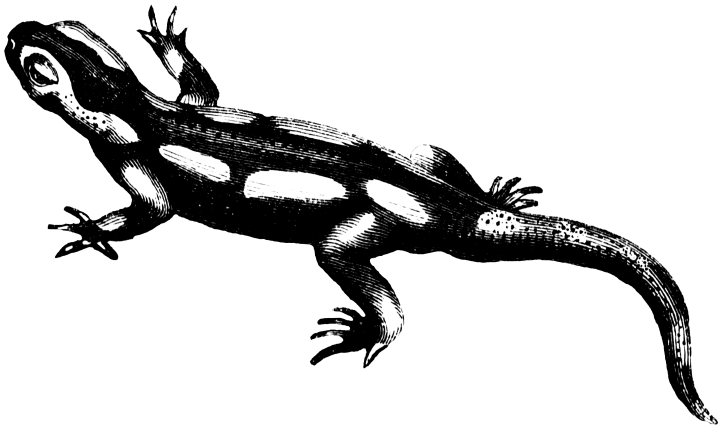
Alex Davies

ajd74@cam.ac.uk

Salamander Virtual Machine



Computer Science Tripos

St John's College

May 14, 2008

Cover image courtesy of Florida Center for Instructional Technology

Proforma

Name:	**Alex Davies**
College:	**St John's College**
Project Title:	**The Salamander Virtual Machine**
Examination:	**Computer Science Tripos, June 2008**
Word Count:	**10263**
Project Originator:	**Alex Davies & Christian Steinruecken**
Supervisor:	**Christian Steinruecken**

Original Aims of the Project

The aim of this project was to design and specify a virtual machine (VM) which allows implementations to provide any subset of the defined instructions. As long as that subset is Turing-complete, it should then allow any program to be run, by emulating the remaining instructions in terms of the ones provided. The project should provide an example implementation on a desktop computer and on a microcontroller device. It should give example programs to demonstrate and test the implementations and instruction emulations. There should be a means to translate Java Virtual Machine (JVM) programs to the project VM format, and of course these should successfully run even on the microcontroller device.

Work Completed

The specification for the VM was produced. The implementations on a desktop and microcontroller device (Altera DE2 board), some examples, and the translator from JVM bytecode were all completed successfully. In addition, I wrote tools to help me to achieve the core aims, notably an assembler and a tool to organise the routines that emulate instructions. Also, as an extension, I implemented the VM in hardware, using the Verilog hardware description language, also for a DE2 board.

Special Difficulties

No unexpected difficulties.

Declaration

I, Alex Davies of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

I'd like to thank:

- Christian Steinruecken for the original idea, and for excellent guidance while supervising the project
- Ben Challenor for helping to clarify my ideas each time I confused myself

Chapter 1

Introduction

My project aimed to develop a specification for a virtual machine (VM) which can operate correctly when its implementation is incomplete and instructions are missing from the full instruction set.

Missing instructions are emulated in terms of existing ones, using a method that is invisible to both the VM implementer and the application programmer.

This is achieved by a carefully designed software layer in the VM bytecode. It tests the underlying VM and builds up the full instruction set from whatever instructions are available. Applications can then run correctly even on minimal VM implementations, while running with the maximum possible performance on more complete implementations.

I have successfully designed this VM, written implementations for a variety of architectures, and demonstrated that it can correctly run a variety of example programs. These include some written in Java and automatically translated to be run in my VM. This fulfills all the requirements of my proposal, and extends it by providing an extra implementation – a hardware machine written in Verilog.

1.1 Terminology

I have named the system the Salamander Virtual Machine (SVM) in homage to the way that these creatures can grow back limbs that they are missing.

I will use the term VM to mean the abstract specification of what a machine must do (some might call this an abstract machine). Contrast this with an actual program which can interpret programs written for a VM, which I will always call an interpreter or an implementation.

1.2 Motivation

Portability of programs across many platforms is an important property that software designers seek, so that they can reach a wider market, and provide better flexibility. One of the most successful techniques to achieve it is to write for a VM, which then has an implementation available on many platforms.

So people have produced VMs, for example the JVM and Microsoft's CLI, which have proven successful. However, the designers of these VMs have had to make a trade-off between the run-time efficiency of the VM and its cost to implement. Because efficiency is so important, the specifications have been left complex.

Teams making hardware devices may be less inclined to implement a VM for their device because of the time they would have to spend providing the whole VM. It may be less costly to retarget the software they need, rather than writing a VM implementation and using existing VM-targeted versions.

My idea is that it is useful to be able to implement an inefficient VM interpreter, at low cost, to start with, and to incrementally improve it. This means that features that aren't initially provided are emulated using routines written in the VM bytecode. This is exactly what is made possible by the Salamander Virtual Machine.

1.3 Multiple paradigms

Building on the basic idea of half-implementing a VM, we can investigate what makes it easy or hard to implement certain features of a programming model. In fact, it isn't usually the individual instructions which are hard to implement, but the memory paradigm within which they act. For example, implementing a JVM requires a stack and a garbage collected heap to be provided. The natural course for the SVM is therefore to allow you to provide one of a variety of memory models, and let the SVM emulate the others.

Chapter 2

Preparation

This chapter introduces the area of virtual machines, and documents how I decided to approach the design of the SVM.

2.1 Existing work

Virtual machines themselves are not a new area of study. Systems as early as BCPL's O-code [1] (first designed in 1966) used the idea to separate the tasks of compiling programs and machine-specific implementation.

The JVM [2] has existed for some years, and is very widely used. It has many features which I will not be attempting to reproduce, like threading, remote method invocation and a vast library. It allows some degree of flexibility, by underspecifying some aspects of the behaviour, for example whether the threading is true multithreading or software.

The Sceptre extensible virtual machine [3] was a prototype system that allowed the programmer to add extra instructions to the VM. However, it was intended for the programmer to improve performance by controlling memory allocation or just-in-time compilation, rather than simplify the interpreter. It didn't allow instructions to be defined that used entirely different memory paradigms.

As far as I know there is no existing system which attempts to do the same as the SVM in terms of allowing incomplete implementations of the VM.

2.2 Project Planning

The task of designing the SVM requires a large number of small components. The dependencies between these are shown in Figure 2.1. I've split the

Figure 2.1: Project dependencies

tasks into those that would be distributable products, and the examples that depend on them.

2.3 The design requirements of the SVM

This project required me to take on different roles at different stages:

1. The designer of the flexible system
2. The designer of instructions and memory models within it
3. The producer of an interpreter

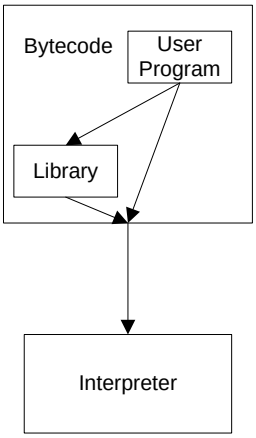


Figure 2.2: The structure of an SVM system

4. The programmer that writes for the system

The overriding design aim for the project is to make life easy for the third of these, the producer of the interpreter. This leads to the following components of a working SVM system:

- The interpreter
- The "library" part of the program, which contains emulation routines for instructions in case they are missing, and logic for choosing how to use them
- The program to be run

A key idea in this project is that I am abstracting what is required of an interpreter from the other two components, as shown in Figure 2.2. Of course, the other components are kept separate, but that's just good software engineering practice, not the purpose of the project. Importantly, the library and program are packaged together to be given to an interpreter.

2.3.1 Common execution model

It was necessary to have some common framework within which all the various paradigms I planned to implement could coexist. For example, trying to find a generalisation of both data-flow and control-flow programming styles wouldn't really get anywhere, as the representation of programs is so different.

I have chosen to use a sequence of instructions as my basic program format, with the intention that each instruction is executed in turn, except some "branch" instructions which can cause the program to continue to be executed from some other place. This is the most widely used and best understood program format, allowing me to concentrate on other parts of the project.

2.3.2 Instruction families

As I mentioned before, the SVM should be able to support multiple memory paradigms. To make this clear, I've designed the specification so that each instruction belongs to a *family*. Instructions within the same family should use the same memory.

2.3.3 Instruction design

While it is inevitable that some instructions will be necessary in every interpreter, the number of these should be kept to an absolute minimum. In particular, none of instructions that use a particular memory paradigm should be obligatory.

Importantly, the obligatory instructions should *not* be Turing-complete. The computational power of the SVM should be attainable by providing any of a wide variety of different sets of instructions.

2.3.4 Input and output

In order for a computer system to do anything practical, it must have a way to show the result of its calculations to the user. It is also helpful to be able to accept information as input from the user.

Since complex I/O isn't the purpose of the project, I only wanted to include a simple model for it. Programs should be able to input and output a byte at a time, which will normally be interpreted as an ASCII character.

2.4 Requirements for the tools

This project is mostly concerned with the interface between the bytecode and the interpreter. However, since it would be tedious to write bytecode manually, I decided to write an assembler tool to produce it in a flexible way from a more abstract source code.

I planned to provide a way to run programs written in Java within the SVM. Since the JVM bytecode is not flexible enough to be compatible with SVM bytecode, I also designed a tool to translate between them.

I decided to write these tools in Ruby (see 2.6.2.1).

2.4.1 Assembler

An assembly language is a human-readable way to represent a sequence of instructions. An assembler then converts this to bytecode. The assembler needed to support some features which are standard for assembly languages:

- Assigning integer values to a name (an alias) so that they can be used as memory location operands more readably
- Defining labels, and being able to use these as operands of branch instructions
- File inclusion directives, to allow code reuse
- Specifying literal operands in either decimal or hexadecimal

It was also useful for it to have some non-standard features which I decided would be useful for me:

- A way to define new instructions easily
- Identifying instructions by both their family and their individual identifier

An important part of making bytecode for the SVM will be organising the library of emulation routines. I decided to separate this job from the main assembler, but still write it in Ruby.

2.4.2 JVM bytecode translator

The requirements for the translator are straightforward. It needs to read in JVM bytecode (which is stored in a *.class* file) and output equivalent Salamander assembly language code.

However, to attempt to emulate the entire functionality of the JVM would take far longer than the time I had available, and would largely be irrelevant detail. As such, I have chosen a useful subset of the Java language to support and ensured that any JVM bytecode that is compiled from that subset can be translated.

Features that I have decided to support include:

- Integer and Boolean arithmetic
- Arrays
- All basic control structures (e.g. if, while, for)
- Input and output, one byte at a time

JVM *.class* files consist of a library of methods, and a pool of constants that they can use. I have had to learn the layout of the *.class* file, as well as the computational model which the JVM uses.

2.5 Instruction families

The instruction families I have defined are intended to serve as examples. I have chosen them because of their different memory paradigms, and specifically because they fitted a particular interpreter or the JVM translator. The core idea of the project is that new families should be easy to add to the system, so this relatively small sample of the area of memory paradigms should be seen as the first step to a growing collection.

- DMM32 – an architecture with a memory of 32-bit unsigned integers, where instructions act directly on memory locations
- STK32 – a stack machine using 32-bit signed integers, designed to be similar to JVM bytecode
- REG16 – a 16-bit unsigned machine with three special-purpose registers

See Appendix A for more details about the families used.

