Virtual Machines
To Dr P.W. Dale

(Uncle Paul)
Preface

I love virtual machines (VMs) and I have done for a long time. If that makes me “sad” or an “anorak”, so be it. I love them because they are so much fun, as well as being so useful. They have an element of original sin (writing assembly programs and being in control of an entire machine), while still being able to claim that one is being a respectable member of the community (being structured, modular, high-level, object-oriented, and so on). They also allow one to design machines of one’s own, unencumbered by the restrictions of a particular processor (at least, until one starts optimising it for some physical processor or other).

I have been building virtual machines, on and off, since 1980 or thereabouts. It has always been something of a hobby for me; it has also turned out to be a technique of great power and applicability. I hope to continue working on them, perhaps on some of the ideas outlined in the last chapter (I certainly want to do some more work with register-based VMs and concurrency).

I originally wanted to write the book from a purely semantic viewpoint. I wanted to start with a formal semantics of some language, then show how a virtual machine satisfied the semantics; finally, I would have liked to have shown how to derive an implementation. Unfortunately, there was insufficient time to do all of this (although some parts—the semantics of ALEX and a part proof of correctness—were done but omitted). There wasn’t enough time to do all the necessary work and, in addition, Stärk et al. had published their book on Java [47] which does everything I had wanted to do (they do it with Java; I had wanted to define ad hoc languages).

I hope to have made it clear that I believe there to be a considerable amount of work left to be done with virtual machines. The entire last chapter is about this. As I have tried to make clear, some of the ideas included in that chapter are intended to make readers think, even if they consider the ideas stupid!

A word or two is in order concerning the instruction sets of the various virtual machines that appear from Chapter Four onwards. The instructions
for the stack machines in Chapter Four seem relatively uncontroversial. The
instructions in the chapter on register machines (Chapter Seven) might seem
to be open to a little more questioning.

First, why not restrict the instruction set to those instructions required to
implement ALEX? This is because I wanted to show (if such a demonstration
were really required) that it is possible to define a larger instruction set so
that more than one language can be supported.

Next, most of the jump and arithmetic instructions seem sensible enough
but there are some strange cases, the jump branching to the address on the top
of the stack is one case in point; all these stack indexing operations constitute
another case. I decided to add these “exotic” instructions partly because,
strange as they might appear to some, they are useful. Somewhere or other,
I encountered a virtual machine that employed a jump instruction similar to
the one just mentioned (I also tried one out in one of the Harrison Machine’s
implementations—it was quite useful), so I included it. Similarly, a lot of time
is spent in accessing variables on the stack, so I added instructions that would
make such accesses quite easy to compile; I was also aware that things like
process control blocks and closures might be on stacks. I decided to add these
instructions to build up a good repertoire, a repertoire that is not
restricted to the instructions required to implement ALEX or one of the extensions
described in Chapter Five.

I do admit, though, that the mnemonics for many of the operations could
have been chosen with more care. (I was actually thinking that an assembler
could macro these names out.) One reason for this is that I defined the register
machine in about a day (the first ALEX machine was designed in about forty-
five minutes!). Another (clearly) is that I am not terribly good at creating
mnemonics. I thought I’d better point these matters out before someone else
does.

I have made every effort to ensure that this text is free of errors. Undoubt-
edly, they still lurk waiting to be revealed in their full horror and to show that
my proof-reading is not perfect. Should errors be found, I apologise for them
in advance.
Acknowledgements

Beverley Ford first thought of this book when looking through some notes I had made on abstract machines. I would like to thank her and her staff at Springer, especially Catherine Drury, for making the process of writing this book as smooth as possible.

My brother Adam should be thanked for creating the line drawings that appear as some of the figures (I actually managed to do the rest myself). I would also like to thank all those other people who helped in various ways while I was writing this book (they know who they are).

Iain Craig
Market Square
Atherstone
14 June, 2005
Contents

1 Introduction ................................................................. 1
  1.1 Introduction ......................................................... 1
  1.2 Interpreters .......................................................... 3
  1.3 Landin’s SECD Machine .............................................. 3
  1.4 The Organisation of this Book ...................................... 5
  1.5 Omissions .............................................................. 7

2 VMs for Portability: BCPL ............................................... 11
  2.1 Introduction .......................................................... 11
  2.2 BCPL the Language .................................................. 12
  2.3 VM Operations ....................................................... 15
  2.4 The OCODE Machine ................................................ 17
  2.5 OCODE Instructions and their Implementation .................. 18
    2.5.1 Expression Instructions ....................................... 18
    2.5.2 Load and Store Instructions ................................... 20
    2.5.3 Instructions Relating to Routines ............................. 20
    2.5.4 Control Instructions ........................................... 22
    2.5.5 Directives ...................................................... 23
  2.6 The Intcode/Cintcode Machine .................................... 24

