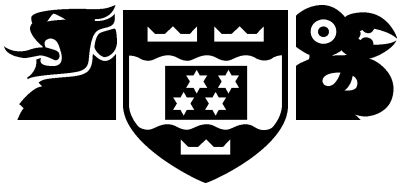
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

SGJ: Sega Genesis Java

Vipul Delwadia

Supervisors: Stuart Marshall, Ian Welch

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Video games are an important part of New Zealand's cultural heritage. A common approach to preservation of legacy software is to migrate it to a modern architecture. This project explores the issues with automated conversion between architectures. SGJ is a prototype conversion tool and framework which implements solutions to some of the conversion issues.

Acknowledgments

I'd like to thank my supervisors Stuart Marshall and Ian Welch for their wonderful support throughout the year, even into the wee hours of the night. I'd also like to thank Stephen Nelson for helping me in creating a parser, and Chris Male for helping me with control flow graphs. I'd also like to thank them both for proof reading my report. Finally I'd like to thank the Memphisonians for being there, all two of the honours students and the rest of the undergrads and grads. Without them I would have no-one to beat in Quake.

Contents

Figures

Chapter 1

Introduction

Video games are an important part of New Zealand's cultural heritage. A common approach to preservation of legacy software is to migrate it to a modern architecture. This project explores the issues with automated conversion between architectures. SGJ is a prototype conversion tool and framework which implements solutions to some of the conversion issues.

This chapter looks at the contributions of the project, as well as the motivations for the project.

1.1 Contributions

This project will contribute two artefacts. One is a methodology for translating from lower-level code such as assembly, to higher-level code. The other is a prototype tool, SGJ, to parse the Motorola 68000 assembly source code and translate it into Java source code.

1.2 Motivation

The preservation of games is good for cultural heritage. Existing approaches include encapsulation (preserving the software and data for future inspection), migration of the code to a new platform (which is usually done manually) and emulation of the original hardware and software environment [13].

Migration of the code has the advantage that it runs natively and can make direct use of local hardware and software resources. It may also be the case that migrated code requires fewer resources than writing a general purpose emulator and this is very important in the context of mobile devices that always have fewer resources than PCs or standalone games machines. Making preserved games playable makes the experience accessible to the public, and mobile phones are a ubiquitous platform capable of playing these types of games.

Manual migration between architectures is a common but long and expensive process. This project will explore some of the approaches to automatic translation, and the problems involved, and finally a prototype solution to some of those problems.

1.3 Structure of the Report

The following chapters of this report are as follows.

Chapter two provides the background to the project, describing previous work in the field, and introduces the hardware the project tackled.

Chapter three discusses the problems encountered performing such a conversion from a low level program to Java, and describes the potential solutions to those problems.

Chapter four illustrates the prototype tool SGJ, including the methodology, design and implementation.

Chapter five gives the results of the translation, and the issues encountered.

Chapter six finishes off the report, summarising the contributions and findings of the project, and suggests future work.

Chapter 2

Background

This chapter discusses some of the reasons for digital archiving, as well as the two major approaches to preservation. The source hardware is introduced, and the assembly code is described in detail.

2.1 Digital Archiving

The rate at which technology advances is ever increasing. Hardware which was state-of-the-art a few years ago is used as a doorstop today. Because of this, the need for digital archiving is also growing. Programs that were designed for previous generations of hardware are now obsolete.

The preservation of these programs is important to both society and businesses. Legacy programs in large businesses is commonplace, and there are a couple of approaches to run these programs on modern hardware. One approach is to emulate the original hardware and software on modern hardware. Virtualisation fits under this category, basically by executing the software on the intended operating system, which in turn executes on virtualisation software. VMWare [18] is an example of such software, which emulates a set of hardware components. If the program is used as the operating system, then other emulators, such as Genesis Plus [2], can emulate the hardware allowing the software to run on it directly.

Another approach is to translate the software such that it executes on modern hardware directly. There are dynamic, as well as static translation approaches, and while a manual translation can be performed [6], automated translation is more common [4, 5, 8, 10, 11, 17].

Figure 2.1 briefly compares the emulation approach with the translation approach.

Category	**Notes**
Speed	Tradtionally native code is fast than emulated code, but there is overhead in translation
Size	Emulation requires the emulator and original program, whereas translation only requires the translated program
Look/Feel	Emulation will give a closer look and feel to the original program than translation
Audience	Emulation requires substantially more knowledge to use than a translated, native application
Legality	It is not clear whether emulation is legal, nor is it clear if translation is legal

Figure 2.1: Comparison between emulation and translation of legacy programs

2.2 Related Work

There has been a large amount of research into automated program translation already. Out of that research, there are three pieces of work which are highly relevant to this project.

The first is Cibyl, which is an environment for language diversity on mobile devices [8]. The program itself, Cibyl, takes compiled MIPS binaries and translates them into Java bytecode. The translation is statically performed, and then the Java bytecode is executed with Cibyl's runtime library. Cibyl fully supports C for the MIPS binaries.

Ephedra is a C to Java migration environment [11]. It translates C source code into Java source code, while performing pointer mappings, type conversions, class conversions and various other C-only to Java conversions. Further, Ephedra has the ability to perform optimisation of the Java source produced.

Finally there are two related works by Cristina Cifuentes. *asm2c* [5] translates SPARC assembly into C source code. It employs various analyses, including data flow, control flow and type analyses to recover control flow structure, high-level language expressions, and high-level type information respectively. *dcc* [4] is a decompiler for the Intel i80286, producing C source code. It performs similar analyses as *asm2c*, as well and dead-register and dead-code elimination, and data type propagation across procedure calls.

2.3 Hardware

A *video game console* (*console* for short) is a computer which connects to a display (typically a television) to play video games. For the purposes of the project a console needs to be selected to translate from. The console that we have selected is the Sega Genesis (Mega Drive). It was a popular gaming machine in the 1990s, and has a lot of similarities to other consoles produced by Sega throughout the years. This is important in thinking about the future and the ability to apply the techniques of this project to other hardware platforms. Further, there is a freely available compiler which produces directly executable code for the Sega Genesis. The important specifications[1] of the Sega Genesis are described in Figure 2.2.