2.6 Approach to writing software

The nature of this project meant that I have had to write many medium-sized programs in various languages, rather than any single large piece of software. This, of course, makes the architecture of each program a less important consideration than the design philosophy used throughout.

2.6.1 Design principles

The techniques that saved me the most time during this project were:

- A version control system with a backup plan, meaning I would always keep the code clean rather than "commenting out" sections.
- Code reuse, for example between the assembler and the library generator
- Using high-level constructions to make modifications easier, for example:
  - Having the assembler dynamically read the definitions of instructions from a spreadsheet
  - Using constants for the numbers that identify instructions in the interpreters, rather than literals
  - Creating reusable blocks of assembly which are required by emulation routines, and using a directive to include them where needed

2.6.2 Languages used

There were a variety languages in which I wrote write significant pieces of software during this project, some of which I didn't know before, and some of my own invention.

2.6.2.1 Ruby

I already had experience in Ruby, making it the obvious choice for the tools. It has the following advantages.

- As a scripting languages, it has very natural access to files. This is important as the source assembly, the destination bytecode and JVM bytecode are all stored in files.
- It has powerful string manipulation, including regular expressions which make parsing simple assembly easy.
- It is a fully interpreted language, which means that the debug cycle is faster.
- The programs are not performance-critical, so the inefficiency of the language is not important.

- It supports richer high-level features than other scripting languages, for example Perl, making the software more maintainable.

2.6.2.2 Java

Again, I already had experience in Java. I used it for the general-purpose desktop interpreter, on which the rest of the system was tested.

2.6.2.3 C

I wrote the microcontroller implementation in C. I have no previous experience in C itself, but due to its close relation to other languages like Java, the learning process wasn't too bad. I also investigated MIPS assembler for the same purpose, but chose not to use it.

2.6.2.4 Verilog

Verilog hardware description language let me declare the design of a processor in a high-level way. I have a little experience in the language, but to implement an SVM required a lot of extra learning. This learning was hindered by differences between features supported by the Altera system and those described in many on-line resources.

2.6.2.5 Salamander assembly

As a language of my own creation, I didn't need to learn SVM assembly, as such. However, I needed to get used to the operations that could be performed in each instruction family to describe the instructions in terms of one another, and to write example programs.

The stack-based and special-purpose register families (STK32 and REG16) were particularly difficult to work with at first, as I had no experience in anything at all similar.

Chapter 3

Implementation

This chapter describes the the Salamander Virtual Machine, and the programs that are used to generate and run SVM programs.

Firstly, we decide how a valid SVM implementation should act, and specify the program format that it should run. This allows us to to write programs in the SVM format and, more importantly, write tools to generate programs from assembly language and Java. Finally, we write example interpreters to run these SVM programs.

This project provided opportunities for many interesting implementations throughout. Some of the highlights are:

- The emulation routine programmer interface (see 3.1.3.3)
- The binding sequence (see 3.2.2)
- Implementing DMM32 memory on a REG16 machine (see 3.3.2)
- The Verilog implementation (see 3.4.3)

3.1 The specification

The specification of the SVM, because it is designed to be so flexible, is very small, consisting only of:

- A description of the execution model
- A description of the program representation which an interpreter must decode
- Details of a small number of obligatory instructions

3.1.1 Execution model

As I described earlier, a Salamander program consists of a sequence of instructions, each of which comes from one of the defined families. They should be executed in order, except after some branch instructions, which can cause execution to jump to a new place.

3.1.2 The bytecode format

I decided to use a sequence of bytes to represent programs, for the practical reason that it is the ubiquitous data format of pretty much all devices at the moment. When a sequence of bytes contains a program for a VM it is traditionally called bytecode.

To express instructions within the bytecode, I had the requirement that the instructions could be of variable length, since different instructions could need very different numbers of operands (and could use different word lengths to represent them). In addition, it is necessary that interpreters can correctly deal with instructions which they don't recognise, in particular recognising their length so as to know where the following instruction starts. So, each instruction is represented by:

- The numbers that represent which instruction it is. This consists of a byte to identify the family, followed by a byte to identify the instruction within the family. Of course this limits us to 256 of each, but provides good performance.
- A byte to indicate the number of bytes of operand that follow
- The bytes of the operands

The structure of the operand bytes is deliberately not specified here, so that each instruction can interpret them however is most appropriate.

3.1.3 The meta-instructions

The Church-Turing thesis states that any computer can completely emulate any other. In particular, it would be possible to write complete SVM interpreters in a variety of minimal instruction sets. The purpose of this project is *not* to do this, as it would normally mean a high performance cost. Instead, the SVM should be able to flexibly emulate missing instructions while directly using the ones that the interpreter provides.

It is obvious that the ability to emulate instructions in terms of each other will need some extra capability to be built into the SVM. There are two possible approaches to achieve this:

- Providing a database of emulation routines and having the interpreter choose between them
- Defining special instructions (which I will call "meta-instructions"), which are mandatory in every interpreter, and suffice to allow emulation

I have chosen to use the second option, since it seems more exciting, and is less complex for the interpreter. A large reason for this is that the interpreter will already have the capability to run instructions in sequence, so adding a few extra instructions is simple. Also, it places the responsibility for deciding how to emulate each instruction, a difficult task, in the bytecode. Since all SVM instructions must belong to a family, I have defined a universal family of instructions to contain the meta-instructions, called UNI, which all interpreters must recognise.

I have identified that, in order to allow this type of emulation, three capabilities need to be provided by a meta-instruction:

- Specifying how to emulate an instruction (binding)
- Making decisions depending on instruction availability (jump if implemented)
- Accessing information about an instruction as it is emulated (get operand)

These meta-instructions are introduced below.

3.1.3.1 The *bind* meta-instruction

There must be a way to bind an unimplemented instruction to a location in the bytecode which should be run to emulate it. This will be used when the program starts running, to make a binding for every unimplemented instruction. When the instruction is encountered, the interpreter must jump to that location.

3.1.3.2 The *jump if implemented* meta-instruction

It is necessary to be able to query whether the interpreter supports a given instruction, then to act differently depending on the answer. An instruction which can achieve this is *jump if implemented*, which takes an instruction number and a location in the bytecode to jump to. There is a choice here which instructions are considered to be *implemented*:

- Only those natively implemented by the interpreter
- Those natively implemented, as well as those that have been bound

Both are enough to serve the purpose. The first option could, as a bonus, allow performance-critical programs to decide between implementations of algorithms depending on which instructions are native (so faster). Despite this, it is better to use the dynamically changing version, for implementation reasons explained in section 3.2.2.2.

3.1.3.3 The *get operand* meta-instruction

A routine to emulate an instruction must do exactly the same as the instruction itself. That means it reads the operands, acts on them, and then returns flow to the following instruction. To do this, there must be instructions which allow it to access:

- The operands of the instruction being emulated
- The program location of the instruction following the emulated one

I'll say *operand* to mean both the operands and program location during this description. For the sake of simplicity, I'll call the meta-instruction which reads this GETOP for now.

It is assumed that each emulation routine must fetch its operands before any other might be invoked. This assumption means that the interpreter only ever needs to hold one set of operands at once, avoiding the need to implement a stack in the interpreter, simplifying it greatly.

There are two very different ways that GETOP can act. The distinction between them is straightforward, but the reason to choose one over the other is subtle – I ended up changing from one to the other during implementation. I'll talk about both ways.

Retrieving the operand into the programmer's memory It seems intuitively sensible to write the operands to the same memory that the program is using for other data. Of course, the GETOP instruction would need to be aware of the memory paradigm, and as such a different version needs to be defined for each family. This doesn't appear to be a problem at first, until you consider binding an entire emulated family in terms of a native one, including a version of GETOP. If GETOP itself needs to be emulated, the assumption above, that GETOP only occurs before any emulated instruction, is violated. That means that operands are overwritten, causing errors.

Copying the operand into the bytecode Another way for the emulation routine to get the values is for GETOP to copy them into the running bytecode, into the operands of instructions later in the emulation routine. This is, in effect, a type of self-modifying code, and as such it would be normal to expect it to be confusing, or cause poor performance. In fact, this fairly restricted form of self-modifying code is easy to reason about, as you could interpret it as directions as to how to replace an unrecognised instruction with its emulation routine. Moreover, it doesn't suffer from the problem described above, and so is the solution I used.

In order to copy a value into an operand, there must be a way to give the location of the operand within the program to GETOP. This leads to the additional requirement for the assembler to be able to use a label definition as an operand (see 3.2.1.2).

3.2 Generating bytecode

Now that we have defined what Salamander bytecode should look like, we will discuss how to actually produce it. Of course, it would be possible to hand-code valid bytecode, and indeed I did so at early stages of testing. However, the following tools to automate the process are indispensable.

- The assembler
- The library generator which organises the emulation routines
- The JVM bytecode translator which allows the programmer to use a higher level language

Figure 3.1 shows how a program, either in Java or Salamander assembly language, is combined with the emulation routines to produce bytecode.

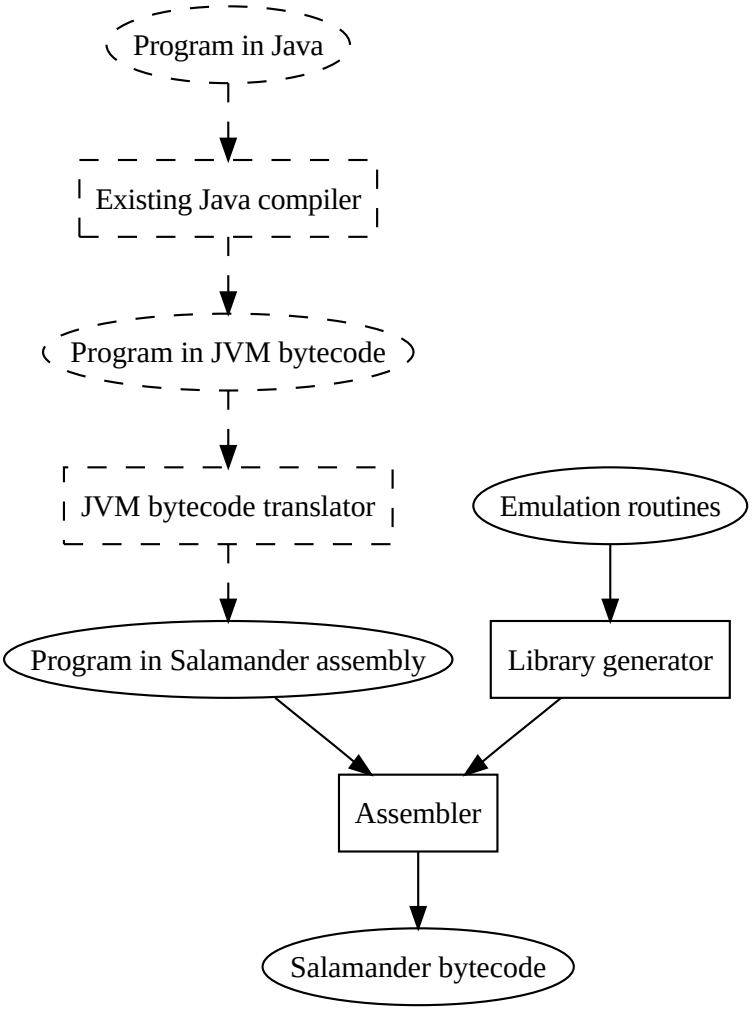


Figure 3.1: The compilation/assembly process

3.2.1 The assembler

The assembler performs a few important functions on which not only the programmer, but also the other tools, rely. Some of these are trivial to implement, but some deserve discussion.