3 The Java Virtual Machine ............................................... 27
  3.1 Introduction .......................................................... 27
  3.2 JVM Organisation: An Overview .................................... 28
    3.2.1 The stack ....................................................... 29
    3.2.2 Method areas .................................................. 30
    3.2.3 The PC register ............................................... 31
    3.2.4 Other structures ............................................... 32
  3.3 Class Files ........................................................... 32
  3.4 Object Representation at Runtime ................................ 40
  3.5 Initialisation ........................................................ 42
  3.6 Object Deletion ..................................................... 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7 JVM Termination</td>
<td>45</td>
</tr>
<tr>
<td>3.8 Exception Handling</td>
<td>45</td>
</tr>
<tr>
<td>3.9 Instructions</td>
<td>46</td>
</tr>
<tr>
<td>3.9.1 Data-manipulation instructions</td>
<td>48</td>
</tr>
<tr>
<td>3.9.2 Control instructions</td>
<td>51</td>
</tr>
<tr>
<td>3.9.3 Stack-manipulating instructions</td>
<td>54</td>
</tr>
<tr>
<td>3.9.4 Support for object orientation</td>
<td>56</td>
</tr>
<tr>
<td>3.9.5 Synchronisation</td>
<td>59</td>
</tr>
<tr>
<td>3.10 Concluding Remarks</td>
<td>59</td>
</tr>
<tr>
<td>4 DIY VMs</td>
<td>61</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>61</td>
</tr>
<tr>
<td>4.2 ALEX</td>
<td>62</td>
</tr>
<tr>
<td>4.2.1 Language Overview</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2 What the Virtual Machine Must Support</td>
<td>65</td>
</tr>
<tr>
<td>4.2.3 Virtual Machine—Storage Structures</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4 Virtual Machine—Registers</td>
<td>68</td>
</tr>
<tr>
<td>4.2.5 Virtual Machine—Instruction Set</td>
<td>70</td>
</tr>
<tr>
<td>4.2.6 An Example</td>
<td>79</td>
</tr>
<tr>
<td>4.2.7 Implementation</td>
<td>81</td>
</tr>
<tr>
<td>4.2.8 Extensions</td>
<td>85</td>
</tr>
<tr>
<td>4.2.9 Alternatives</td>
<td>88</td>
</tr>
<tr>
<td>4.2.10 Specification</td>
<td>93</td>
</tr>
<tr>
<td>4.3 Issues</td>
<td>96</td>
</tr>
<tr>
<td>4.3.1 Indirect and Relative Jumps</td>
<td>97</td>
</tr>
<tr>
<td>4.3.2 More Data Types</td>
<td>98</td>
</tr>
<tr>
<td>4.3.3 Higher-Order Routines</td>
<td>106</td>
</tr>
<tr>
<td>4.3.4 Primitive Routines</td>
<td>106</td>
</tr>
<tr>
<td>4.4 Concluding Remarks</td>
<td>107</td>
</tr>
<tr>
<td>5 More Stack-Based VMs</td>
<td>109</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>109</td>
</tr>
<tr>
<td>5.2 A Simple Object-Oriented Language</td>
<td>110</td>
</tr>
<tr>
<td>5.2.1 Language Overview</td>
<td>110</td>
</tr>
<tr>
<td>5.2.2 Virtual Machine—Storage Structures</td>
<td>111</td>
</tr>
<tr>
<td>5.2.3 Virtual Machine—Registers</td>
<td>113</td>
</tr>
<tr>
<td>5.2.4 Virtual Machine—Instruction Set</td>
<td>113</td>
</tr>
<tr>
<td>5.2.5 Extensions</td>
<td>116</td>
</tr>
<tr>
<td>5.2.6 Alternatives</td>
<td>116</td>
</tr>
<tr>
<td>5.3 A Parallel Language</td>
<td>117</td>
</tr>
<tr>
<td>5.3.1 Language Overview</td>
<td>117</td>
</tr>
<tr>
<td>5.3.2 Virtual Machine—Storage Structures</td>
<td>119</td>
</tr>
<tr>
<td>5.3.3 Virtual Machine—Registers</td>
<td>121</td>
</tr>
<tr>
<td>5.3.4 Virtual Machine—Instruction Set</td>
<td>122</td>
</tr>
<tr>
<td>5.3.5 Implementation</td>
<td>124</td>
</tr>
</tbody>
</table>
5.3.6 Extensions .............................................. 126
5.3.7 Alternatives ............................................ 128
5.3.8 Issues .................................................... 129
5.4 Concluding Remarks ...................................... 129
  5.4.1 Some Optimisations ................................... 129
  5.4.2 Combining the Languages ............................... 130

6  Case Study: An Event-Driven Language .................. 131
  6.1 Introduction ........................................... 131
  6.2 The Structure of Rules .................................. 133
  6.3 Events .................................................. 136
  6.4 Execution Cycle ........................................ 136
  6.5 Interpretation Rules .................................... 138
  6.6 VM Specification ....................................... 141
    6.6.1 States and Notational Conventions ................. 142
    6.6.2 Infra-Rule Transitions ............................. 145
    6.6.3 Extra-Rule Transitions ............................. 148
    6.6.4 VM-Only Transitions ............................... 150
    6.6.5 Introspective Operations ........................... 151
  6.7 Rule Equivalences ...................................... 153
  6.8 Concluding Remarks .................................... 154

7  Register-Based Machines ................................. 157
  7.1 Introduction .......................................... 157
  7.2 The Register-Transfer Model ........................... 158
  7.3 Register Machine Organisation ......................... 161
  7.4 Parrot—General Organisation ........................... 165
  7.5 Parrot Instruction Set .................................. 168
    7.5.1 Control instructions ............................... 169
    7.5.2 Data management instructions ...................... 169
    7.5.3 Register and stack operations ...................... 170
  7.6 DIY Register-Based Virtual Machine .................... 171
    7.6.1 Informal Design .................................. 172
    7.6.2 Extensions ........................................ 176
    7.6.3 Transition Rules ................................... 177
  7.7 Translating ALEXVM into RTM ............................ 183
  7.8 Example Code .......................................... 186
  7.9 Correctness of the Translation ......................... 186
  7.10 More Natural Compilation .............................. 196
  7.11 Extensions .......................................... 200
8 Implementation Techniques .............................................. 201
  8.1 Stack-Based Machines ............................................. 202
    8.1.1 Direct Implementation ..................................... 202
    8.1.2 Translation .................................................. 203
    8.1.3 Threaded Code .............................................. 207
  8.2 Register Machines ................................................ 209
    8.2.1 Register sets ................................................ 210
    8.2.2 Addressing .................................................. 210
    8.2.3 Translation to Another VM .................................. 212
  8.3 Using Transitions ................................................ 212
  8.4 Concluding Remarks .............................................. 213