The Sega Genesis operates in the same manner as most other consoles of the same era. The programs are distributed in cartridge format, and are inserted into the console before powering the machine on. Once the machine is powered on, the system boots, loads the program on the cartridge into memory, and executes it until the machine is powered off. The programs on the cartridge are in binary format, and are in machine code for the Sega Genesis specifically.

2.4 Assembly

The assembly code for the machine is typical of low level assembly. There are a number of basic constructs, such as flow control and data manipulation, as well as a number of immediate registers and of course memory manipulation.

There are eight general purpose 32-bit registers, `d0` to `d7`, used to store any form of data. There are also eight 32-bit address registers, `a0` to `a7`, used to store memory addresses. Register `a7` can be used as the user stack pointer, `SP`, in order to treat a section of memory as a stack. Finally, there is a status register, `SR`, which contains the condition codes that reflect the results of a previous operation. Condition codes are described later in this section.

Instructions in the assembly can have up to two operands, in the addressing modes as specified above. Further, there are a number of different instruction types, discussed

[1] The components not listed are not considered as part of this project.

Component	Specification	Notes
CPU	Motorola 68000 16-bit microprocessor	This is the primary CPU, and is clocked at either 7.16 MHz (PAL) or 7.67 MHz (NTSC)
Main RAM	64 KB	This is the addressable section of memory
Video RAM	64 KB	This memory is only accessible via the Video Display Processor (VDP)
Cartridge ROM	Up to 4MB	Addressable as main memory. Larger cartridges can use bank switching
VDP	Texas Instruments TMS9918	Four resolutions: 256x224 and 320x224 (NTSC), 256x240 and 320x240 (PAL)
Main sound	Yamaha YM2612	Clocked at 7.16 MHz, six FM channels
Secondary sound	Texas Instruments SN76489	Four-channel PSG

Figure 2.2: Specifications of the Sega Genesis

in more detail in Section 2.5. Briefly, some of the relevant instruction types include: *Data Movement*, *Integer Arithmetic*, *Logical Operations*, *Shift and Rotate Operations*, *Bit (Field) Manipulation*, *Binary-Coded Decimal Arithmetic*, *Program/System Control*, *Cache Maintenance*, and *Memory Management*.

An instruction's operand has many addressing modes[2]. The addressing mode specifies the manner in which the machine interprets the operand given. The simplest addressing mode is the *Register Direct Mode*, where the operand is specified by either a data register or an address register. The next mode is *Address Register Indirect Mode*, where the effective address field specifies the address of the operand in memory. *Address Register Indirect with Postincrement/Predecrement Mode* is similar to the previous mode, however the effective address is an address register, and it is incremented/decremented by the size of the operand. Along a similar tract, *Address Register Indirect with (8-bit) Displacement Mode* is where the effective address is the sum of the address register and the (8-bit) 16-bit displacement integer specified, along with the index in the 8-bit version, and the operand is the address in memory. In *Absolute Addressing Mode*, the operand is in memory, and the address of it is the effective address, which is sign extended if it is a short. Finally, in *Immediate Data Mode*, the operand is the effective address given. Examples of each addressing mode are given in Figure 2.3. The first operand demonstrates the addressing mode, and the second is always `d0`, in register direct mode.

The hardware has support for conditions, which are use to test various aspects of the latest result of an operation, in the form of conditions. Conditions can be set or cleared (since they are a single bit long) depending on the result of an operation, and there are five different conditions in total:

Carry (C) Set when a carry out of the MSB of an operand occurs for addition, or if a borrow occurs in subtraction, and is cleared otherwise.

Overflow (V) Set if an arithmetic overflow occurs, otherwise cleared.

[2]There are a few more very similar displacement based modes, which are not discussed here, but are in the programmer's reference manual [12]

Addressing Mode	Operand	Instruction
Register direct mode	`d1`	`move.l d1, d0`
Address register indirect mode	`(a0)`	`move.l (a0), d0`
Address register indirect mode (postincrement)	`(a0)+`	`move.l (a0)+, d0`
Address register indirect mode (predecrement)	`-(a0)`	`move.l -(a0), d0`
Address register indirect mode (displacement)	`16(a0)`	`move.l 16(a0), d0`
Absolute addressing mode	`$42`	`move.l $42, d0`
Immediate data mode	`#42`	`move.l #42, d0`

Figure 2.3: Examples of operand addressing modes using the `move` instruction

Zero (Z) Set if the result is zero, otherwise cleared.

Negative (N) Set if the MSB of the result is set, otherwise cleared.

Extend (X) Set to the value of the carry bit for arithmetic operations, otherwise not affected.

A complete assembly file can be found in Appendix B.

2.5 Language Differences

The Java language is a strongly typed language that does not allow pointer arithmetic or arbitrary changes to program flow. On the other hand, assembly is untyped, allows arbitrary pointer arithmetic and arbitrary changes to program flow. In addition, Java does not allow data to be treated as code whereas there is nothing preventing this in assembly. In converting from assembly to Java source code[3], the most important differences are best described in terms of the instruction types introduced in Section 2.4:

Data Movement These are the instructions which move data between locations, including main memory and registers. While Java can assign values to variables ("registers"), it doesn't have a way to manipulate "main" memory.

Integer Arithmetic These instructions are the basic operations, such as `add`, `sub`, `mul` and `div`. Java supports these four operations, however other instructions such as `cmp`, the compare operation, do not have a counterpart in Java.

Logic These are the four basic logic operations, `and`, `or`, `eor` (exclusive or), and `not`. Java has these operations natively.

Shift and Rotate These instructions move, rotate or swap bits of the operand. Java has some, but not all, equivalent operations.

Bit Manipulation These instructions operate on a single bit of the operator. Java can do this through a combination of operators.

Program Control These are the instructions which control the flow of the program. There are a number of different types, including: *branch if condition*, *decrement and branch if condition*, *set if condition*, *branch*, *jump*, *subroutine call*, *return from subroutine*, and *test condition*. The instructions which relocate the program to a new address can be approximated by the `goto` construct. However, Java doesn't have this construct.