- Understanding string representations of the instructions and constructing operands of the right bit-width
- Managing the definition and use of names, as a result of the ALIAS directive
- Defining and using labels so that branch instructions can point to another place in the code in a maintainable way
- Defining NAMESPACE regions, which limit the scope of both ALIAS names and labels
- Using the INCLUDE directive to assemble the content of another file into this one
- Outputting a valid bytecode file

Since this is solely a tool to help me during the development, I have defined the assembly language to be what the assembler will process.

See Figure 3.2 for an example fragment of code, which will introduce the features that I will describe. Its purpose is to emulate a bitwise AND instruction in terms of OR and NOT.

The first two lines are ALIAS definitions. The names X and Y are bound to a newly chosen integer, for which they will be substituted wherever they are used. The memory locations addressed by these integers are used to store intermediate values later in the program.

The lines beginning with a ' (quote) are interpreted to be comments and ignored.

Notice how each actual instruction begins with its family (e.g. UNI, DMM32) then states its instruction (e.g. OPCOPY, NOT) before listing its operands.

The first OPCOPY instruction instructs that 4 bytes of operands, starting at byte 0, from the emulated AND instruction should be copied to the location XADDRESS. The semantics of AND are that this operand is the memory address of the first argument to the bitwise AND function.

The first NOT instruction contains a label definition for XADDRESS, designated by the : (colon) after the name. This means that the space in the

```
alias x
alias y

'Get the operands
uni opcopy 0 4 xOperand
uni opcopy 4 4 yOperand

'Get where to store the result and where to return to
uni opcopy 8 4 resultOperand
uni epccopy 2 returnAddress

'Not both of them
dmm32 not xOperand: x
dmm32 not yOperand: y

'Do an OR
dmm32 or x y x

'Not the result
dmm32 not x resultOperand:

'Return to the previous location in the program
uni jimpl UNI:JIMPL returnAddress:
```

Figure 3.2: An implementation of AND in terms of OR and NOT

Family	Number	Name	Semantics	No. operands	Width 1	Width 2	Width 3	Width 4
DMM32	2	MUL	Dereferences and multiplies the first two operands, storing the result in the third. Overflows silently.	3	4	4	4	
DMM32	3	DIV	Dereferences and divides the first two operands, storing the quotient in the third and the remainder in the forth	4	4	4	4	4
DMM32	4	JMPEQ	Compares the first two operands and jumps to the third if they are equal (the address is absolute)	3	4	4	x	

Table 3.1: A fragment of the instruction spreadsheet

bytecode for this operand is initially left blank, and when the OPCOPY above is run, it is filled by the appropriate value.

Similarly, the JIMPL instruction, which is used to return once the emulation has been completed, has the old program counter copied into it by the EPCCOPY instruction.

3.2.1.1 Understanding the instructions

The definition of the instructions is stored in an XML spreadsheet to simplify adding new ones. The assembler parses this XML, to build a database of the details of instructions. This includes the number and bit-width of operands.

Table 3.1 shows a fragment of the input spreadsheet used in this process. The widths are specified in bytes, since operands must occupy entire bytes of the bytecode anyway. A width of x indicates that the operand has no fixed length, and is usually used for an address within the bytecode. The assembler may then choose whatever length is needed to store it. These can only be used for the last operand of an instruction, or the boundaries between operands would be ambiguous.

The full spreadsheet is available in Appendix A.4.

3.2.1.2 Resolving labels

Labels can be defined either before instructions or as an operand, in order to allow emulation routines to copy operands. In order to do this, we have

to make two passes over the code. Firstly, we need to construct the layout of the bytecode, leaving placeholders wherever a label is used, and making a note of the address of each label. Subsequently, we go back and fill in the value of each of the places a label is used.

3.2.2 The library generator

The designer of instructions has to provide implementations of their instructions in terms of others, which I call *emulation routines*. I have decided to have the bytecode choose how to bind each instruction (using the meta-instructions, see 3.1.3). This means a section of the bytecode must be generated to perform this, and to actually hold the emulation routines: I will refer to it as the *library*.

The emulation routines are kept in assembly form, and the library generator outputs the assembly for the library. This is so the task of resolving labels only needs to be done once, by the assembler.

Inlining the emulation routines themselves into the library is a trivial task; the difficulty is that most instructions have more than one emulation routine, so we need to choose which to use. This will depend on the instructions that are provided natively by each particular interpreter. We need an algorithm to make this decision, and to finish with an optimal set of bindings, so that each binding uses the minimum number of layers of emulation.

3.2.2.1 The design of the binding sequence

There is a spectrum of possible approaches to this task, running from the fully static (where the computation is done as the library is generated) to the fully dynamic (where it is done as the bytecode is being interpreted).

The static method would be to consider every possible combination of provided instructions statically and choose what to do in each situation. The dynamic method would read which emulation routines require which instructions at run time, and choose which to use to bind all the instructions.

Both of these methods are in fact unusable. The static version would take an intractable amount of time to do the static calculation, producing an unusably large library. The dynamic version seems a good idea, except that we don't know which instructions are available on which to run the algorithm until the algorithm is finished.

So the algorithm needs to run half-statically, half-dynamically. Loops over the instructions and emulation routines need to be unrolled statically, but branches depending on whether the interpreter implements a certain

instruction are left to be done dynamically. This means only the instructions that we know are always provided are used during the binding sequence.

3.2.2.2 The binding sequence implementation

Figure 3.3 shows the pseudo-code for the algorithm I used. It is an iterative approach, gradually building up the set of implemented instructions by only binding new emulation routines when all the instructions they use are implemented.

The lines in normal text correspond to lines of Ruby in the library generator, while the lines in italics correspond to lines of assembly that are emitted.

This approach relies on the *jump if implemented* instruction considering instructions which have been bound, as well as those provided by the interpreter, to be implemented. As discussed in 3.1.3.2, an alternative, and equally valid, semantic would be to only jump on natively implemented instructions. If we only had that type of *jump if implemented*, the binding sequence would need to statically navigate the tree of possible combinations of native instructions.

In fact I originally implemented that method, which lead to the binding sequence having a large bytecode size. To minimise this, I had implemented an algorithm based on a Markov chain, which determined the instructions most used in emulation routines and dealt with them first, so they didn't need to be checked again when considering other instructions. Unfortunately, even with this improvement, the bytecode eventually became larger than would fit on the hardware implementation, so I had to switch to the less interesting method.

3.2.3 The JVM bytecode translator

The largest part of the JVM translator is the code to parse Java *.class* files. Fortunately, Ruby's expressive power meant that I needed only one line of code to parse each element of the *.class* file to build up a data structure containing the bytecode. Due to the the *.class* format's complexity, this was still a significant task.

Once I had the bytecode for the MAIN method, it was a line-by-line translation to SVM bytecode. The bulk of the JVM instructions correspond to exactly one SVM instruction from the STK32 family (since that family was designed for the purpose, see Appendix A.2). Error checking was necessary to prevent use of the features of Java which aren't supported by the SVM. The output is an SVM assembly file.

```
DoAnotherLoop:

for each instruction
    if notImplemented(instruction)

        // Go through the emulation routines for this
        // instruction.
        for each routine in instruction

            // Check whether all the instructions that the
            // routine uses are implemented
            for each dependency in routine
                if notImplemented(dependency) then goto RoutineFails
            next

            // All dependencies are implemented - bind
            bind(instruction, routine)
            break

            RoutineFails:
        next
    endif
next

//See whether everything is now bound, and loop again if not
for each instruction
    if notImplemented(instruction) then goto DoAnotherLoop
next
```

Figure 3.3: Pseudo-code for algorithm to bind instructions

3.3 Emulation routines

Now we have discussed what Salamander bytecode looks like, and how a program is converted to bytecode, the reader may be interested to see the bytecode for a program, which can be found in Appendix D. A large part of the bytecode is occupied by the emulation routines, and much of the rest is the binding sequence, which is also generated from the emulation routines.

I have written three example families of instructions, and the emulation routines to translate between them (see Appendix A). Also I have written routines to implement individual instructions in terms of others in the same family.

3.3.1 Instructions within the same memory paradigm

Figure 3.4 shows a graph depicting some of the emulation routines which I have written between instructions of the DMM32 family.

While most of these emulation routines are fairly obvious implementations of well-known equivalences, a couple were more interesting.

3.3.1.1 DMM32 ADD

In the event that an interpreter provides logical and shifting instructions, but no arithmetic, the rather elegant algorithm for addition in Figure 3.5 is used. Of course this isn't a very likely situation in computers at the moment.

3.3.1.2 DMM32 DIV

The most complex of the instructions to emulate was the division. It uses a binary long-division algorithm, the full code of which can be found in Appendix B.1.

3.3.2 Implementing one memory paradigm in another

There is much more scope for complex implementation where the emulation routines are providing a memory which isn't provided by the interpreter.

3.3.2.1 STK32 ADD implemented using DMM32

This is a simple example of how the stack machine family STK32 is implemented over the three-address family DMM32. Figure 3.6 gives the complete code of the emulation routine which will serve as a useful introduction to the ideas used for all the emulation routines.