9 Open Issues .......................................................... 215
  9.1 Security .......................................................... 215
  9.2 New Languages ................................................... 216
  9.3 Typed Instruction Sets and Intermediate Codes ................. 216
  9.4 High-Level Instructions ......................................... 218
  9.5 Additivity and Replacement ..................................... 218
  9.6 Compiler Correctness ............................................ 218
  9.7 Dynamic Code Insertion ......................................... 219
  9.8 Instrumentation .................................................. 220
  9.9 Including more Information about Source Code ................. 221
  9.10 Integration with Databases ..................................... 222
  9.11 Increased Inter-Operability .................................... 222
  9.12 Code Mobility ................................................... 223
  9.13 Small Platforms ................................................ 224
  9.14 Real-Time VMs .................................................. 226
  9.15 Code Morphing .................................................. 227
  9.16 Greater Optimisation .......................................... 227
  9.17 Operating System Constructs .................................. 228
  9.18 Virtual Machines for more General Portability ............... 229
  9.19 Distributed VMs ................................................ 229
  9.20 Objects and VMs ................................................ 229
  9.21 Virtual VMs ..................................................... 230
  9.22 By Way of a Conclusion ........................................ 231

A Compiling ALEX ....................................................... 233
  A.1 Introduction ..................................................... 233
  A.2 Notational Conventions ......................................... 233
  A.3 Compilation Rules .............................................. 235

B Harrison Machine Compilation Rules .................................. 241
  B.1 Introduction ..................................................... 241
  B.2 Compilation Rules .............................................. 241
Contents

C Harrison Machine Instruction Set .................................... 257
References ........................................................................... 261
Index .................................................................................... 265
1

Introduction

1.1 Introduction

There are, basically, two ways to implement a programming language: compile it or interpret it. Compilers are usually written for a single target machine; the GNU C compiler is a partial counter-example, containing, as it does, code generators for a number of target architectures (actually, the compiler has to be compiled for a specific target and it is only the full distribution that contains the complete set of code generators). Interpreters are thought to be slow but easy to port.

An interpreter can operate on the source structure of a program (as many LISP interpreters do) or can execute an internal form (for example, polish notation), while virtual machines combine both compilation and interpretation. Virtual machines consist of a compiler and a target architecture implemented in software. It contains a core that deals with the execution of code that has been compiled into the instruction set for the virtual machine’s software architecture. The core executes these instructions by implementing the operations defined by the instruction set (which can be seen as a form of emulation or interpretation). Much of the traditional runtime package functionality associated with compiled code is implemented as part of a virtual machine; this clearly serves as an invitation to expand available functionality to provide rich execution environments for programs. It also opens up the possibility that traditional linkage methods (as exemplified by the linkage editor or by dynamic linkage of modules) can be eliminated in favour of more flexible methods.

Virtual machines are used as a method for ensuring portability, as well as for the execution of languages that do not conform well (or at all) to the architecture of the target architecture. As noted in the last paragraph, they afford opportunities to enrich the execution environment as well as greater flexibility.

It is the case that code in compiled form executes considerably faster than interpreted code, with interpreted code running at one or two orders of magnitude slower than the corresponding compiled form. For many, optimising
compilers are the *sine qua non*, even though the output code can bear little resemblance to the source, thus causing verification problems (there is, and never can be, a viable alternative to the selection of good or, yet better, optimal algorithms) but optimising compilers are highly platform specific. The virtual machine is also a method for increasing the general speed of execution of programs by providing a single site that can be tuned or improved by additional techniques (a combination of native code execution with virtual machine code).

In a real sense, virtual machines constitute an execution method that combines the opportunities for compiler optimisation with the advantages of interpretation.

Although virtual machines in the form of “abstract machines” have been around for a long time (since the mid-1960s), the advent of Java has made them a common (and even fashionable) technique for implementing new languages, particularly those intended for use in heterogeneous environments. As noted above, many languages (Prolog, Curry and Oz, to cite but three) have relied upon virtual machines for a long time.

It is clear that the sense in which the term “virtual machine” is construed when considering execution environments for programs in particular programming languages relates to the other senses of the term. To construct a virtual machine for some programming language or other amounts, basically, to the definition of mechanisms that correspond to the actions of some computational machine (processor) or other.\(^1\)

In the sense of the term adopted in this book, existing hardware imposes no constraints upon the designer other than the semantics of the programming language to be executed on the virtual machine. This view now seems to underpin ideas on the production of more general “virtual machines” that are able to execute the code of more than one programming language and to provide support to executing programs in other ways.

Virtual machines constitute an active research area. This book is intended as an invitation to engage in and contribute to it. This is manifested in a number of ways:

- The use of transitions as a way of specifying virtual machine instructions. (This leads to the idea of completely formal specifications, although this is not followed up in this book—for a formal description of the JVM, [47] is recommended.)
- The use of register-based virtual machines. Most virtual machines are based on stacks. In the register-based approach, it seems possible to widen the scope of virtual machines by providing more general instruction sets that can be tailored or augmented to suit particular languages.