[3]Throughout this document Java is referring to the Java source code language, and not the Java bytecode language

System Control Many of these instructions fall under three broad headings: *privileged data/-control*, *trap generating*, and *condition code register*. The privileged data instructions have no counterpart in Java – all Java code operates at the same privilege level. The privileged program control or the trap generating instructions, such as `stop` or `trap`, also do not have Java equivalents, but can be recreated.

In addition to those instructions, some other fundamental language differences exist:

Operand width Instructions on the 68000 can have up to three different operand widths. For example, the `move` instruction can operate on `long`, `word` or `byte` sized operands. In Java, while there is support for similarly sized primitives, `int`, `short` and `byte`, respectively, there are complications in the handling of overflow and signing, resulting in a less than perfect parallel.

Memory manipulation On the 68000 main memory is directly accessible via multiple instructions, and the effective address is specified as an operand, much like a "pointer". In Java, direct memory access is not allowed, and the only "pointer"-type construct is a reference to an `object`.

Chapter 3

Converting to Java

The conversion from a low level language such as assembly, to a high level language, such as Java, is non-trivial. As discussed briefly in Chapter 2, the differences between the languages is quite vast. The following sections will describe in detail what major problems and issues arise when attempting to perform this kind of translation. There are three broad categories which the issues fall under: Memory, Control Flow and Input/Output.

First, the main requirements for a prototype tool to be created during the project are:

- The prototype must produce code which is executable on the Java Virtual Machine, version 1.5.
- The code produced is not required to be well designed or human readable
- The Java code produced must be executable and have a similar look and feel as the original program

3.1 Memory

These are the aspects of the conversion that are memory related in one form or another. There are four specific issues that need to be explored: self-modifying code, variably sized operands, memory manipulation and untyped data.

3.1.1 Self-Modifying Code

One major problem with doing any kind of static translation from assembly is the issue of self-modifying code. Self-modifying code is broadly any program that loads, generates, or mutates code at runtime [3]. There are some reasonable uses of self-modifying code in console programs and games, such as runtime code generation, compression to fit in the required space, or to hide code as prevention against disassembly or debugging.

```
mycode: move.l #42, d0
        ...
        move.l mycode, a0
        add.l #1, a0
        add.l #1, (a0)
        jmp mycode
```

Figure 3.1: An example of self-modifying code in 68000 assembly

It is unclear whether programs for the Sega Genesis (hereafter "source hardware") actually use self-modifying code, however it is technically very feasible to do so. For example, consider the code in Figure 3.1. The original command was placing the value 42 in the register `d0`. However, the next set of commands place the address of `mycode` in the register `a0`, then increment that register by 1, and then increment the value of the bit of memory pointed to by `a0`, which is now the address of the first operand of the move instruction, by 1. Effectively this has modified the command at `mycode` from `move.l #42, d0` to `move.l #43, d0`.

If there is self-modifying code in the source assembly, then doing a static analysis of the source code difficult. A simple evaluation of the control flow will not discover the modified code. This means that quite complex algorithms are required to detect it, let alone handle it and determine the code paths.

3.1.2 Variably Sized Operands

As discussed in Section 2.5, the assembly and hardware has support for multiple sizes of operands. For instance, the `move` instruction can operate on `long` (32-bit), `word` (16-bit) or `byte` (8-bit) sized operands, denoted by `move.l`, `move.w` and `move.b` respectively.

Java has primitive data types with the same numbers of bits in the form of `int` (32-bit), `short` (16-bit) and `byte` (8-bit). However, there are many complications stemming from the behaviour of the hardware making a one-to-one mapping is difficult, and requires many more additional checks and conversions in the Java code.

Registers in the source hardware are 32-bit native, meaning that size of each register is fixed at 32 bits long. In an instruction involving a register, if the size of the operand is `word` or `byte`, only that many bits of the register are read/written.

In the hardware, when the result of an operation is too large to fit in the given size, the value wraps around. For example, the instruction `add.b #1, d0` adds 1 to `d0` and stores it back into `d0` as a `byte`. In binary, if `d0` contains the value 127, this is what is happening:

```
   01111111  127
+         1  1
----------
=  10000000  -128
```

So the result is -128, and this is placed back into `d0`. In addition to the correct value being place in `d0`, the hardware also sets the overflow condition `V`. In Java, although the correct behaviour is observed when using the same size data type, there is no indication that the overlap has occurred, and so this must be checked manually.

With variable width operands, the MSB signifying whether the value is negative or not has multiple locations. Depending on the representation chosen for the smaller sized data types, care must be taken to ensure that the value is indeed positive/negative.

Further, there are some instructions which require that the operands, if specified as `word` or `byte`, need to be sign extended to 32 bits long. Again, this requires additional code in Java to handle this correctly.

3.1.3 Memory Manipulation

Apart from the immediate registers, any data that is stored during the lifetime of an assembly program is in main memory. In order to access and modify that memory, the assembly provides a number of instructions which are dedicated to such a task. In addition, most of the standard instruction set is capable of manipulating main memory, by giving a memory address as an operand.

Instruction	Description
`bra label`	Unconditionally branches to `label`
`bxx label`	Branches to `label` if the condition `xx` is met
`jsr label`	Calls the subroutine at `label`
`rts`	Returns from a subroutine call
`jmp label`	Unconditionally jumps to `label`

Figure 3.2: Overview of the major flow control instructions.
Note: `xx` is a condition, such as `eq`

The latter behaviour is typical in a language like C, in the form of a pointer. Pointers allow memory to be treated as a large array, and as mentioned in Section 2.3, memory can be used as a stack as well.

While this is easily represented in C, since it has native pointer support as well as direct memory access, this is not the case in Java. Java does have references, but because they are strongly-typed and the address cannot be manipulated using arithmetic, they are not sufficient. Further, there is no direct memory access because Java has a cross-platform architecture.

3.1.4 Untyped Data

Assembly as a language essentially has no type information. Java is a strongly-typed language, so any conversion will result in untyped Java unless the type information is inferred/extrapolated.