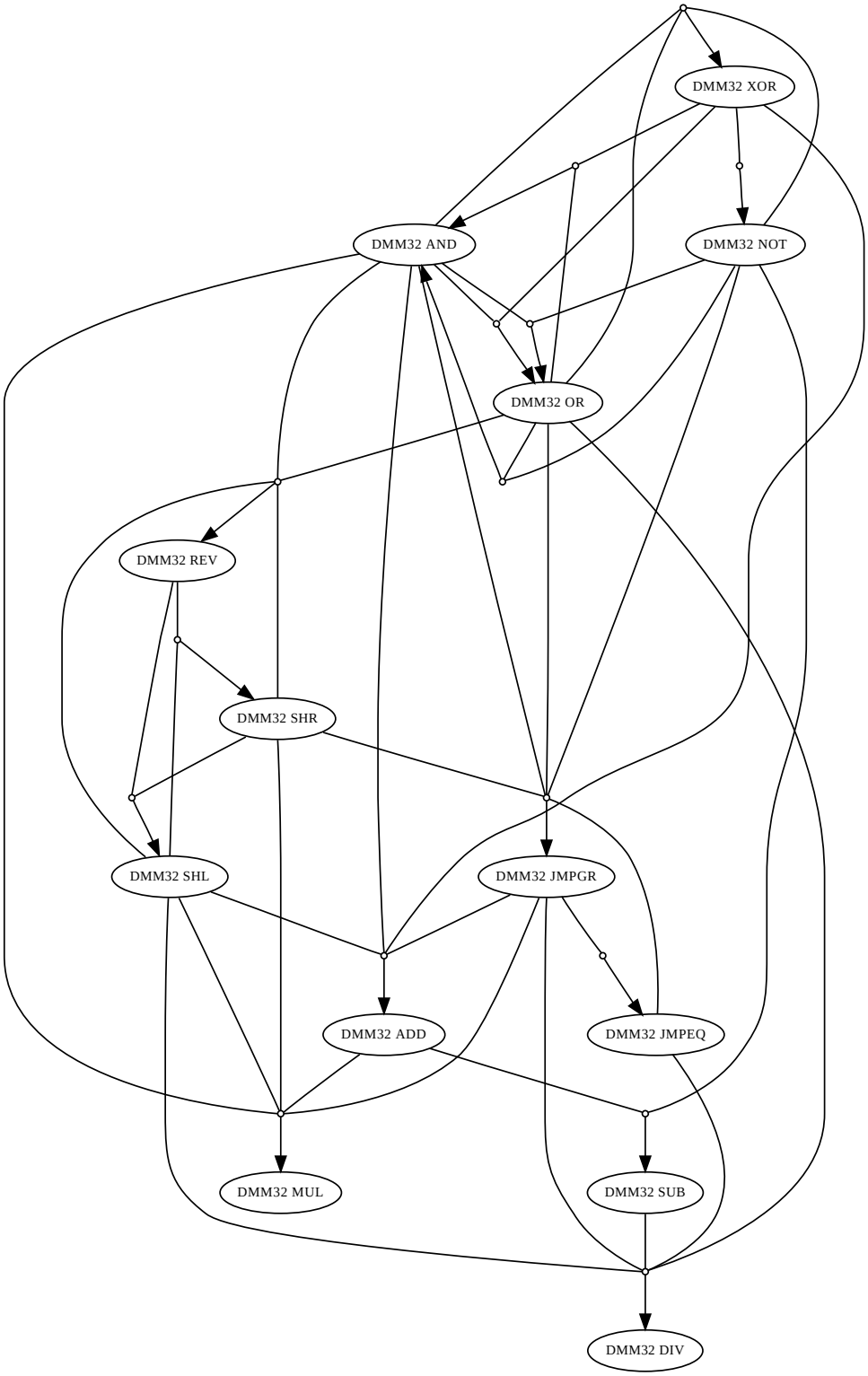


Figure 3.4: Emulation routines available between instructions of DMM32

```
repeat
    newX = x xor y
    y = (x and y) << 1
    x = newX
until y becomes 0
```

Figure 3.5: Pseudo code for addition from bitwise logic

```
Emulate Stk32 ADD
StartNameSpace
    alias x
    alias y

    uni epccopy 2 returnAddress
    dmm32 load sp x
    dmm32 sub sp one sp
    dmm32 load sp y

    dmm32 add x y x

    dmm32 store x sp
uni jimpl UNI:JIMPL returnAddress:
EndNamespace
EndEmulate
```

Figure 3.6: STK32 ADD implemented using DMM32

The first and last line define the scope of the emulation routine, and are read by the library generator. The body of the routine is not modified by the library generator, and is passed as a unit to the assembler.

The NAMESPACE lines define the scope of the aliases X and Y. The other alias used here, SP (stack pointer), is defined at a wider scope. It is used to store the address of the current top of the STK32 stack within the DMM32 address space. Being an alias at a wider scope, its value persists between all the emulation routines.

The actual calculation performed, even in this simple case, is dominated by maintaining the emulated memory structure (in this case, decrementing the stack pointer and reading and writing the values in the stack). This becomes even more apparent where the memory is harder to emulate, as we will see next.

3.3.2.2 DMM32 implemented using REG16

All of the emulation routines that provide DMM32 instructions using the very limited REG16 instruction family (see Appendix A.3) are too large to include here, but I will discuss some of the difficulties encountered.

The REG16 family only has three special-purpose registers, which means that most operations require a lot of moving values between registers and to and from memory. This limitation makes even simple tasks difficult to implement.

The smaller word size means that the memory space addressable in REG16 is much smaller than the DMM32 memory space (128kB instead of 16GB). This limitation means that I had to use 4 levels of paging, with each page table taking only 256 words of memory. Each level of page is addressable using one byte, totalling the 32 bits that form a DMM32 address.

Also, the values that can be stored in the DMM32 memory are 32 bits wide rather than just 16, so all the operations need to operate on the low and high words separately. If there is a dependency between them (for example a bit is carried from the low to high word of an addition) this needs to be remembered.

I made heavy use of the INCLUDE directive of the assembler to reuse the code which reads and writes the emulated memory.

3.4 Example interpreters

SVM bytecode contains emulation routines. The bytecode will run on an SVM interpreter with missing instructions, provided that the emulation rou-

tines can fill them in.

I have written three different example interpreters, each with a different purpose.

- A Java implementation to run on desktop computers
- A C implementation to run on a MIPS microcontroller
- A Verilog (hardware) implementation

3.4.1 The desktop implementation

The primary requirement for the desktop implementation was that it could be maintained easily, for example, in case I needed to change the specification. As the first interpreter, it needed to serve as the test ground for all the other parts of the project.

3.4.1.1 Basic features

I decided to write it in Java, for the good balance of speed and maintainability. The instruction family it provides is DMM32 (see Appendix A.1), as well as the obligatory meta-instructions. The bytecode is loaded from a file into a byte array, from which it is run. The basic structure of the program is predictable – a WHILE loop, containing code to read an instruction followed by large switch-case statements which do the appropriate action depending on the instruction.

The memory (being too large to use a single array) is implemented using a three-level paging system. Each page is only created when it is first written to.

3.4.1.2 Debug features

Since this interpreter is the used to test the rest of the project, it was important that it could allow debugging. To do this, it has a tracing mode, where it outputs the details of everything it does. Of course, when the program being run becomes complex (especially when multiple nested emulation routines are involved) this becomes very verbose. This is a fundamental limitation of this project, even providing a fully-featured step-through debugger would suffer from the layers of emulation, making a single instruction require many steps of the debugger.

The best solution I found was to modify the interpreter to programmatically detect the conditions that were suspected to result from a bug.

3.4.1.3 Testing features

I wrote the implementation of every DMM32 and universal instruction for this interpreter, but it was important to be able to change which instructions were provided so that various emulation routines could be tested. To do this, I included a system so the program could modify which instructions it claimed to provide.

This system later became useful for testing the performance of the emulation routines, by allowing a way to systematically try combinations of instructions.

3.4.1.4 REG16 version

I also adapted a copy of the Java interpreter to use the REG16 instruction set instead. This was used to test the REG16 emulation routines, before they were used in the hardware implementation. This proved a very useful endeavour, as debugging on the desktop is much faster than on the hardware implementation.

3.4.2 The microcontroller implementation

There is a soft-core MIPS implementation for the Altera DE2 board, developed in the Cambridge Computer Lab, called "Tiger". I have implemented an SVM interpreter in C for use on this processor.

Due to the similarity of C and Java, it was a relatively simple procedure to translate the bulk of the interpreter for use on the board. However, there were difficulties with the tasks that were previously easy.

Loading the bytecode onto the board was a challenge. The eventual solution of compiling it as a constant into the interpreter, and uploading both to the board together, was less elegant than I would have liked. However, I did automate the procedure and use a C INCLUDE directive to get the bytecode from a separate file, meaning that the details were hidden from the user of the system.

I implemented the input and output through a terminal connection to a normal computer. This required translating the routines for this from the MIPS assembler, provided with the board, to C.

3.4.3 The hardware implementation

The final SVM implementation I have written is for a Field Programmable Gate Array (FPGA) system on an Altera DE2 board. Verilog is a hardware description language, which means that it is high-level, but can be translated

directly into hardware, or to the layout of an FPGA. The available operations include basic manipulation of numbers of various bit-widths, as well as assigning to registers at clock edges.

This implementation provides the REG16 instruction family, as well as the universal family.

3.4.3.1 Core implementation

Performance was not a priority for this implementation – it is simply a proof-of-concept. Each instruction takes multiple clock cycles to complete, as each of the following phases is completed in one cycle:

1. Reading the family number
2. Incrementing the program counter so it points to the instruction identifier
3. Reading the instruction number
4. Incrementing the program counter so it points to the number of operands
5. Reading the number of operands
6. Incrementing the program counter for each operand
7. Reading an operand
8. Performing the operation

The 6th and 7th phases are repeated as many times as there are operands. The 8th phase may take multiple clock cycles, depending on the instruction being performed (for example the meta-instructions which write to the bytecode need a clock cycle for every byte they write).

Using a hardware description language forced me to use a programming style which wasn't ideal. Lines of Verilog have to be separated into those that update registers, and those that hold values on a wire (for example a data line of a memory). This means that pieces of code that occur at the same time for the same reason are separated.

3.4.3.2 Memory

To provide the memory required for the user of the REG16 instruction set (a 16-bit addressable, 16-bit wide memory), I used an SRAM chip provided by the DE2 board. Because there is only one register that is used to address memory in the REG16 instruction family, that register could be connected permanently to the address line of the SRAM. This made reading as simple as latching the contents of the data line, and writing only needed the "write enable" bit to be set for a clock cycle.

The memory in which the bytecode is stored, and the memory in which the bindings between instructions and their emulation routines are stored, were a little more complex. For them, I used "megafunction" memories, which are on the FPGA itself.

3.4.3.3 Input & Output

The DE2 board has, among other peripherals, a two line LCD display and a PS2 port to which a keyboard can be attached. These were the most appropriate devices to the model of I/O that the SVM uses.

I adapted some examples for the use of both the LCD display and PS2 keyboard that are provided on-line.[1]

I wrote a controller for the LCD which displayed the characters which the program output on the lower line of the LCD. The line feed character causes the contents of the lower line to move to the top line, for a scrolling effect.

In order to receive useful input from the keyboard, I needed to translate from keyboard scan codes to ASCII, and synchronise the transfer characters with the execution of input instructions by the interpreter.

3.4.3.4 Debug features

Thanks to the use of the REG16 version of the Java implementation, and the unit tests (see 4.1.1), there were relatively few issues that needed to be investigated on the device itself.

For those that remained, I used the DE2 board's 7-segment displays to show the values of registers within the processor. I could change which value I was inspecting using the switches on the board. Another of the switches allowed me to pause the execution of the program, to step through it, ensuring the registers had the correct values.

[1] Thanks to Dr John S Loomis of the University of Dayton, OH for the examples http://www.johnloomis.org/digitallab/

Chapter 4

Evaluation

The goals of this project were achieved and surpassed. This chapter describes how I have verified that the project worked, and how well it does so.

4.1 Testing for correctness of implementation

All components of this project have been tested thoroughly to ensure they work correctly. I have tested them using both unit tests and as a system.

4.1.1 Unit tests

For each instruction family, I wrote an assembly program that exercised all the instructions in as many ways as I could think of. For example, the unsigned division instruction was tested with:

- Small numbers that divide exactly
- Large numbers that divide exactly (large meaning that the number couldn't be represented in signed 32-bit arithmetic, and so any errors with the unsigned interpretation would show).
- Small and Large numbers that produced a remainder
- With a smaller numerator than denominator
- With zero as the numerator

When I discovered a bug in an interpreter or emulation routine, I would add it to a unit test so that it couldn't happen in any other.

These were perhaps the most useful tool I had during the project. While usually a unit test is only useful to ensure that one program works, I could use these to test all the interpreters and emulation routines.

To test the routines which emulate instructions within the same family, I used the desktop interpreter (see 3.4.1). It allows me to toggle whether each instruction is provided. I could systematically choose sets of instructions to make sure that every emulation routine was exercised by the unit test.