\(^1\) This latter sense is the one adopted by the designers of IBM’s VM operating system; it implemented the underlying hardware as a software layer.
1.3 Landin's SECD Machine

- The idea of translating ("morphing") code from one virtual machine for execution on another. This raises correctness issues that are partially addressed in this book.

1.2 Interpreters

Since the 1950s, it has been possible to execute programs in compiled form or in interpreted form. LISP was originally implemented in interpreted form, as was BASIC. The LISP interpreter was only a first stage of the project (since then, extremely high-quality LISP compilers have been built) but BASIC was intended from the outset to be an interpreted language. Since then, interpreters have been implemented for a great many languages.

Gries, in his [23], devotes a single chapter to interpreters. He gives the example of the interpretation of the Polish form of a program and describes the organisation of an interpreter, as well as runtime storage allocation. The techniques involved in interpretation are a subset of those in compilation to native code.

1.3 Landin's SECD Machine

In [30], Landin introduced the SECD machine. This was originally intended as a device for describing the operational semantics of the $\lambda$-calculus. Landin showed how the machine could be used to implement a functional programming language called ISWIM ("If you See What I Mean"$^2$). Since its introduction, the SECD machine has been adapted in various ways and used to describe the operational semantics of a great many languages, some functional, some not. The machine has shown itself easy to adapt so that features like lazy evaluation, persistence and assignment can easily be accommodated within it.

Since the SECD machine is arguably the first virtual machine (or "abstract machine" as they used to be called),$^3$ it is useful to sketch its major points. A brief sketch of the machine occupies the remainder of this section.

The SECD machine gets its name from its main components or registers (often erroneously called "stacks"):

- $S$: The state stack.
- $E$: The environment stack.
- $C$: The control list.
- $D$: The dump stack.

$^2$ Many have observed that it should be "Do you See What I Mean" — DYSWIM just doesn't have the ring, though.

$^3$ I.e., the first thing to be called an "abstract machine" in technical usage and almost certainly the first to be so called in the literature.
Each of these components will be described in turn.

The S, state, register is a stack that is used for the evaluation of expressions. It is usually just called the stack. To evaluate an expression such as $5 + 3$, the values are pushed onto the S register (in reverse order) and then the operator $+$ is applied to them. Just prior to the application of the addition operation, the stack would be:

$$5 \cdot 3 \cdot \ldots$$

After application of $+$, the S register becomes:

$$8 \cdot \ldots$$

(The $S$ register is assumed to grow to the left. The raised dot, $\cdot$, just separates the values.)

The $E$ register is the environment register (usually just called the environment). The environment contains variable bindings. That is, it contains mappings from variables to their values. When a function is called, actual parameters are supplied. The environment for a function will record the mapping from formal to actual parameters, thus allowing the value of each parameter to be looked up when it is required.

For example, consider the unary function $f(x)$. When this function is applied to an argument, say $f(4)$, the binding of 4 to $x$ is recorded somewhere in the $E$ register. Inside $f$, when the value of $x$ is needed, it is looked up in the environment and the value 4 is obtained. The environment is also used to store the values of local variables. The code to access the environment, both to bind and to lookup variable bindings is stored in the $C$ register and is produced by a compiler generating SECD machine code.

The $C$ register contains a sequence of SECD machine instructions. It is not really a stack but a simple list or vector. A pointer runs down the $C$ register, pointing to each instruction in turn; in other machines, this pointer would be called the instruction pointer or the program counter; in most SECD implementations, the topmost element in the $C$ register is shown.

The instructions used by an implementation of the SECD machine define what is to be done with the $S$, $E$ and $D$ registers (it is not impossible for them to define changes to the $C$ register but it is rather rare). For example, the addition instruction states that the top two elements are to be popped from $S$, added and the result pushed onto $S$.

The final register is the $D$ register, or the dump. The dump is used when the state of the machine must be stored for some reason. For example, when a routine is called, the caller’s local variables and stack must be saved so that the called routine can perform its computations. In the SECD machine, the registers are saved together in the dump when a routine is called. When a routine exits, the dump’s topmost element is popped and the machine’s registers are restored.

---

4 It will be given a more precise interpretation later in this book.
To make this a little clearer, consider an SECD machine. It is described by a 4-tuple \( S, E, C, D \). When a call is made within one routine to another routine, the current instruction in the \( C \) register could cause the following state transition:

\[
(s, e, c, d) \text{ becomes } (\emptyset, e', c', (s, e, c, d) \cdot d')
\]

That is, an empty stack is put into the \( S \) and a new environment established in the \( E \) register; the code for the called routine is put into the \( C \) register. Meanwhile, the dump contains a 4-tuple consisting of the state of the calling routine. That state is suspended until the called routine exits.

On exit, the called routine executes an SECD machine instruction that effects the following transition:

\[
(s', e', c', (s, e, c, d) \cdot d') \text{ becomes } (s, e, c, d')
\]

I.e., everything is put back where it belongs! (Transitions, more completely formalised, will be used later in this book.)

In addition, the SECD machine requires some storage management, typically a heap with a garbage collector. In most implementations, the \( S, E, C \) and \( D \) registers are implemented as lists. This implies that some form of heap storage is required to manage them. The Lispkit implementation described in [24] implements the three registers in this way and includes the (pseudo-code) specification of a mark and sweep garbage collector.