3.2 Control Flow

There are three control flow issues in the conversion, goto elimination, interrupts, high level constructs. The first two are essential to a complete conversion of any program. The third is an interesting issue aiming to improve the quality and efficiency of the translated code.

3.2.1 Goto Elimination

The set of instructions which can alter the flow of control on the 68000 is relatively small, with four common, user level instructions, as described in Figure 3.2.

In order to faithfully replicate this construct in a higher level language, there is an obvious and sufficient mechanism in the form of the `goto` keyword. For example, `goto` in the C programming language transfers execution of the program from the point `goto label` is used to the point in code where `label` is located. This can be used directly to approximate `bra` and `jmp`, and when coupled with a conditional construct (such as `if`) can be used to approximate `bxx`.

For example, consider Figure 3.3. The left hand code is some example assembly with a `jmp` instruction, and the right hand code is some similar C code with the same jump imitated by a `goto`.

However, `goto` is a reserved keyword and is not implemented in the latest release of Java. Java byte code does have `goto`, but it is limited to an address which is within the same method as the `goto` itself [9]. The problem of reliably and correctly providing the equivalent of `goto` is a difficult one.

3.2.2 Interrupts

An interrupt is an asynchronous signal from the hardware or a synchronous event in software requesting a change in execution. Interrupts in hardware are used to perform a number of crucial functions, such as exception handling and IO. Hardware interrupts result in the processor saving execution state (such as the value of registers), and switch control flow to an interrupt handler.

Software interrupts operate in a similar fashion to hardware interrupts, but are invoked via instructions in the instruction set. The 68000 has a number of instructions which interrupt, such as the `TRAP #vector` instruction, which causes a trap exception with the given `vector`.

The issues that arise with interrupts are that they can occur at any point in the program. While they can ignored by masking them, non-maskable interrupts cannot, and are handled when generated regardless of the value of the interrupt mask.

From the conversion point of view, interrupts are difficult to deal with. It is not know at which point an interrupt will occur, and so every point in the program must be treated as interruptible. Further, any converted code will have to be able to generate the interrupts independent of the main execution of the program.

There are three types of interrupts that typically occurs in the life of a program on the Sega Genesis: the generic interrupt, a VDP horizontal retrace and a VDP vertical retrace. The two VDP retrace interrupts are used just to notify the user program of graphics updates. The generic interrupt is for all other types of interrupts, such as exceptions.

3.2.3 High Level Constructs

Assembly code is very unstructured, and at times hard to follow. Control flow is achieved purely through the use of what is essentially C-style `goto` statements. While there are no explicit high level constructs such as `if` and `while` statements, they are most definitely in the source, created by branching to labels.

As an example, consider the following set of commands:

```
cmp.w d1,d0
sne d0
tst.b d0
beq IF_JUMP_0
```

Essentially what is happening is that the two registers `d0` and `d1` are being compared, then `d0` is being set to 0s if the result was not zero (`sne`), then `d0` is being tested for 0s (`tst`), and finally the code branches if the result is zero.

While this set of instructions seems to be complex, it is actually a very simple conditional statement, which can be represented by the following C code:

```
    add.l #1, d0
    jmp destination
    ...
destination:
    sub.l #1, d0
```

(a) Assembly code

```
    x = x + 1;
    goto destination;
    ...
destination:
    x = x - 1;
```

(b) C code

Figure 3.3: `goto` in assembly and C

```
if (d0 == d1)
    goto IF_JUMP_0;
```

There are many sets of instructions in assembly which can be replaced with high-level conditional and looping constructs. Further, there are other sets of instructions which can be replaced with other high-level constructs, such as method calls (with parameters), and also code which gets and sets memory representing variables.

3.3 Input/Output

These final three issues are also important to a successful conversion. Of the three, graphics and input are essential to a correct conversion, and sound is key to the goal of a similar look and feel between the original programs and their translations.

3.3.1 Graphics

The graphics system in the Sega Genesis consists of a video display processor (VDP). Access to the VDP memory is permitted, so graphics operations can be performed by writing to the memory directly. Additionally, operations can be performed by direct memory access (DMA).

Graphics operations can also be performed using sprite operations. Data is written to a section of memory designated for sprite operations, and there are mechanisms in place for collision detection of sprites.

Programs and games for the Sega Genesis have two options to produce graphics on the screen, either by generating them programmatically, or by loading pre-existing graphics from the cartridge into memory.

Graphics are an issue in Java because direct access to the hardware is not feasible. Instead, because of Java's cross-platform compatibility, any graphics operations must be done through APIs. Because of this, much more work has to be done to bridge the source hardware's graphics with Java's graphics, rather than a straight-forward alignment with hardware based graphics.

3.3.2 Sound

There are two hardware components available to make sound on the Sega Genesis, the PSG control and the FM sound control.

Again there is memory available to write to in order to produce sound, as well as registers in the FM control unit. The FM unit supports 6 channels of FM sound, and producing sound using the FM unit is a matter of writing memory in the correct fashion.

The PSG on the other hand only has four channels, three of which are tone generators and the fourth a noise generator. The PSG is controlled through a single physical port on the CPU (port `$7F`), by writing a pair of bytes.

Sound is also an issue in Java, again because direct access to the hardware is not feasible due to the cross-platform compatiblity. This means that sound output in the translation requires more work in bridging the source hardware sound system with Java's sound API.

3.3.3 Input

User input on the Sega Genesis is primarily by a joypad. Again, this is achieved by reading main memory at a specific address. Since the controller only has a small number of

functions, the state of all the buttons can be read using a single read, since each button is represented by a single bit in the data read.

The joypad for the Sega Genesis has 4 directional keycodes, up, down, left and right, as well as three action buttons and a start button.

Chapter 4

Prototype

This chapter describes the prototype tool created, called Sega Genesis Java (SGJ), which parses Motorola 68000 assembly and produces Java source code. The solutions given to the problems from Chapter 3 are implemented in the prototype.