Testing the routines that provide an instruction family from another was necessarily more ad-hoc, as I needed to use other interpreters as well as the desktop one. Despite this, the nature of the project meant that any one test provided a high level of confidence. This is because running a unit test for the STK32 family on REG16 interpreter showed that DMM32 was also correct (see Appendix A.1). In fact, this technique of using multiple levels of emulation in testing sometimes identified deficiencies in the unit tests.

4.1.2 System tests

It would have been inappropriate to unit test the tools written for this project since each of them fulfilled a large number of simple requirements. The unit test would have been as verbose as the tool itself. Instead, I tested the tools by using them as part of the system.

The assembler was tested by processing the very large volume of code that the library generator produced, as well as the example programs. The fact that the assembly language was defined by what the assembler would accept means also helped.

The library generator was used as the number of emulation routines grew, producing a working library at each stage.

The java translator was used on multiple JVM bytecodes. I have a high confidence that the code to parse the *.class* file format is correct, since it is relatively simple. The translation of the bytecode is probably the least well tested component of the system, as I had no way to systematically induce the Java compiler to use all the possible JVM instructions that are supported.

4.2 Testing the performance of the system

Since the highest-performance SVM interpreter is itself written in Java, there is no way to make a useful comparison between the performance of the SVM and the existing JVM. I am confident, however, that there is no reason that it would be any slower when the instructions being used are provided natively

```
dmm32 imm 1234567854 alpha
dmm32 imm 234567 beta

Loop:
    dmm32 add alpha beta gamma
    dmm32 sub alpha beta gamma
    dmm32 mul alpha beta gamma
    dmm32 div alpha beta gamma delta
    uni break 'This instruction does nothing, but is counted
uni jimpl UNI:JIMPL Loop:
```

Figure 4.1: The integer arithmetic benchmark (Salamander assembly code)

by the interpreter, if the same technology were used (for example just-in-time compilation).

4.2.1 Emulation within the same family

That leaves the interesting question of how much slower the system becomes when the instructions used by the program are not provided by the interpreter, and are instead emulated. For the purpose of testing this, I have written a system to measure the performance of a program given a certain set of provided instructions. It counts the number of times a special instruction is run in a given length of time.

Integer arithmetic has the most interesting emulation characteristics of all of the tasks that the SVM can perform, since all four operators can be emulated, or not emulated, independently. I have used a benchmark that repeatedly uses all four arithmetic operators. The code of the benchmark is given in Figure 4.1. The results are shown in table 4.1. The result represents the number of times the operators can run in a second on my otherwise unloaded machine. Averages were taken appropriately, and results are given to 2 significant figures.

Of course it would be impractical to test all combinations, so I have restricted myself to testing each type of instruction (bit-shifting, logic, comparison then arithmetic) then leaving them implemented for the rest of the tests. There are some informative general patterns that these results give.

The instructions often come in pairs which perform dual roles, and typically one of them is needed for the interpreter to be Turing-complete. Such pairs include:

- *Shift left* and *shift right* (SHL and SHR). Also *reverse* (REV) is related

ADD	SUB	MUL	DIV	JMPEQ	JMPGR	SHL	SHR	REV	OR	AND	XOR	NOT	Operations/s
				✔			✔	✔	✔			✔	32
				✔		✔		✔	✔			✔	34
				✔		✔	✔		✔			✔	36
				✔		✔	✔	✔	✔			✔	37
				✔		✔	✔	✔		✔		✔	39
				✔		✔	✔	✔	✔	✔		✔	62
				✔		✔	✔	✔	✔	✔	✔		62
				✔		✔	✔	✔	✔	✔	✔	✔	93
					✔	✔	✔	✔	✔	✔	✔	✔	290
				✔	✔	✔	✔	✔	✔	✔	✔	✔	300
✔				✔	✔	✔	✔	✔	✔	✔	✔	✔	3000
	✔			✔	✔	✔	✔	✔	✔	✔	✔	✔	3200
✔	✔			✔	✔	✔	✔	✔	✔	✔	✔	✔	6200
✔	✔	✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	12000
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	44000

Table 4.1: Benchmark results of combinations of provided instructions

to these.

- AND and OR
- XOR and NOT
- *Jump if equal* and *jump if greater than* (JMPEQ and JMPGR)
- ADD and subtract (SUB)

The results show that in general, which one of each pair is provided makes very little difference to performance. However, providing both of them tends to improve performance by a much larger amount.

The exception to this is the JMPEQ and JMPGR pair. Here we see that JMPGR is much more important to performance. This is because emulating JMPGR requires a routine that scans through the bits of the inputs, whereas a JMPEQ can be implemented using only two JMPGR instructions.

The overall performance difference between implementing the minimum number of instructions and all of them is roughly one thousand. This sounds a lot, until you compare this to an interpretation-based solution that also builds division and multiplication from the minimum set of instructions. A

Family Used	Operations/s
REG16	35000
DMM32	44
STK32	27

Table 4.2: Benchmark results of inter-family emulation on a REG16 interpreter

Family Used	Operations/s
DMM32	64000
STK32	3100

Table 4.3: Benchmark results of inter-family emulation on a DMM32 interpreter

more likely use would be to emulate division on an architecture that doesn't provide it, and this testing shows a speed decrease of only four times, which is respectable considering the complexity of division in software and the number of division operations the benchmark used.

4.2.2 Emulation between families

In order to measure the amount of performance lost by running a program in a family that is not natively provided, we need to perform the same calculations on the native family and on the emulated family. This isn't as easy as it sounds, due to the vast differences of computational model between the families (notably that REG16 only has 16-bit integers and no division).

So I have used a benchmark that can be performed on all the families: running a repeated subtraction version of Euclid's algorithm on four digit numbers. Even this was fairly difficult on REG16 since there is no instruction to compare the size of two numbers. The results, again averaged appropriately, are shown in tables 4.2 and 4.3.

The first test was performed on the REG16 desktop implementation, and the second on the DMM32 desktop implementation. The score is the number of times the algorithm could run in a second.

The most marked performance loss is in emulating the 32-bit families in terms of the 16-bit one. This is unsurprising, because of the enormously costly paging system used to access the emulated 32-bit memory.

The most surprising result is that the performance loss caused by emulating STK32 on DMM32 is much less when DMM32 is already being emulated than it is on a native DMM32 interpreter (a factor of 2 rather than 20). This

may be because the meta-instructions that make up the inefficiency in the directly emulated case are relatively cheap in the two-level case, since they are still provided natively by the interpreter.

4.3 Testing the capabilities of the system

Here I will discuss two uses of the SVM to do something that would be much more difficult without it.

- Running an interactive Java program on an SVM computer I implemented in hardware
- Running a Java SVM interpreter within the SVM

4.3.1 Interactive Java program on hardware

This is a demonstration of the way that the SVM allows useful programs to be run on new devices with little effort.

I have written a pocket calculator emulator in Java. It, of course, features the four arithmetic operators, which can operate on integers typed into the program. Some other keys perform a selection of useful functions for the computer scientist:

- Highest prime factor (F)
- Euler's totient function (T)
- Highest common factor (H)

The Java code that calculates the highest common factor is given in Figure 4.2. This is translated to SVM bytecode by the JVM bytecode translator.

Using the hardware implementation to run this calculator gives a portable calculator. This set-up is shown in Figure 4.3, where I have typed the number 912, then pressed the T key to find its totient, 288.

4.3.2 SVM interpreter running on the SVM

The JVM translator allows Java programs to be translated to SVM bytecode. The desktop implementation of the SVM is written in Java. The obvious challenge is to run the interpreter in itself.

Unfortunately, the main desktop implementation (see 3.4.1) is too complex to translate to SVM bytecode, because of its debug and performance

```
// Compute the highest common factor (Euclid)
while (a != b) {
    //Sneaky way to do 1-based mod using 0-based mod
    if (a > b){
        a = ((a-1) % b) + 1;
    } else {
        b = ((b-1) % a) + 1;
    }
}
```

Figure 4.2: The highest common factor routine from the pocket calculator

Figure 4.3: Photograph of the DE2 board running the pocket calculator

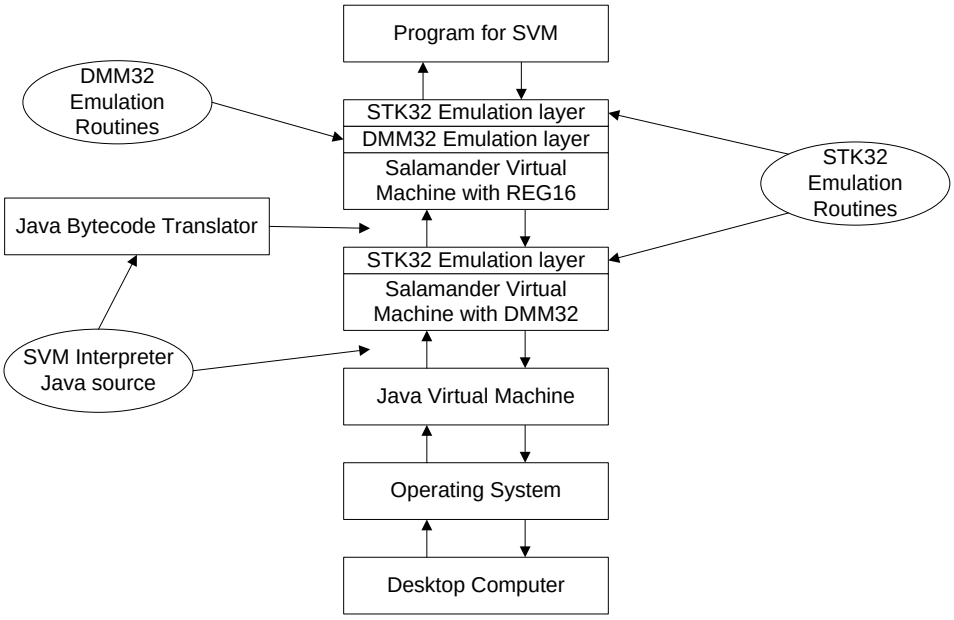


Figure 4.4: Running an SVM interpreter in the SVM

testing features, as well as the 32-bit memory being implemented using object orientation.

However, the REG16 version (see 3.4.1.4) is simpler, and can be translated to SVM. It runs in the main desktop implementation, and can successfully pass the unit test for each of the instruction families. Unfortunately, the performance is decreased by a factor of roughly 100000, meaning that no useful programs can be run in a sensible time.

The route that the computation takes is depicted in Figure 4.4.

4.3.3 Conway's Game of Life

As an additional example program, I wrote an implementation of John Conway's cellular automaton "Game of Life" [4]. It, of course, allows the full set of interesting patterns, including the "Gosper glider gun" shown in Figure 4.5. It is written in Java and translated to SVM bytecode, so it runs on all SVM implementations (although the LCD screen used by the hardware implementation can't show enough of the output at once to be meaningful).