There are many, different publications containing descriptions of the SECD machine. The book by Field and Harrison [18], as well as Henderson’s famous book on Lispkit [24] are two, now somewhat old, texts containing excellent descriptions of the SECD machine.

### 1.4 The Organisation of this Book

The chapter that immediately follows this (Chapter Two) is concerned with the BCPL OCODE and Cintcode/Intcode machines (in older versions, the bootstrap code was called Intcode, while in the newer, C-based, ones it is called Cintcode). BCPL is a relatively old language, although one that still has devotees,\(^5\) that was always known for its portability. Portability is achieved through the definition of a virtual machine, the OCODE machine, that executes BCPL programs. The OCODE machine can be implemented from scratch or bootstrapped using Cintcode Intcode, a process that involves the construction of a simple virtual machine on each new processor that is used to implement the full OCODE machine. The OCODE machine and its instruction set are described in that chapter.

Chapter Three contains a relatively short description of the Java Virtual Machine (JVM), possibly the most famous and widely used virtual machine

---

\(^5\) Such as the author.
at the time of writing. The JVM’s main structures are described, as is its instruction set.

Doing it yourself\(^6\) is the subject of Chapter Four. First, a simple procedural language, called ALEX, is introduced and informally defined. The main semantic aspects of the language are identified. A simple stack-based virtual machine for ALEX is then described in informal terms; this description is then converted into an Algol-like notation. Some extensions to the virtual machine (driven by extensions to the language) are then considered. An alternative organisation for the virtual machine is then proposed: it employs two stacks (one for control and one for data) rather than one, thus requiring alterations to the definition of the instruction set. This machine is then specified using transition rules. A compiler for a large subset of ALEX is specified in Appendix A; the compiler translates source code to the single-stack virtual machine.

The DIY theme continues in Chapter Five. This chapter contains the descriptions of two virtual machines: one for a simple object-oriented language, the other for a language for pseudo parallelism. The base language in both cases is assumed to be the simple dialect of ALEX with which Chapter Four started. In each case, extensions are considered and discussed (there appears to be more to say about the pseudo-parallel language).

The idea of introducing the DIY virtual machines is that they can be introduced in a simple form and then subjected to extensions that suit the various needs of different programming languages. Thus, the ALEX virtual machine starts with a call-by-value evaluation scheme which is later extended by the addition of call by reference; ALEX first has only vectors but records are added at a later stage. In addition, the DIY approach allows the extension and optimisation of the instruction set to be discussed without reference to an existing (and, hence, fixed) language and associated virtual machine.

By way of relief, an event-based language is considered in Chapter Six. This language is somewhat different and has a semantics that is not entirely procedural (although it contains procedural elements) and is not a straight pseudo-parallel language (although it can be related to one); the system was designed (and implemented) as part of the author’s work on computational reflection. The virtual machine is a mixture of fairly conventional instructions, instructions for handling events and event queues and, finally, instructions to support (part of) the reflective behaviour that was desired. In order to make the virtual machine’s definition clearer, a more mathematical approach has been adopted; transitions specify the instructions executed by the virtual machine. A compiler for the language executed by this virtual machine is specified in Appendix B.

\(^6\) For readers not familiar with the term, “DIY” stands for “Do It Yourself”. It usually refers to home “improvements”, often in kitchens and bathrooms. The result is often reminiscent of the detonation of a medium-calibre artillery shell (or so it seems from TV programmes on the subject). The author explicitly and publicly denies all and any knowledge of home improvements.
An alternative to the stack-based virtual machine organisation is considered in Chapter Seven. This alternative is based on the register-transfer model of computer processors. An argument in favour of this model is first given; this is followed by a short description of the Parrot virtual machine for Perl6 and Python (and, it is to be hoped, many other languages). After this, a DIY register machine is described, first informally and then using transitions. After considering possible extensions, a translation from the two-stack virtual machine code to the register-based virtual machine is presented (it is intended as a motivating example for code translation between virtual machines, an issue, referred to as "code morphing" and discussed in Chapter 9). The correctness of this translation is considered in a semi-formal way. Finally, a more natural translation from ALEX to register-based code is considered before more extensions are discussed.

Register-based virtual machines are discussed because they appear to be an effective alternative to the more usual method of using stack (or zero-address) machines. The author experimented with such a virtual machine as part of the work on the Harrison Machine, the system described in Chapter Six (although not discussed there). The discovery that the Parrot group was using a similar approach for Perl6 appeared a strong basis for the inclusion of the topic in this book.

The implementation of virtual machines is considered in Chapter Eight. Implementation is important for virtual machines: they can either be considered theoretical devices for implementing new constructs and languages or practical ways to implement languages on many and many platforms.

In Chapter Eight, a number of implementation techniques are considered, both for stack- and register-based virtual machines. They include the direct translation to a language such as C and to other virtual machines. The use of different underlying organisations, such as threaded code, is also discussed.

The last chapter, Chapter Nine is concerned with what are considered to be open issues for those interested in pushing forward the virtual machine approach. This chapter is, basically, a somewhat loosely organised list—a brainstorming session—of ideas, some definitely worth investigating, some possibly dead ends, that are intended to stimulate interest in further work.

1.5 Omissions

Virtual machines are extremely popular for the implementation of languages of all kinds. It is impossible in a book of this length to discuss them all; it is also impossible, realistically, to discuss a representative sample.