4.1 Methodology

The project was conducted in a rapid development fashion [19]. The planning game resulted in a number of tasks with various estimates of the time involved for each one. The project had fortnightly iterations, there and were approximately 9 iterations in total. In addition, weekly meetings were held to evaluate the status of the project and to set the velocity accordingly.

Rapid development was chosen because the project was highly changing in terms of tasks. Issues that arose during the course of the project could not be known prior to starting the project, and so rapid development allowed for easily modifiable tasks.

In order to translate programs for the Sega Genesis into Java, the programs in either source or binary format are required. Further, once that choice is made, the program has to be parsed, analysed and converted into some intermediate representation.

The Sega Genesis was released over 15 years ago, and so games for the platform in their original cartridge form are difficult to find, especially ones in good condition. Having found such games, there is still the matter of making a digital bit-for-bit copy, otherwise known as a *ROM* image. ROMs can also be found on the internet, however the legality of making, distributing, and/or downloading these ROMs is uncertain, due to the copyright issues.

Instead, one can write their own programs in assembler, and then by using a freely available assembler can compile the programs into ROMs suitable for use on an emulator. Further, there is a compiler called BasiEgaXorz (BEX) [14], which compiles programs written in its custom dialect of BASIC into ROMs which run on both a compiler and the hardware itself. This is the approach that SGJ takes.

4.2 Design

Parsing of the binary ROM is somewhat difficult, however it must be done in order to create an intermediate representation. Alternatively, parsing of the assembly is much easier, and in this case SGJ can take advantage of existing parser generators. The binary can even be decompiled into assembler, also using freely available tools. BEX also outputs the assembly corresponding to the binary, and this is what SGJ uses.

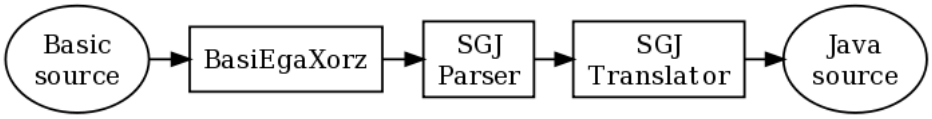


Figure 4.1: Overview of the translation process

The assembly that BEX produces is somewhat more readable than the output of a decompiler, simply because the binaries do not carry any superfluous information, such as named labels or comments.

SGJ parses the assembly output of BEX using a parser generated by ANTLR [15]. ANTLR generates a lexer and LL(*) parser[1]. The full ANTLR grammar can be found in Appendix A. After parsing is complete, a tree structure of the program is now in memory, where the root is a collection of labels, and within each label is the set of commands between it and the label immediately following it.

SGJ then traverses the tree using the Visitor Pattern [7], translating each label, command, definition and labelled data into Java source code as described below in Section 4.6.

Finally, the execution environment takes the translated and compiled Java, and executes within a graphical interface. The execution environment provides all the necessary instruction implementations, as discussed in Section 4.6, as well as the libraries for memory, graphics, input and more.

A simple diagram showing the translation process can be found in Figure 4.1.

4.3 Solutions to Memory Issues

This section discusses potential and actual solutions to the memory issues presented in Section 3.1.

4.3.1 Self-Modifying Code

This issue is difficult to solve, as discussed in Section 3.1.1, especially when performing a static analysis of the source code. Given this, the issue is outside of the scope of the project.

There have been approaches to model and understand self-modifying code, such as the *state-enhanced control flow graph* [1].

4.3.2 Variably sized operands

In Java, there are a few ways to represent registers, such as a 32-bit data type (`int`), or 4 8-bit data types (`bytes`), or even an array of 32 `boolean` values. In these cases, there needs to be some conversion using bitwise operations between the 32-bit register and the 8 or 16-bit operand. SGJ represents registers as 32-bit `ints`, performing the narrowing of bits dynamically if required.

Overflow is detected by checking the result after every operation. Since the data type handles the overflow in Java in the desired manner, only the overflow (V) condition needs to be set. Further, the MSB is checked, and the negative (N) condition is set accordingly.

Listing 4.1: `Pointer` Interface

```
public interface Pointer {
    public int getValue();
}
```

Listing 4.2: `RegisterPointer` Interface

```
public interface RegisterPointer implements Pointer {
    private Register reg;
    public final PointerType type;
    public RegisterPointer(Register r);
    public RegisterPointer(Register r, PointerType t);
    public Register getRegister();
    public int getValue();
    public static enum PointerType {
        POSTINC, PREDEC, NEUTRAL;
    }
}
```

Listing 4.3: `LiteralPointer` Interface

```
public interface LiteralPointer implements Pointer {
    private int address;
    public LiteralPointer(int i);
    public int getValue();
}
```

Listing 4.4: `Memory` Interface

```
public class Memory {
    public static final int MEMORY_LIMIT = (64 << 18) - 1;
    private static final byte[] ram = new byte[MEMORY_LIMIT + 1];

    public static void clear(Pointer p, Size s);

    public static int get(Size s, int address) ;

    public static void set(int i, Size s, int address);
}
```

Figure 4.2: SGJ interfaces for the pointer and memory interfaces

4.3.3 Memory Manipulation

The main problems found for this issue in Chapter 3 were the lack of pointers in Java, and that direct memory access is not feasible.

SGJ has an array of `bytes` to represent main memory, and getter and setter methods have a size argument to allow retrieval/placement of `long`, `word` or `byte` sized values. SGJ will return an error when attempting to read or write memory outside the addressable range.

Pointers are represented by two separate types, the `LiteralPointer` and the `RegisterPointer`. A `LiteralPointer` is used when the value of the pointer is either constant, such as a definition, or is a literal value, as the operand. These are actually immutable pointers, which are sometimes called references. A `RegisterPointer` is used when the pointer is the value of a register. Further, these pointers have the additional complication in that they may be supplied in pre-decrement or post-increment mode (see Section 2.3). The interfaces for the pointer and memory types in SGJ are given in Figure 4.2

It is not known exactly which parts of memory are allowed to be read or written, however it is the case that memory access outside the range (less than 0 or greater than the size of memory) is not allowed. In this case there is a runtime check and an exception is thrown if the memory address is outside the addressable range.