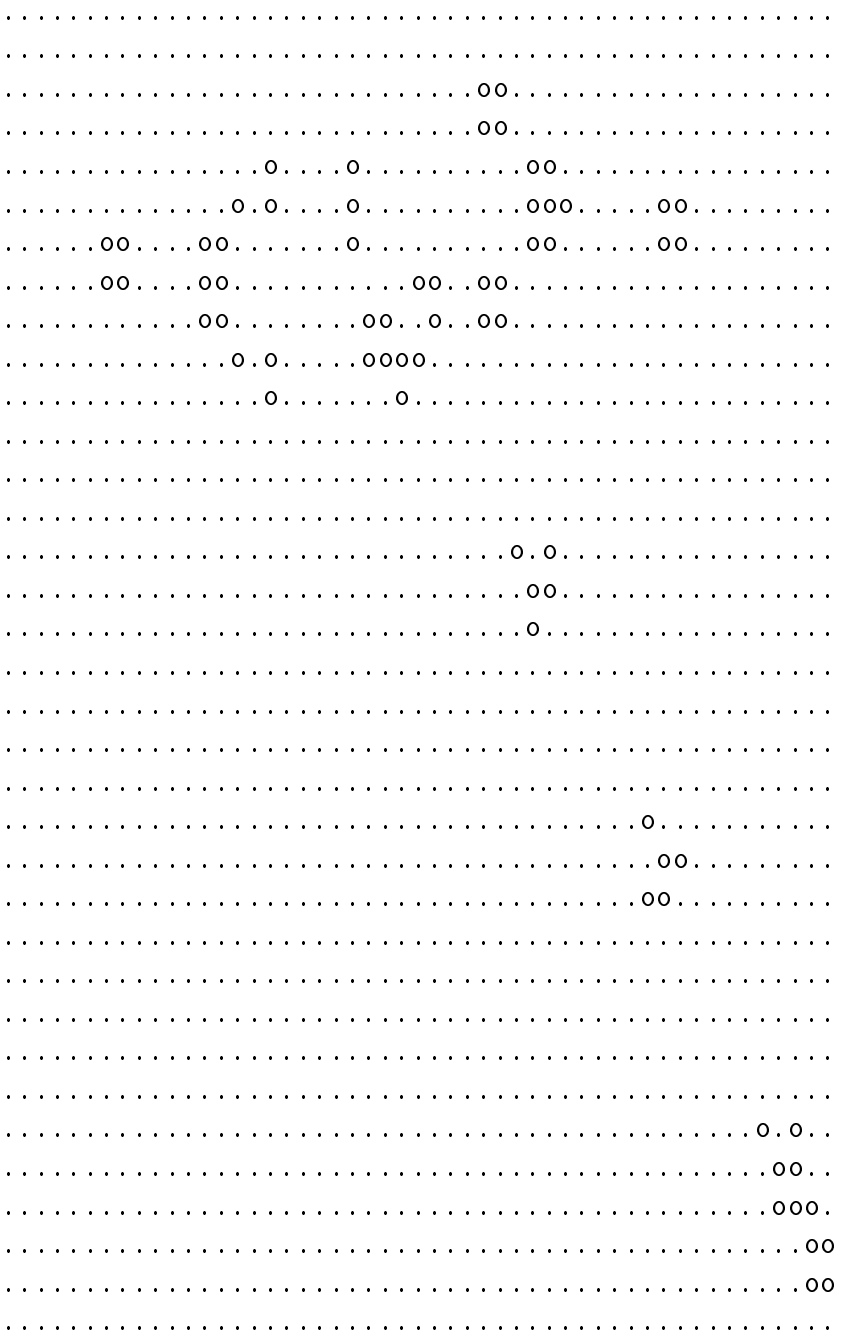


Figure 4.5: One output iteration of the implementation of Conway's Game of Life

4.4 Limitations

As the above tests have shown, the system doesn't seem to be limited in what emulations are possible, only in how quickly it will run them. This is justifiable, given that other systems that did the same by means of interpretation would be even slower.

Of course, I could have written more instruction families, interpreters and example programs, and perhaps would have learnt more from doing so. This could have included floating-point arithmetic, support for exceptions, or a more powerful I/O system. However, there was only limited time, and it was more important to concentrate on the core of the project.

One way to improve the performance of the system fairly dramatically would be to include a form of just-in-time compilation (JIT) in an interpreter. It would have allowed the emulation routines for unrecognised instructions to be copied directly into the program in the place of the unrecognised instruction. This would eliminate the time spent:

- Branching to the emulation routine
- Performing the meta-instructions to copy the operands into the emulation routine
- Branching back to the program code

This would bring the project a level of performance similar to a static translation system, while keeping the dynamic flexibility that is the core of the SVM.

Even with these limitations, the SVM is a powerful system that could easily be used for programming devices. When it is being used in a way that other VMs can be used, it will perform equally well, while it can also be used to allow previously impossible flexibility with reasonable performance.

Chapter 5

Conclusion

This project concerned the creation of the Salamander Virtual Machine, which allows any program to run on an SVM implementation that only provides a small portion of the features used.

The project was a complete success, with the goal of running Java programs on a microcontroller device being achieved, and surpassed by running them on a hardware implementation of the SVM as well.

The project fulfils its design objective to make it easy to implement an SVM interpreter. This is shown by the size of the smallest interpreter, only 160 lines of Java. A guide to producing an interpreter is included in Appendix C.

As far as I know, the SVM breaks new ground in emulation of programming paradigms: with flexibility that matches interpretation, at efficiency that comes close to translation. I hope that it provides the foundation for further study, taking forward some of the exciting new ideas that have been such an enjoyable challenge to investigate.

Bibliography

[1] Martin Richards. *The Portability of the BCPL Compiler.* Software - Practice and Experience, Vol. 1, No. 2, pp.135-146, 1971

[2] Tim Lindholm, Frank Yellin. *The Java^{TM} Virtual Machine Specification (Second Edition)* 1999

[3] Timothy Harris. *An extensible virtual machine architecture*, Proceedings of the OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, 1999

[4] Martin Gardner. *The fantastic combinations of John Conway's new solitaire game "life"*, Scientific American 223, pp.120-123, 1970

Appendix A

Instruction families

The three families I provided are implemented in terms of each other as shown in Figure A.1.

The dotted arrow indicates an implementation that was included for completeness, but could not be tested thoroughly since no STK32 interpreter was written.

A.1 Direct memory manipulation "DMM32"

This family has a 32-bit wide, unsigned, 32-bit addressable memory space on which its instructions can operate. Its arithmetic operations largely use three memory addresses as operands, two sources and a destination. This means that there are no registers in the architecture.

It is designed to be a general-purpose family which is easy to implement on a desktop machine, and which is easy to implement other families on top of. The reason I chose to make it unsigned was as a challenge more than anything else.

A.2 Stack machine "STK32"

This family has an unlimited stack of 32-bit wide signed integers on which to operate. The stack can also hold references to arrays which can be created on a heap. Operations tend to occur by taking values from the top of the stack, doing something with them, then returning the result to the top of the stack. The family is designed to be close to the paradigm used by a subset of the JVM, so that JVM bytecode can be easily translated to SVM bytecode using STK32.

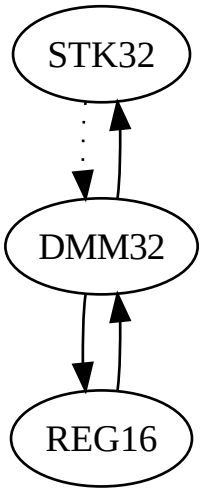


Figure A.1: Inter-family emulation graph

A.3 Special-purpose register machine "REG16"

This family is designed to be tiny and very easy to implement. It was made to be used with the hardware implementation (see 3.4.3). It has three registers, all of which have a special purpose:

- An accumulator, which gets the result of all operations, and acts as the first operand of all operations
- An index register which acts as the memory address for all memory accesses
- A second operand register

It also has a 16-bit wide, 16-bit addressable memory space. The reason that it is 16-bit rather than 32-bit is partly because the Altera DE2 board on which the example hardware implementation was produced has a large 16-bit SRAM available. It is also partly to give me the opportunity to demonstrate implementing one memory size in terms of another (see 3.3.2.2).

A.4 Instruction spreadsheet

Below is the complete listing of instructions that I defined for the SVM.

Family	Number	Name	Semantics	Operands	No. Bytes			
UNI	0	OUT	Writes a byte to the output (as ASCII probably)	0				
UNI	1	IN	Takes a byte from input if one is waiting (polled) and puts it in the accumulator	0				
UNI	2	BIND	Ties the given instruction to the code at the address	3	1	1	x	
UNI	3	JIMPL	Jumps if an instruction is implemented natively	3	1	1	x	
UNI	4	JNIMPL	Jumps unless an instruction is implemented natively	3	1	1	x	
UNI	5	OPCOPY	Takes op2 bytes from the op1 of emulated operands and places it in op3 in the program	3	1	1	x	
UNI	6	EPCCOPY	Takes op1 bytes of the emulated program counter and places it in op2 in the program	3	1	x		
UNI	7	BREAK	No defined meaning, used for testing	0				
DMM32	0	ADD	Dereferences and adds the first two operands, storing the result in the third. Overflows silently.	3	4	4	4	
DMM32	1	SUB	Dereferences and subtracts the first two operands, storing the result in the third. Underflows silently.	3	4	4	4	
DMM32	2	MUL	Dereferences and multiplies the first two operands, storing the result in the third. Overflows silently.	3	4	4	4	
DMM32	3	DIV	Dereferences and divides the first two operands, storing the quotient in the third and the remainder in the forth	4	4	4	4	4
DMM32	4	JMPEQ	Compares the first two operands and jumps to the third if they are equal (the address is absolute)	3	4	4	x	
DMM32	5	JMPGR	Compares the first two operands and jumps to the third if A>B (the address is absolute)	3	4	4	x	
DMM32	6	SHL	Shifts the first operand left by the low 5 bits of the second operand, filling with 0	3	4	4	4	

Family	Number	Name	Semantics	Operands	No. Bytes			
DMM32	7	SHR	Shifts the first operand right by the low 5 bits of the second operand, filling with 0	3	4	4	4	
DMM32	8	REV	Reverses the operand	2	4	4		
DMM32	9	OR	Bitwise OR	3	4	4	4	
DMM32	10	AND	Bitwise AND	3	4	4	4	
DMM32	11	XOR	Bitwise XOR	3	4	4	4	
DMM32	12	NOT	Bitwise NOT	2	4	4		
DMM32	13	COPY	Copies SRC to DEST	2	4	4		
DMM32	14	LOAD	Dereferences SRC, then copies its contents to DEST	2	4	4		
DMM32	15	STORE	Copies SRC to the location pointed to by DEST	2	4	4		
DMM32	16	IMM	Stores an immediate (first operand) to memory (second operand)	2	4	4		
DMM32	17	ACCSET	Copies a number into the accumulator (for output) Truncates to low 8 bits	1	4			
DMM32	18	ACCGET	Copies the accumulator into where the operand point2	1	4			
STK32	0	CONST	Pushes the specified constant	1	4			
STK32	1	LOAD	Pushes the value at the specified location in the stack	1	1			
STK32	2	ALOAD	.., arrayref, index => value Reads from an array	0				
STK32	3	STORE	Pops a value and writes it to the specified location in the stack	1	1			
STK32	4	ASTORE	...arrayref, index, value => .. Writes a value to an array	0				
STK32	5	POP	Pops a value and throws it away	0				
STK32	6	DUP	.. A=> .. A, A Duplicates the top of the stack	0				
STK32	7	DUPx1	..B, A=> .. A, B, A Duplicates the top of the stack to 1 down	0				
STK32	8	DUPx2	.. C, B, A=> .. A, C, B, A Duplicates the top of the stack to 2 down	0				