Prolog is a good example of a language that has been closely associated with a virtual (or abstract) machine for a long time. The standard virtual machine is that of Warren [52] (the Warren Abstract Machine or WAM). A description of the WAM was considered and then rejected, mostly because of the excellent book by Ait-Kaci [3] on the WAM. Readers interested in logic
programming languages would be well advised to read and completely digest [3]; readers just interested in virtual machines will also find it a pleasure to read.

The Scheme language (a greatly tidied-up LISP dialect with static scope) [28] has been associated with compilers since its inception. However, there is a virtual machine for it; it is described in [1] (the chapter on register machines). The implementation there can be used as the basis for a working implementation (indeed, many years ago, the author used it as a stage in the development of a compiled system for experimenting with reflection). Although intended for undergraduates, [1] is highly informative about Scheme (and is also a good read).

Pascal was distributed from ETH, Zürich, in the form of an abstract machine (VM) that could be ported with relative ease. The UCSD Pascal system was also based on an abstract machine. The notion of using a virtual machine to enhance portability is covered below in the chapter on BCPL (Chapter 2). BCPL is simpler in some ways than Pascal: it only has one primitive type (the machine word) and a few derived types (tables and vectors). BCPL’s machine is a little earlier than that of Pascal, so it was decided to describe it. (BCPL will also be less familiar to many readers and was a major influence on the design of C.)

Smalltalk [21] also has a virtual machine, which is defined in [21] in Smalltalk. The Smalltalk VM inspired the pseudo-parallel virtual machine described in Chapter 5; it was also influential in the design of the Harrison Machine (Chapter 6). A full description of the Smalltalk VM would have taken a considerable amount of space, so it was decided to omit it.

The Poplog system [42], a system for AI programming that supports CommonLISP, Prolog, Pop11 and Standard ML, uses a common virtual machine. Pop11 is used for all systems programming, so the virtual machine is tailored to that language. However, the Lisp, Prolog and ML compilers are written in Pop11 and generate virtual machine code. The Prolog compiler is based on a continuation-passing model, not on the Warren Abstract Machine, so the Poplog instruction set can be utilised directly. The Pop11 language is, in the author’s opinion, worth studying in its own right; the virtual machine and the compilation mechanisms are also worth study. The Poplog system distribution contains on-line documentation about itself.

There are many other virtual machines that could not be included in this book. They include VMs for:

- Functional languages (e.g., the G-machine [25] and derivatives [39]; the FPM [7]);
- Functional-logic programming languages;
- Constraint languages (the Oz language is an interesting example).

The author hopes it brings a smile to the lips of British readers, as well as fond and not-so fond memories.
Some readers will also ask why no attention has been paid to Just-In Time (JIT) compilers, particularly for Java. One reason is that this is a technique for optimising code rather than a pure-virtual machine method. Secondly, JIT compilers are a method for integrating native code (compiled on the fly) with a virtual machine. As such, it requires an interface to the virtual machine on which other code runs. In the treatment of the Java virtual machine, the native code mechanism is outlined; this is one method by which native code methods can be integrated.

Given the plethora of virtual machines, the reader might ask why it was decided to describe only three mainstream ones (BCPL, Java and Parrot) and to rely on (probably not very good) home-grown ones. The reasons are as follows:

- If the book had been composed only of descriptions of existing virtual machines, it would be open to the accusation that it omits the $X$ virtual machine for language $L$. This was to be avoided.
- Home-grown ones could be developed from scratch, thus making clear the principles that underpin the development of a virtual machine.
- In the mainstream, only the Java virtual machine combines both objects and concurrency. It was decided to present new, independent virtual machines so that differences in language could be introduced in various ways. The home-grown approach allows language and virtual machine features to be included (or excluded) *ad libitum* (even so, an attempt has been made to be as comprehensive as possible within the confines of a book of this length—hence the various sections and subsections on extensions and alternatives).
- At the time of writing, the Parrot virtual machine appears to be the only generally available one based on the register-transfer model. The author independently came to conclusions similar to those of the designers of Parrot as to the merits of register-based machines (and on treating virtual machines as data structures) and wanted to argue for this alternative model. As a consequence, the mapping between stack- and register-based models was of importance (as are some of the suggestions for further work in the Chapter 9).
- The derivation of transitions specifying many virtual machines would not have been possible in the time available for the writing of this book. Furthermore, an existing virtual machine is an entity, so the introduction of new instructions (e.g., branches or absolute jumps) would have been less convincing; the *ad hoc* virtual machines described below can be augmented as much as one wishes.  

---

8 Interested readers are actively encouraged to implement the virtual machines in this book and augment them as they see fit, as well as introducing new instructions by defining new transitions.
• Finally, the definition of a virtual machine can be a testing, rewarding and enjoyable exercise. An aim of the current book is to encourage people to do it for themselves and to use their imagination in defining them.
2.1 Introduction

BCPL is a high-level language for systems programming that is intended to be as portable as possible. It is now a relatively old language but it contains most syntactic constructs found in contemporary languages. Indeed, C was designed as a BCPL derivative (C can be considered as a mixture of BCPL and Algol68 plus some *sui generis* features). BCPL is not conventionally typed. It has one basic data type, the machine word. It is possible to extract bytes from words but this is a derived operation. All entities in BCPL are considered either to be machine words or to require a machine word or a number of machine words. BCPL supports addresses and assumes that they can fit into a single word. Similarly, it supports vectors (one-dimensional arrays) which are sequences of words (multi-dimensional arrays must be explicitly programmed in terms of vectors of pointers to vectors). Routines (procedures and functions) can be defined in BCPL and are represented as pointers to their entry points. Equally, labels are addresses of sequences of instructions.

