) = LDS) ⇒
3 s5, s6, s7, s8, s9: integer_string
3 k, l: integer
  (local_stack(#p) = s5 • k • s6 • l • s7
  and |s5| = addr(l) and
  |s7| = |s4| and
  local_stack(p) = s6 • l • s9 and
  |s8| = |s5| and
  (addr_type(l) = IMM) ⇒
3 s5, s6: integer_string
3 k: integer
  (local_stack(#p) = s5 • k • s6 and
  |s6| = |s4| and
  Δ(indexed_memory(p), (addr(j))) and
  indexed_memory(p)(addr(j)) = k))))}

and

\[
\text{instruction_register}(p) = \\text{instruction_memory}(p)(\text{program_counter}(\#p)) \quad \text{and}
\]

\[
((\text{operation}(\text{instruction_register}(\#p)) = \text{RETURN_FIN}) \Rightarrow
\]

\[
\delta(p, (\text{return_stack}, \text{program_counter}, \text{instruction_register},
& \text{param_addr_stack}, \text{param_count}, \text{local_stack})) \quad \text{and}
\]

\[
3 \quad \text{i: integer} \quad (\text{local_stack}(\#p) = \text{local_stack}(p) \ast i) \quad \text{and}
\]

\[
\text{return_stack}(p) = \text{return_stack}(\#p) \ast \text{program_counter}(p)
\]

\[
\text{and} \quad \text{param_count}(p) = 0 \quad \text{and}
\]

\[
\text{instruction_register}(p) = \\text{instruction_memory}(p)(\text{program_counter}(\#p)) \quad \text{and}
\]

\[
((\text{operation}(\text{instruction_register}(\#p)) = \text{NEW_FR}) \Rightarrow
\]

\[
\delta(p, (\text{program_counter}, \text{facility_register},
\text{instruction_register})) \quad \text{and}
\]

\[
\text{facility_register}(p) = \\text{facility_memory}(p)(\text{facility_register}(\#p)) +
\]

\[
\text{base}(\text{instruction_register}(\#p))
\]

\[
\text{and}
\]

\[
\text{return_stack}(p) = \text{return_stack}(\#p) \ast \text{facility_register}(p)
\]

\[
\text{and}
\]

\[
\text{program_counter}(p) = \text{program_counter}(p) + 1 \quad \text{and}
\]

\[
\text{instruction_register}(p) = \\text{instruction_memory}(p)(\text{program_counter}(\#p))
\]

\[
((\text{operation}(\text{instruction_register}(\#p)) = \text{RESTORE_FR}) \Rightarrow
\]

\[
\delta(p, (\text{program_counter}, \text{facility_register},
\text{instruction_register}, \text{return_stack})) \quad \text{and}
\]

\[
\text{program_counter}(p) = \text{program_counter}(p) + 1 \quad \text{and}
\]

\[
\text{instruction_register}(p) = \\text{instruction_memory}(p)(\text{program_counter}(\#p))
\]

\[
\text{return_stack}(\#p) = \text{return_stack}(p) \ast \text{facility_register}(p)
\]

\[
((\text{operation}(\text{instruction_register}(\#p)) = \text{NEW_FDR}) \Rightarrow
\]

\[
\delta(p, (\text{program_counter}, \text{facility_data_register},
\text{instruction_register}, \text{return_stack})) \quad \text{and}
\]

\[
\text{facility_data_register}(p) = \\text{indexed_memory}(p)(\text{facility_data_register}(\#p)) +
\]

\[
\text{base}(\text{instruction_register}(\#p))
\]

\[
\text{and}
\]

\[
\text{return_stack}(p) = \text{return_stack}(\#p) \ast \text{facility_data_register}(p)
\]

\[
\text{and}
\]
program_counter(p) = program_counter(#p) + 1 and
instruction_register(p) =
  instruction_memory(p)(program_counter(#p))

{\text{operation}(\text{instruction_register}(#p)) = \text{RESTORE_FDR}} \Rightarrow

& (p, \{\text{program_counter, facility_data_register,}
  \text{instruction_register}\}) \quad \text{and}

\begin{align*}
\text{program_counter}(p) &= \text{program_counter}(#p) + 1 \\
\text{instruction_register}(p) &= \\
\text{instruction_memory}(p)(\text{program_counter}(#p))
\end{align*}

\begin{align*}
\text{return_stack}(#p) &= \text{return_stack}(p) \cdot \text{facility_register}(p)
\end{align*}

end do_instruction
end Interface

description

A processor to support the Resolve programming language.
end description

dend Virtual_Machine_Template
Two notations used in the template are defined below.

\[ \Delta(f, \{d_1, \ldots, d_m\}) \]  
- \( f \) is a function.
  \( \{d_1, \ldots, d_m\} \) are values in the domain of \( f \) for which corresponding range values may be changed; at all other places in the domain of \( f \), the function is unchanged.

\[ \delta(r, \{f_1, \ldots, f_n\}) \]  
- \( r \) is a record.
  \( \{f_1, \ldots, f_n\} \) is the set of fields in \( r \) that may be changed; all other fields are unchanged.
BIBLIOGRAPHY