Family	Number	Name	Semantics	Operands	No. Bytes			
STK32	9	SWAP	..a, b => ..b, a Swaps the top two of the stack	0				
STK32	10	ADD	.. a, b => a + b Addition - overflows silently	0				
STK32	11	SUB	.. a, b => a - b Subtraction - underflows silently	0				
STK32	12	MUL	.. a, b => a x b Multiplication - overflows silently	0				
STK32	13	DIV	.. a, b => a / b Integer division	0				
STK32	14	REM	.. a, b => a % b Remainder (always positive)	0				
STK32	15	NEG	.. a => .. -a Negate the top of stack	0				
STK32	16	SHL	.. a, b => a << b	0				
STK32	17	SHR	.. a, b => a >> b	0				
STK32	18	USHR	.. a, b => a >>> b	0				
STK32	19	AND	.. a, b => a & b Bitwise and	0				
STK32	20	OR	.. a, b => a | b Bitwise or	0				
STK32	21	XOR	.. a, b => a ^ b Bitwise xor	0				
STK32	22	IFEQ	.. a, b => ... Jump if a=b	1	x			
STK32	23	IFNE	.. a, b => ... Jump if a!=b	1	x			
STK32	24	IFLT	.. a, b => ... Jump if a<b	1	x			
STK32	25	IFLE	.. a, b => ... Jump if a<=b	1	x			
STK32	26	IFGT	.. a, b => ... Jump if a>b	1	x			
STK32	27	IFGE	.. a, b => ... Jump if a>=b	1	x			
STK32	28	GOTO	Unconditional jump	1	x			
STK32	29	NEWARRAY	.. Count => .., arrayref Creates a new array on the heap	0				
STK32	30	ARRAYLENGTH	.. Arrayref => ... length Gets the length of an array	0				
STK32	31	ACCSET	Pops the top of the stack and puts its low 8 bits into the accumulator	0				

Family	Number	Name	Semantics	Operands	No. Bytes			
STK32	32	ACCGET	Pushes the accumulator onto the stack (without sign extending it)	0				
REG16	0	LDA	Load a constant into A	1	2			
REG16	1	LDB	Load a constant into B	1	2			
REG16	2	LDC	Load a constant into C	1	2			
REG16	3	LDI	Load indirectly A := *B	0				
REG16	4	STI	Store indirectly *B := A	0				
REG16	5	TAB	Copy the value of A into B	0				
REG16	6	TAC	Copy the value of A into C	0				
REG16	7	TBA	Copy the value of B into A	0				
REG16	8	TBC	Copy the value of B into C	0				
REG16	9	TCA	Copy the value of C into A	0				
REG16	10	TCB	Copy the value of C into B	0				
REG16	11	BZE	Branch if A is 0	1	x			
REG16	12	ADD	A := A+C	0				
REG16	13	SUB	A := A - C	0				
REG16	14	AND	A := A & C	0				
REG16	15	OR	A := A | C	0				
REG16	16	XOR	A := A ^ C	0				
REG16	17	SHL	A := A << 1	0				
REG16	18	SHR	A := A >> 1	0				
REG16	19	ACCSET	Put A's low 8 bits in global acc	0				
REG16	20	ACCGET	Load global acc into A	0				

Appendix B

Example code

B.1 Division algorithm in Salamander assembly language

```
Emulate dmm32 DIV
'Calculates num/dem
StartNameSpace
    alias num
    alias den
    alias quotient
    alias i
    alias den<<i
    alias bitI
    alias zero
    alias one

    'Get the operands
    uni opcopy 0 4 numAddress
    uni opcopy 4 4 denAddress

    'Get where to store the result and where to return to
    uni opcopy 8 4 quotientAddress
    uni opcopy 12 4 remainderAddress
    uni epccopy 2 returnAddress

    dmm32 imm 0 zero
    dmm32 imm 1 one
    dmm32 imm 0 quotient
```

```
    'Dereference the operands
    dmm32 imm numAddress: num
    dmm32 load num num
    dmm32 imm denAddress: den
    dmm32 load den den

    'Start i off at 31
    dmm32 imm 32 i

    Loop:
        'Decrement i
        dmm32 sub i one i
        'See whether bit i should be set
        dmm32 shl den i den<<i
        dmm32 jmpgr den<<i num Continue

        'If den<<i has become 0, then all the set bits
        'have been shifted off and this should be skipped
        dmm32 jmpeq den<<i zero Continue

        'Yes, it should, so set it and subtract the
        'shifted denominator from numerator
        dmm32 shl one i bitI
        dmm32 or quotient bitI quotient
        dmm32 sub num den<<i num

        Continue:
        'Loop back round if i is over 0
    dmm32 jmpgr i zero Loop

    dmm32 copy quotient quotientAddress:
    dmm32 copy num remainderAddress:
uni jimpl UNI:JIMPL returnAddress:
EndNamespace
EndEmulate
```

Appendix C

Guide: How to produce a basic Salamander interpreter

So, you have a device, and would like to run the vast array of existing programs written for the Salamander Virtual Machine (SVM)? That's not a problem – writing an SVM interpreter is easy! Just follow these instructions to get a basic one on your platform. Then, if you find that you need more performance, you can spend more time optimising your implementation.

C.1 Choosing a family

Every Salamander interpreter must provide instructions from two families. One of these is UNI, which everyone must provide. You can choose the other one. A good choice would be:

- Similar in operation to the device. So if you have 32-bit signed arithmetic, choose a family that does too.
- As close as possible to the family of programs you expect to run. Of course, any Salamander programs will run on your interpreter, but if the family is similar, they will be faster.

You also need to choose which of the instructions of the family you'd like to implement. Some families allow very few instructions to be implemented, while some need all of them. The more instructions, the better the performance, but the more effort you have to put in.

C.2 Choosing a way to read the bytecode

Salamander bytecode is stored in *.svm* files. You may find you need to convert to another file format to make it easiest to transfer bytecode to your device. It's completely up to you! Remember that the first 4 bytes of an *.svm* file are its size (little-endian, so the lowest byte comes first).

Make sure that you can easily and efficiently read and write the bytes of the bytecode as you write the interpreter for your device.

C.3 Set up the variables you need

Now, the first thing you need in the interpreter is a set of registers to that you'll need to read and write while the bytecode is being run.

- A program counter – which needs to be able to take values up to the size of the bytecode.
- A emulation program counter – of the same type as the program counter. It's used to store the old program counter when an emulation routine starts.
- A emulation operands array – which needs to be an array of bytes. It's used to store the operands of an instruction being emulated.
- An accumulator – which is just one byte, and is used for I/O.
- A bindings map – which maps from two bytes (the family number and instruction number of an instruction) to a program counter value (to jump to when that instruction needs emulating). To save space, you could implement this as an array of arrays, and only allocate the second-level array for a particular family when it is first used.
- The memory used by the family that you have chosen to implement. This may need registers, an addressable memory space, a stack, or anything else.

C.4 Begin the main loop

Loop forever! Each iteration will be one instruction executed.

Now, the first thing to do inside the loop is read the details of the instruction from the bytecode. Use the program counter to address the bytecode,

and increment it after each byte is read. The bytes that make up an instruction are:

1. The family number
2. The instruction number
3. The number of operands
4. The operands (remember, there could be any number of these, so loop through getting them)

How you store the operands is your choice. You may prefer not to copy them, and just keep a note of their location in the bytecode.

C.5 See whether the instruction is implemented

The next task is to check whether the instruction is one of the ones you are implementing. If it isn't, you need to use the emulation routine to emulate the instruction. First, copy the program counter into the emulation program counter, and the operands into the emulation operands. Then, read the emulation routine's address from the bindings map, and copy it into the program counter. Then start the main loop again to begin executing the emulation routine.

If the binding map doesn't contain an entry for the instruction, either the bytecode has a bug, or your interpreter has run it wrongly. Either way, you probably want to produce some kind of error message.

C.6 The big case switch

Here is where you will give the implementations for all the instructions. Depending on which instruction you are executing, you'll need to act differently.

C.6.1 UNI OUT and UNI IN

These two instructions are the only way to perform I/O on a Salamander machine, making it very easy for you. Depending on your device, you'll need to choose the most appropriate way to interact with the user. To implement OUT, you need to output the contents of the accumulator byte; and for IN, you need to take a byte from the user and store it in the accumulator. Most Salamander programs interpret the byte as an ASCII character, but you could write a program that uses it entirely differently.

C.6.2 UNI BIND

Here, you need to write to the binding map. The operands are:

1. The family number to be bound – one byte
2. The instruction number to be bound – one byte
3. The location in the bytecode that needs to be written to the binding map – the rest of the bytes, little endian

C.6.3 UNI JIMPL and UNI JNIMPL

These instructions, *jump if implemented* and *jump if not implemented*, are conditional jumps, depending on whether an instruction is implemented. An instruction is implemented if you provide it, or if it has been bound. The operands, in both cases, are:

1. The family number – one byte
2. The instruction number – one byte
3. The location in the bytecode to jump to, if the instruction described is implemented (in the case of JIMPL), or not implemented (in the case of JNIMPL) – the rest of the bytes, little endian

C.6.4 UNI OPCOPY

This instruction copies some of the bytes of the emulation operands into the bytecode (yes, you heard me right, it lets the program modify itself). The operands are:

1. The index within the emulation operand bytes to start copying – one byte
2. The number of emulation operand bytes to copy – one byte
3. The location within the bytecode to copy them to – the rest of the bytes, little endian

Go through and copy the bytes across, overwriting whatever was at that place in the bytecode previously.

C.6.5 UNI EPCCOPY

Similarly, this copies the emulation program counter into the bytecode. The operands are:

1. The number of bytes to copy – one byte
2. The location within the bytecode to copy them to – the rest of the bytes, little endian

C.6.6 The instructions of the family you chose

Read the specification of the family you have chosen for guidance how to implement it. It shouldn't be too hard.

C.7 That's it!

All you need to do now is go back to the beginning of the main loop and run the next instruction.

C.8 Testing your interpreter

You should test your interpreter using the unit test provided for its instruction family first. Then move on to the unit tests for the other families to ensure that your interpreter performs emulation routines correctly.

Once this is done, you should be able to run any program written for Salamander!

Appendix D

Visualisation of Salamander bytecode

Figure D.1 shows an image representing the bytecode for the SVM implementation of Conway's Game of life (see 4.3.3). Each pixel represents one byte, with 00 represented by black and FF represented by white.

Distinct sections of the bytecode's structure are easy to see:

1. The binding sequence is the section near the beginning with regular diagonal lines. This appears this way because all the instructions used in the binding sequence are of the same length. It starts dark and gets lighter because the program locations that branch instructions specify are getting larger.

2. The emulation routines reside in the large, unstructured area in the middle. The predominant colour is dark because the instructions and family numbers are all low. The section with grey diagonal lines is the implementation of the reverse (REV) instruction in DMM32, which uses a large, statically unrolled loop.

3. The program itself starts at the dark section near the end. Because it is translated from JVM bytecode, it uses the stack-based instruction family, which doesn't need to specify memory locations in the way that the other families do. This means the only bytes that have high values are the program locations for branches. This program happens to have very few branch instructions in the first half, making it dark.