4.3.4 Untyped Data

The inference of typed data from the assembly is outside the scope of the project. SGJ treats all data as the same type, which is integer data. Registers are typed as 32-bit `ints`, and main memory is an array of 8-bit `bytes`. All method arguments are of type `int`, and narrowed down as required. Condition codes are represented as booleans, and constants are `ints` as well.

4.4 Solutions to Control Flow Issues

This section discusses some possible solutions to the control flow issues presented in Section 3.2.

4.4.1 Goto Elimination

Section 3.2.1 gave an introduction to the assembly code's branching instructions, and compared them to C's `goto`. However, as mentioned before, Java does not implement the `goto` keyword.

SGJ takes a simple yet effective approach, where `goto` is simulated using a `switch` statement. Any desired destination label is added to an `Enum` of labels, and a `switch` on a variable (`currentLabel` for example) is placed inside a `while` loop. When a `goto` is required, the value of `currentLabel` is set to the destination label. Upon reaching the case statement, the method representing the section of code at the destination is called. Figure 4.3 gives the C code from Figure 3.3 showing an example of `goto`. Figure 4.4 shows similar code setting the label to `goto`, and Figure 4.5 shows the supporting code which invokes the method to goto.

The handling of subroutine calls requires minor changes to the goto handling. First the `currentLabel` variable becomes a stack, and each "`goto`" instruction calls the method on the top of the stack. When a subroutine is called, the label of the subroutine is pushed onto the stack, and the `goto` handler employed. When the return subroutine (`rts`) instruction is

[1] An LL(k) parser is a top down parser which parses the input from **L**eft to right and constructs a **L**eftmost derivation, with *k* tokens of look-ahead.

```
source:
    x = x + 1;
    goto destination;

destination:
    x = x - 1;
```

Figure 4.3: C code showing an example of a `goto`

```
method_source() {
    x = x + 1;
    currentLabel = destination;
    return;
}
method_destination() {
    x = x - 1;
}
```

Figure 4.4: Java code which approximates `goto` in conjunction with the code in Figure 4.5

executed, the stack is popped. This ensures that subroutine calls can be returned when there are `goto` statements inside the subroutine.

An alternative to SGJ's approach is to use Java's labelled `break` or `continue` within a loop, which is how Ephedra [11] handles `goto`s. However, since all of the flow control in the assembly programs is by the use of `goto`, it is simpler and more effective to use a `switch`.

4.4.2 Interrupts

As discussed in Chapter 3, interrupts are essentially unexpected and unpredictable changes in control flow. It is similar to the `Exception` in Java, and at first glance it seems that they can be simulated using exceptions. However, there are couple of problems that need to be dealt with in any solution:

- interrupts are generated by threads independent of the program thread, requiring inter-thread interruption
- interrupts are frequently not caused by errors but by the VDP, so program state needs to be the same before and after the interrupt.

One approach could be to check the status of some artificially created interrupt flags, and call the appropriate interrupt handler if an interrupt has occurred. However, this is clearly

```
while(true) {
    switch(currentLabel) {
        ...
        case destination:
            method_destination();
            break;
        ...
}
```

Figure 4.5: Java code for the `switch` to approximate the `goto` statement

an expensive way, both in terms of speed/efficiency and in terms of size of the compiled code.

SGJ does not deal with interrupts because it is outside the scope of the project.

4.4.3 High Level Constructs

High level constructs such as conditional and looping statements can be inferred using control flow analysis techniques [4, 5]. Various control flow analyses, such graph structuring, are quite effective for loop and conditional inference for the source assembly.

Inference of high level constructs are outside the scope of the project, due to the complexities of the task.

4.5 Solutions to Input/Output Issues

This sections explores some of the solutions to the Input/Output problems in Section 3.3.

4.5.1 Graphics

Sega Genesis graphics are emulated by SGJ using a Java `Graphics` object inside a Swing GUI. Graphics are bridged using a bridging interface, which has methods invoked from the translated program and operates on the `Graphics` object.

Graphics operations at the pixel level can be emulated by standard Java API methods, or by directly implement pixel based graphics operations on the `Graphics` object.

Graphics operations at the sprite level do not have a direct counterpart in Java's standard API, instead they need to be emulated such that the translated program interfaces with a sprite API which operates on the `Graphics` object.

4.5.2 Sound

Sega Genesis sound can be emulated using Java sound APIs. The midi interface provides enough functionality to emulate both the PSG and the FM sound found in the source hardware.

SGJ does not deal with sound, it is outside the scope of the project.

4.5.3 Input

Input is handled quite differently due to the joypad being polled in the user program rather than being set. Here SGJ's GUI has a keyboard listener, and when a preselected key is pressed, it is translated from it's Java AWT keycode into the Sega Genesis gamepad keycode.

4.6 Implementation

SGJ is essentially comprised of three parts. The first part is the parsing of the assembly source, and loading the representation into memory. The second (and main) part is the actual translation from assembly to Java. The third part is the GUI and execution of the translated program within the provided framework.