BCPL stands for "Basic CPL", a subset of the CPL language. CPL was an ambitious lexically scoped, imperative procedural programming language designed by Strachey and others in the mid-1960s as a joint effort involving Cambridge and London Universities. CPL contained all of the most advanced language constructs of the day, including polymorphism. There is a story that the compiler was too large to run on even the biggest machines available in the University of London! Even though it strictly prefigures the structured programming movement, BCPL contains structured control constructs (commands) including two-branch conditionals, switch commands, structured loops with structured exits. It also supports statement formulae similar to those in FORTRAN and the original BASIC. Recursive routines can be defined. BCPL does support a goto command. Separate compilation is supported in part by the provision of a "global vector", a vector of words that contains pointers to externally defined routines. BCPL is lexically scoped. It implements call-by-value semantics for routine parameters. It also permits higher-order
programming by permitting routine names to be assigned to variables (and, hence, passed into and out of routines).

BCPL was intended to be portable. Portability is achieved by bootstrapping the runtime system a number of times so that it eventually implements the compiler’s output language. This language is called OCODE. OCODE is similar to a high-level assembly language but is tailored exactly to the intermediate representation of BCPL constructs. OCODE was also defined in such a way that it could be translated into the machine language of most processors. Associated with OCODE is an OCODE machine that, once implemented, executes OCODE, hence compiled BCPL. The implementation of an abstract machine for OCODE is relatively straightforward.

In the book on BCPL [45], Richards and Whitby-Strevens define a second low-level intermediate language called Intcode. Intcode is an extremely simple language that can be used to bootstrap OCODE. More recently, Richards has defined a new low-level bootstrap code called Cintcode. The idea is that a fundamental system is first written for Intcode/Cintcode. This is then used to bootstrap the OCODE evaluator. The definition of the Intcode and Cintcode machines is given in the BCPL documentation. The BCPL system was distributed in OCODE form (more recent versions distribute executables for standard architectures like the PC under Linux). At the time the book was published, an Intcode version of the system was required to bootstrap a new implementation.

The virtual machines described below are intended, therefore, as an aid to portability. The definitions of the machines used to implement OCODE and Intcode/Cintcode instructions include definitions of the storage structures and layout required by the virtual machine, as well as the instruction formats and state transitions.

The organisation of this chapter is as follows. We will focus first on BCPL and its intermediate languages OCODE and Intcode/Cintcode (Cintcode is part of the current BCPL release and access to the documentation is relatively easy). We will begin with a description of the OCODE machine. This description will start with a description of the machine’s organisation and then we move on to a description of the instruction set. The relationship between OCODE instructions and BCPL’s semantics will also be considered. Then, we will examine Cintcode and its abstract machine. Finally, we explain how BCPL can be ported to a completely new architecture.

2.2 BCPL the Language

In this section, the BCPL language is briefly described.

BCPL is what we would now see as a relatively straightforward procedural language. As such, it is based around the concept of the procedure. BCPL provides three types of procedural abstraction:

- Routines that update the state and return no value;
2.2 BCPL the Language

- Routines that can update the state and return a single value;
- Routines that just compute a value.

The first category refers to procedures proper, while the second corresponds to the usual concept of function in procedural languages. The third category corresponds to the single-line functions in FORTRAN and in many BASIC dialects. Each category permits the programmer to pass parameters, which are called by value.

BCPL also supports a variety of function that is akin to the so-called "formula function" of FORTRAN and BASIC. This can be considered a variety of macro or open procedure because it declares no local variables.

BCPL supports a variety of state-modifying constructs. As an imperative language, it should be obvious that it contains an assignment statement. Assignment in BCPL can be simple or multiple, so the following are both legal:

\[
\begin{align*}
x &:= 0; \\
x, y &:= 1, 2; \\
\end{align*}
\]

It is worth noting that terminating semicolons are optional. They are mandatory if more than one command is to appear on the same line as in:

\[
\begin{align*}
x &:= 0; \\
y &:= 2 \\
\end{align*}
\]

Newline, in BCPL, can also be used to terminate a statement. This is a nice feature, one found in only a few other languages (Eiffel and Imp, a language used in the 1970s at Edinburgh University).

Aside from this syntactic feature, the multiple assignment gives a clue that the underlying semantics of BCPL are based on a stack.

In addition, it contains a number of branching constructs:

- **IF ... DO.**\(^1\) This is a simple test. If the test is true, the code following the DO is executed. If the test is false, the entire statement is a no-operation.
- **UNLESS ... DO.** This is syntactic sugar for **IF NOT ... DO.** That is, the code following the DO is executed if the test fails.
- **TEST ... THEN ... ELSE.** This corresponds to the usual if then else in most programming languages.
- **SWITCHON.** This is directly analogous to the **case** statement in Pascal and its descendants and to the **switch** statement in C and its derivatives. Cases are marked using the **CASE** keyword. Cases run into each other unless explicitly broken. There is also a an optional default case denoted by a keyword. Each case is implicitly a block.

In general, the syntax word **do** can be interchanged with **then.** In the above list, we have followed the conventions of BCPL style.

BCPL contains a number of iterative statements. The iterative statements are accompanied by structured ways to exit loops.

---

\(^1\) Keywords must be in uppercase, so the convention is followed here.
BCPL has a goto, as befits its age.