Figure D.2 shows a hex dump of SVM bytecode, containing a portion of the binding sequence and emulation routines.

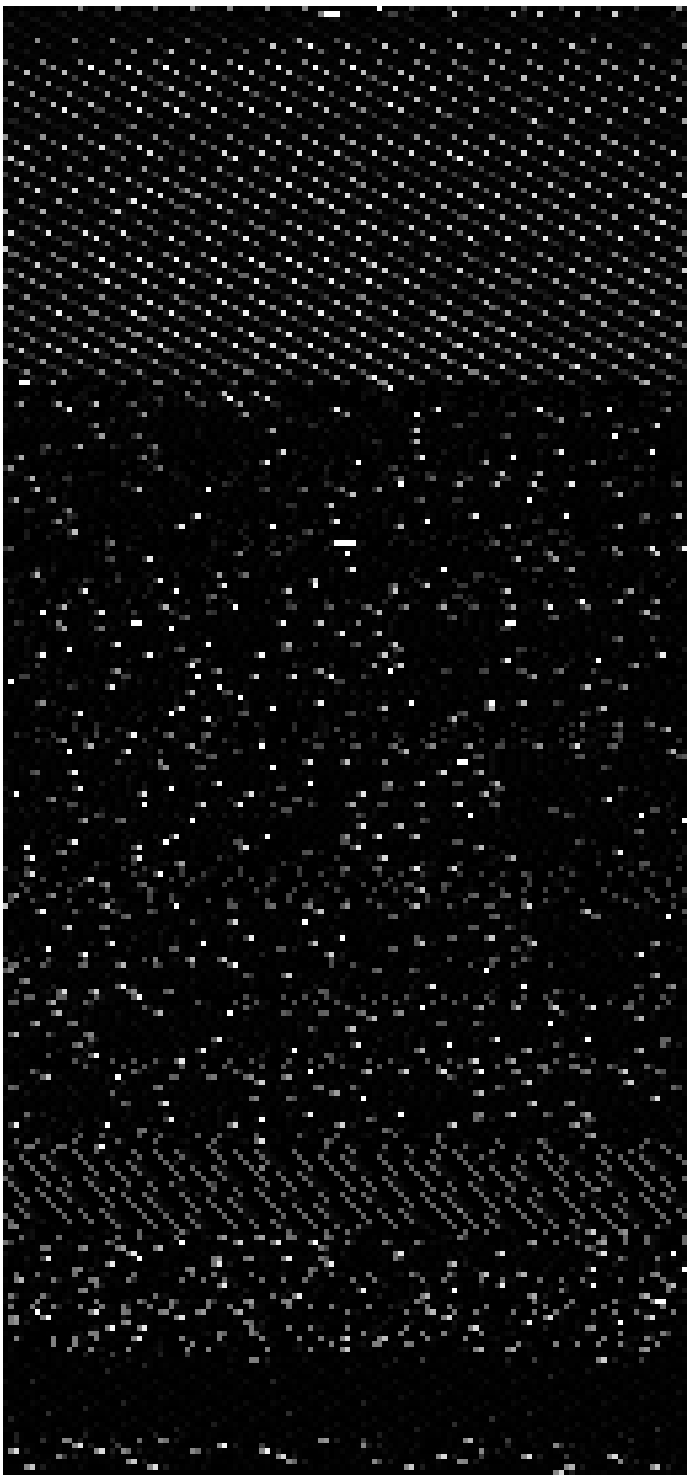


Figure D.1: A Salamander bytecode interpreted as an image

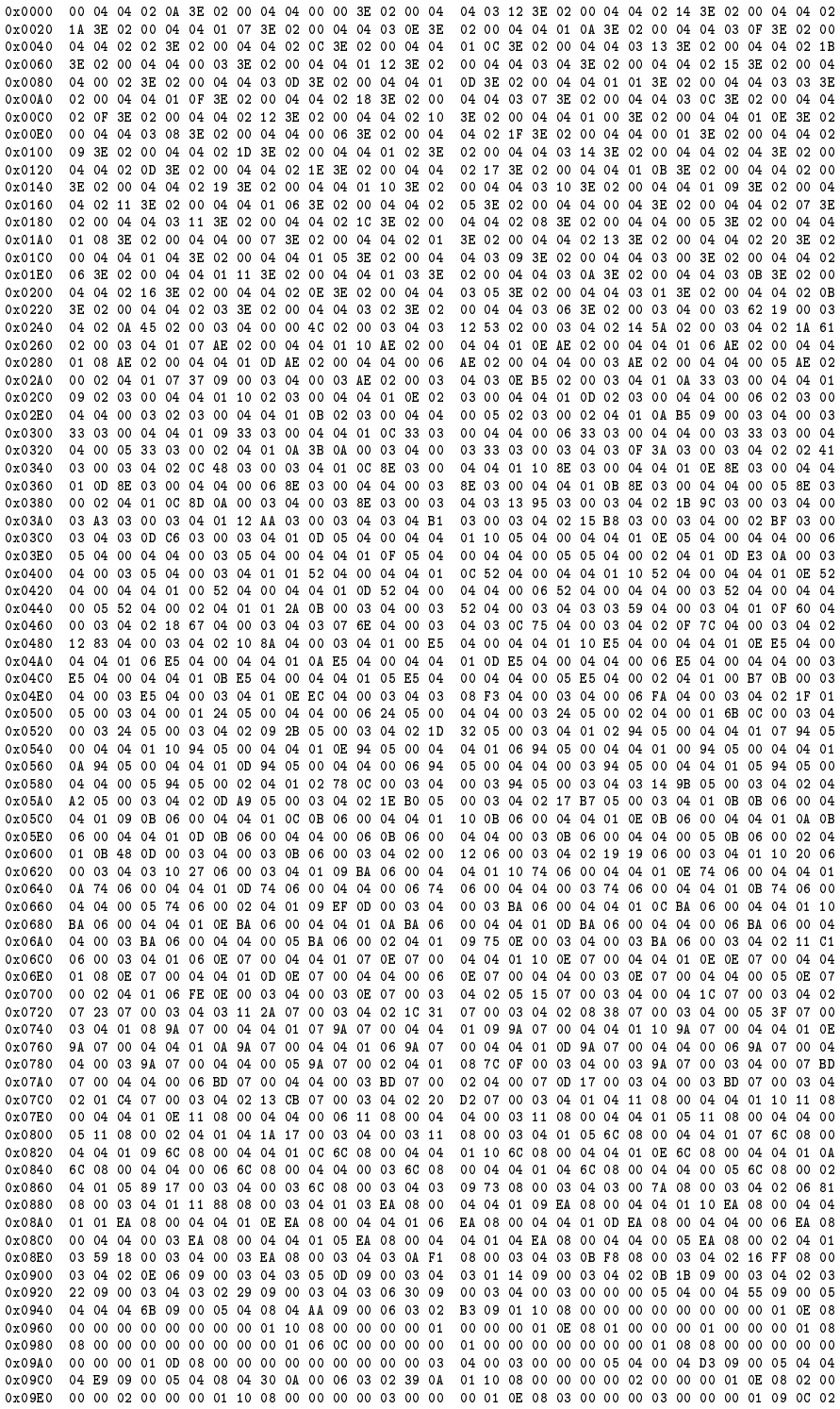


Figure D.2: The first 0X1000 bytes of the SVM library in bytecode, hex

Appendix E

Original Proposal

Alex Davies

St John's

AJD74

Computer Science Tripos Part II Project Proposal

A system to allow rapid deployment of programs to new architectures

17/10/07

Project Originator: Alex Davies & Christian Steinruecken
Resources Required: Ordinary computers and an ECAD labs board. See Project Resource form.
Project Supervisor: Christian Steinruecken
Director of Studies: Robert Mullins
Overseers: Andrew Pitts and Peter Sewell

E.1 Introduction

I would like to create a system which enables new and existing programs to be executed on a new architecture with minimal investment. This system would be aimed to enable prototyping, but also to allow greater efficiency to be added incrementally as the architecture becomes more mature.

New architectures and platforms are being designed at an increasing rate, as technology allows computing to be used for diverse applications. Many companies produce specialised devices for their own purposes. These could

often be used by another company for a completely different purpose, but because of the cost of experimentation with the platform, then translation of existing code to the new architecture, opportunities to do so are missed.

Virtual machines are already in widespread use with the intention to make programs portable. However, previous VMs have always been designed by trading off efficiency against implementation cost. This project will create a virtual machine with a flexible specification, so that only a very small amount need be provided for a minimal implementation, but fully featured implementations can provide more in order to be more efficient.

This will mean that a company whose existing programs are written for this VM can experiment with a new platform at minimal cost. Then, if they choose to use the new platform, they can improve the efficiency of their VM implementation to make it ready for production. They never need to translate, or make cut-down versions of their programs.

I will use the terminology "Virtual Machine" or "VM" to mean the system for which programs will be written. There will be various implementations of the VM. I will call each of these an "interpreter".

E.2 Key implementation concepts

The virtual machine will specify that interpreters must provide a Turing-complete subset of the instruction set. When an instruction is encountered which is not implemented, it will be looked up in a library of instructions in terms of other instructions.

The virtual machine will provide instructions which include many provided by the Java Virtual Machine (JVM). Although, due to the restrictive standard of the JVM, it will not be possible to run Java bytecodes directly in my VM, I will write a tool to do a conversion so that programs written in Java and compiled using the Java compiler can be run. Only a subset of Java will be supported.

E.3 Criterion for success

The project can be considered a success when a program written in Java can be run on a microcontroller device. This covers both the requirement that the VM is cheap to implement on new devices and the requirement that it is sufficiently powerful for practical use.

E.4 Resources

I will use ordinary computers for the majority of the project. I have secured use of an ECAD board from the computer lab to use as an example of a microcontroller device.

E.5 Starting point

I have no experience of specifying or implementing virtual machines, but extensive experience of using them (specifically Java and .net). I am confident with a number of programming languages, but because of the cross-architecture nature of this project, I may decide to also use some that I don't already know.

E.6 Structure of the project

The project will require me to:

- Design and specify a flexible virtual machine, to which various paradigms and instructions can later be added.
- Design, implement and unit test an interpreter, firstly on a normal computer.
- Design and implement a library to translate any instructions that are not provided by a particular interpreter
- Write and run example programs that use instructions provided by the library (this will form the system test for the library and the interpreter)
- Research the JVM spec, ensure that my VM has equivalent instructions and write a translator (instruction-for-instruction) from JVM bytecode to my own bytecode
- Write some example programs in Java, compile them using the Java compiler and run them on the primary interpreter (this will form testing for all three things).
- Design, implement and test a second interpreter on a microcontroller device

E.7 Timetable and milestones

These milestones correspond to the work areas I have identified above, with the exception of the progress report and dissertation.

A specification	02-Nov
An interpreter on a desktop computer	23-Nov
The first iteration of the library of instructions in terms of each other	07-Dec
The interpreter and library are tested and confident have no bugs. Some useful examples.	01-Feb
Progress report	29-Feb
The VM can run programs written in java	29-Feb
Useful example programs and confidence that the translation is tested thoroughly	14-Mar
A working interpreter on a microcontroller device, running all existing programs	04-Apr
The dissertation	09-May