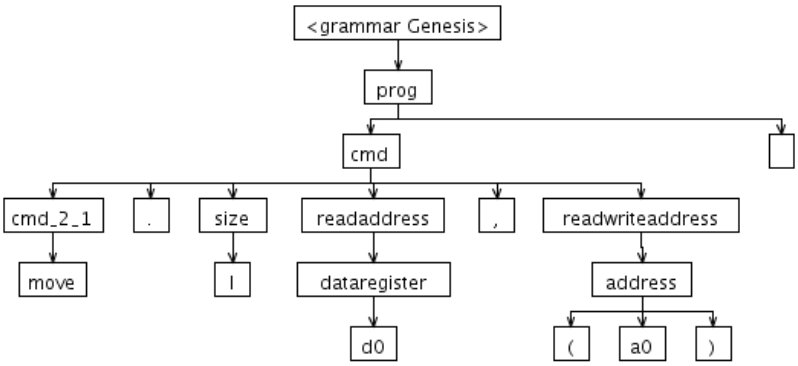


Figure 4.6: Abstract Syntax Tree for a `move` instruction

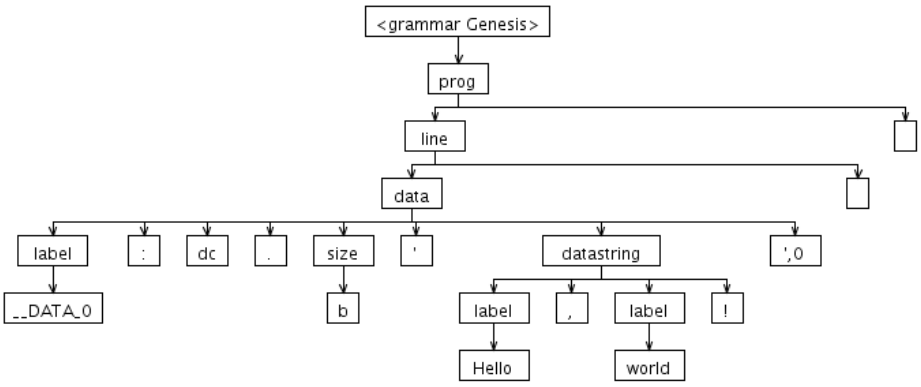


Figure 4.7: Abstract Syntax Tree for labelled data

4.6.1 Parsing

SGJ's parser is very straightforward, performing a single pass on the assembly source code to create the internal representation. As discussed previously, there are four different types of statements that can be parsed.

The first is the command type, which is an instruction, up to one size, and up to two operands. An example is `move.l d0, (a0)`, which is then parsed into the abstract syntax tree given in Figure 4.6. These statements are further distinguished by the mode and number of operands. For each command, a corresponding object is created in the internal representation, encapsulating the instruction, the size, and the operand(s).

The label type consists of just a string of characters, optionally followed by a colon. The form of a typical label is `__LABEL_drawloop`. This is also parsed into a object in the internal representation, and each label encapsulates all the commands between it and the very next label.

The third type, definition, is the assembly equivalent of a C `#define` macro. The form is the same for all instances of a definition, which is a string, then the keyword `EQU`, then the value. For example, `__INTEGER_ship EQU $FF0032` defines the value of `__INTEGER_ship` to be `0xff0032` in hexadecimal. Each define statement is parsed into a string → value mapping in the table of constants in the internal representation.

The final type is the labelled data. This is typically used to store ASCII strings in the assembly, and is structured as a label, followed by the instruction `dc.b`, followed by the string, and then a `',0'`. An example is the statement `__DATA_0: dc.b 'Hello, world!',0`,

essentially allowing the string "Hello, world!" to be referenced by the definition `__DATA_0` (Figure 4.7). Each labelled data statement is parsed such that the value of the label (such as `__DATA_0`) is an index into a data table, which contains the string. It is a string → index → string mapping.

4.6.2 Translation

Once the parsing is complete and the internal representation is built, SGJ executes the translation phase. The process is again quite simple, following this sequence of operations:

1. For each label statement in the set of parsed labels, visit the label.
2. Upon visiting a label, the signature for a new method of the name

   ```
   method_<label name>
   ```

   is created. Then for each command in the set of commands the label contains:

   (a) The method call to the Java version of the instruction for the command is output.

   (b) For each operand, the Java code for that operand is inserted as an argument to the method call.

   For example, the command

   ```
   move.l d0, a0
   ```

   translates into the Java

   ```
   DataMovement.move(Registers.d0, Registers.a0, Size.l);
   ```

   If the command being translated is a subroutine call, and the subroutine doesn't exist in the source program, because it is a library function (such as a graphics operation like `print_w`), then a nominal method is created in the same manner as for a label above. The implementation of these "missing" methods are then provided by the execution environment.
3. For each define statement in the set of parsed defines, output Java code to insert the <String, value> pair into the constants table.

   As an example, the define statement

   ```
   __INTEGER_i EQU $FF002E
   ```

   translates to the Java

   ```
   Constants.set("__INTEGER_i", Integer.parseInt("FF002E", 16));
   ```
4. For each labelled data statement in the set of parsed labelled data, output the Java code to insert the <String, index> pair into the constants table, and the <index, String> pair into the data table.

   An example labelled data statement is

   ```
   __DATA_0: dc.b 'Hello, world!',0
   ```

which becomes the Java statement

```
DataTable.set("__DATA_0", "Hello, world!");
```

where `DataTable.set` is defined as

```
public static void set(String name, String value) {
        Constants.set(name, idx);
        dataTable.add(idx++, value);
}
```

Once the assembly code has been translated, there is still some Java code required to be added to the translation to make a compilable and executable Java program. Apart from the library and framework required to run the translated program, the following needs to be added to the translation:

- Items such as the package declaration, and the import statements
- The class definition including `extends Thread`, since the translated program is executed in a multi-threaded environment.
- The declaration of class fields, such as the `Display` and `Input` object references.
- The constructor, which initialises fields, and executes the translated definition and labelled data statements.
- The `run` method, which calls the `gotoLabel` method.
- The `gotoLabel` method, which repeatedly calls the method specified by the current label using reflection.
- Finally comes the methods which have been translated from the labels as mentioned above.

Figure 4.8 is an example assembly program which prints 5 strings (see Figure 4.10). The translation of it using SGJ is shown in Figure 4.9.

4.6.3 Execution

The final part of SGJ is the execution environment (EE). This consists of a Swing based graphical interface, and the full set of libraries required to execute a translated program.