BCPL statements can be made to return values. This is done using the pair of commands VALOF and RESULTIS. The VALOF command introduces a block from which a value is returned using the RESULTIS command; there can be more than one RESULTIS command in a VALOF block. The combination of VALOF and RESULTIS is used to return values from functions. The following is a BCPL procedure:

```bcpl
LET Add.Global (x) BE
  $(
    globl := globl + x;
  )

The following is a BCPL functional routine:

```bcpl
LET Global.Added.Val (x) =
  $(
    VALOF $(
      RESULTIS(x+globl);
    )
  )
```

From this small example, it can be seen that the body of a procedure is marked by the BE keyword, while functional routines are signalled by the equals sign and the use of VALOF and RESULTIS (BCPL is case-sensitive).

BCPL is not conventionally typed. It has only one data type, the machine word, whose size can change from machine to machine. The language also contains operators that access the bytes within a machine word. Storage is allocated by the BCPL compiler in units of one machine word. The language contains an operator that returns the address of a word and an operator that, given an address, returns the contents of the word at that address (dereferencing).

BCPL supports structured types to a limited extent. It permits the definition of vectors (single-dimension arrays of words). It also has a table type. Tables are vectors of words that are indexed by symbolic constants, not by numerical values. In addition, it is possible to take the address of a routine (procedure or function); such addresses are the entry points of the routines (as in C). The passing of routine addresses is the method by which BCPL supports higher-order routines (much as C does).

It also permits the definition of symbolic constants. Each constant is one machine word in length.

BCPL introduces entities using the LET syntax derived from ISWIM. For example, the following introduces a new variable that is initialised to zero:

```bcpl
LET x := 0 IN
```

The following introduces a constant:
2.3 VM Operations

The summary of BCPL above was intended to expose the major constructs. The identification of major constructs is important for the design of a virtual machine which must respect the semantics of the language as well as providing the storage structures required to support the language.

At this stage, it should be clear that a BCPL machine should provide support for the primitive operations needed for the manipulation of data of all primitive types. The virtual machine support for them will be in the form of instructions that the machine will directly implement. In BCPL, this implies that the virtual machine must support operations on the word type: arithmetic operations, comparisons and addressing. Byte-based operations can either be provided by runtime library operations or by instructions in the virtual machine; BCPL employs the latter for the reason that it is faster and reduces the size of the library. In addition, BCPL supports vectors on the stack; they must also be addressed when designing an appropriate virtual machine.

The values manipulated by these operations must be stored somewhere: a storage area, particularly for temporary and non-global values must be provided. Operations are required for manipulating this storage area. Operations are also required to load values from other locations and to store them as results. More than one load operation might be required (in a more richly typed language, this might be a necessity) and more than one store operation might be required. It is necessary to look at the cases to determine what is required.

LET x = 0 IN

Multiple definitions are separated by the AND keyword (logical conjunction is represented by the “&” symbol) as in:

LET x := 0
AND y = 0
IN

Routines are also introduced by the LET construct.

Variables and constants can be introduced at the head of any block.

In order to support separate compilation and to ease the handling of the runtime library, a global vector is supported. This is a globally accessible vector of words, in which the first few dozen entries are initialised by the runtime system (they are initialised to library routine entry points and to globally useful values). The programmer can also assign to the global vector at higher locations (care must be taken not to assign to locations used by the system). These are the primary semantic constructs of BCPL. Given this summary, we can now make some observations about the support required by the virtual machine (the OCODE machine).
BCPL employs static scoping. The compiler can be relied upon to verify that variables, etc., are not required. Static scoping requires a stack-like mechanism for the storage of variables. The virtual machine is, therefore, built around a stack. Operations are required to allocate and free regions of stack at routine entry and exit; the return of results can also be implemented by means of stack allocation and addressing. The compiler generates instructions that allocate and free the right amount of stack space; it also generates instructions to handle returned values and the adjustment of the stack when routines return. Evaluation of expressions can be performed on the stack, so we now are in a position to define the instructions for data manipulation.

With expressions out of the way, the following families of construct must be handled by the compiler and OCODE instructions generated to implement them:

- Control constructs, in particular, conditionals, iteration, jumps;
- Assignment;
- Routine call and return;
- Parameter passing and value return from routines and valof.

Note that we assume that sequencing is handled implicitly by the compiler.

Control structure is handled, basically, by means of labels and jumps. There are clear translations between most of the control structures and label-jump combinations. The problem cases are **FOR** and **SWITCHON**. The former is problematic because it requires counters to be maintained and updated in the right order; the latter because the best implementation requires a jump table.

Assignment is a relatively straightforward matter (essentially, push a value onto the stack and pop it off to some address or other). Multiple assignment is also easy with a stack machine. The values are pushed onto the stack in some order (say left to right) and popped in the reverse order. Thus, the command:

\[ p, q := 1, 2 \]

has the intention of assigning 1 to \( p \) and 2 to \( q \). This can be done by pushing 1, then 2 onto the stack and assigning them in reverse order. An interesting example of multiple assignment is:

\[ p, q := q, p \]

Swap! It can be handled in exactly the manner just described.

Finally, we have routine calls and **VALOF**. There are many ways to implement routine calls. For software virtual machines, relatively high-level instructions can be used (although low-level instructions can also be employed). The OCODE machine provides special instructions for handling routine entry and exit, as will be seen.

BCPL is a call-by-value language, so the runtime stack can be directly employed to hold parameter values that are to be passed into the routine.