The graphical interface (Figure 4.10) provides a drop-down list to select the desired translated program to be run, and a set of control buttons. The run button launches the translated program (in a concurrent thread), and is disabled once the program has started execution. The stop button pauses execution of the program, and changes to a start button when the program is paused, which is used to resume the button. The quit button exits the execution environment.

```
        move.l  #0,d0
        move.w  d0,(__INTEGER_i)
__FOR_JUMP_0
        move.l  #__DATA_0,a0
        jsr     push_string
        jsr     popdisplay
        jsr     print_tab
        clr.l   d0
        move.w  (__INTEGER_i),d0
        jsr     print_w
        jsr     print_cr
__FOR_JUMP3_0:
        move.l  #5,d0
        cmp.w   (__INTEGER_i),d0
        beq     __FOR_JUMP2_0
        move.l  #1,d0
        add.w   d0,(__INTEGER_i)
        bra     __FOR_JUMP_0
__FOR_JUMP2_0:
forever:
        bra     forever
```

Figure 4.8: Source code excerpt of an assembly program which prints hello world. The full assembly code is listed in Appendix B.

```
  public void method_START() throws InterruptedException {
    DataMovement.move(0, Registers.d0, Size.l);
    DataMovement.move(Registers.d0, new LiteralPointer(Constants
        .get("__INTEGER_i")), Size.w);
    method___FOR_JUMP_0();
  }

  public void method___FOR_JUMP2_0() throws InterruptedException {
    method_forever();
  }

  public void method_forever() throws InterruptedException {
    currentLabel.pop();
    currentLabel.push("forever"); return;
  }

  public void method___FOR_JUMP3_0() throws InterruptedException {
    DataMovement.move(5, Registers.d0, Size.l);
    IntegerArithmetic.cmp(new LiteralPointer(Constants.get("
        __INTEGER_i")),
         Registers.d0, Size.w);
    if (Conditions.isSet(Conditions.Z)) {
      currentLabel.pop();
      currentLabel.push("__FOR_JUMP2_0"); return;
    }
    DataMovement.move(1, Registers.d0, Size.l);
    IntegerArithmetic.add(Registers.d0, new LiteralPointer(
        Constants
         .get("__INTEGER_i")), Size.w);
    currentLabel.pop();
    currentLabel.push("__FOR_JUMP_0"); return;
  }

  public void method___FOR_JUMP_0() throws InterruptedException {
    DataMovement.move(Constants.get("__DATA_0"), Registers.a0, Size
        .l);
    method_push_string();
    method_popdisplay();
    method_print_tab();
    IntegerArithmetic.clr(Registers.d0, Size.l);
    DataMovement.move(new LiteralPointer(Constants.get("__INTEGER_i
        ")),
         Registers.d0, Size.w);
    method_print_w();
    method_print_cr();
    method___FOR_JUMP3_0();
  }
}
```

Figure 4.9: Translated Java code of the assembly program given in Figure 4.8 (excerpt). The full Java source code is listed in Appendix C.

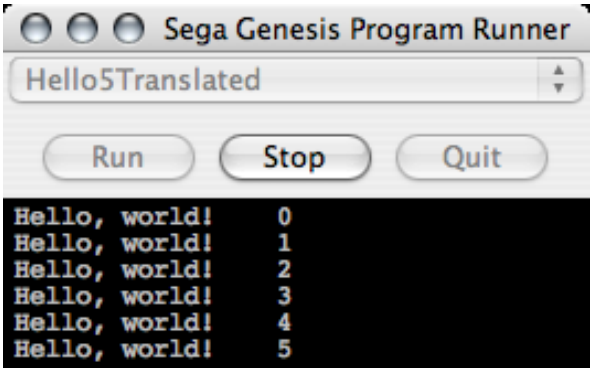


Figure 4.10: GUI for execution of translated programs

The execution environment, in addition to providing a graphical interface, provides implementations for all the machine instructions, as well as all the library calls for graphical functions. The following is a detailed description of the various functions provided.

Instructions The EE implements the instructions that are called from the user program. For example, if the user program calls

```
DataMovement.move(Registers.d0, Registers.a0, Size.l);
```

then the EE has an implementation of `DataMovement.move(Register, Register, Size);` which is given in Figure 4.11

Memory The EE provides memory read and write functions to user programs.

Resources The EE provides the Condition, Register, Pointer, Size, DataTable and Constants implementations, as emulations of the respective constructs in the source hardware.

Graphics The EE provides the implementation of the "missing" methods as discussed above, so that graphics operations in the user program are performed on the `Graphics` object in the GUI.

Input The EE listens for key presses and sets a pre-determined register to the Sega Genesis gamepad keycode according. The user program reads that pre-determined register to detect keypresses.

Machine The EE has a multi-threaded architecture to allow the user program to run independent of the GUI. This part allows for suspending and resuming the user program.

```
public static void move(Register r1, Register r2, Size s) {
  int val = r1.getValue(s);
  r2.setValue(val, s);
  setMoveCondition(val, s);
}
```

Figure 4.11: Execution environment implementation of a `move` instruction

Chapter 5

Demonstration

This chapter discusses the progress made on SGJ, including the number of instructions supported and the level of support for library functions. The final section walks through some example programs and their translations.

5.1 Instruction Set

The Motorola 68000 instruction set is supported at varying levels throughout SGJ. The following sections discuss the coverage in stages.

5.1.1 Parser

The parser has support for the core instruction set, which consists of data movement, arithmetic, system control, logical and shift/rotate operations. All of these core instructions are parsed fully. In addition, other assembly source code, such as labels, definitions and labelled data is also fully parsed.

Adding support for the non-core instructions, such as floating point instructions, can be as simple as adding those instructions to the set of instructions the parser knows about.

5.1.2 Translator

The translator supports the core instruction set as well. Each instruction is replaced by the appropriate SGJ library method call, meaning that adding translation support for other (non-core) instructions is straightforward.

5.1.3 Execution Environment

The execution environment has full support for the core instructions. Each instruction has a corresponding implementation in the SGJ library.

The ability to add support for non-core instructions is varied. New data movement and other similar simple instructions are straightforward to implement. More complex instructions requiring library support, such as interrupts, will clearly require a great deal of work to add support for.

5.2 Implementation of Proposed Solutions

The progress made in the project on SGJ is outlined in the following sections.

5.2.1 Goto

The solution for `goto` that SGJ implements is complete, in that all five of the supported control flow statements, as per Figure 3.2, are functional.

5.2.2 Memory

The memory emulation in SGJ is fully functional, as are pointers. The only check SGJ does on memory access is whether the address is inside the addressable range.

Pointers are fully functional as well, including post-increment and pre-decrement operations, allowing memory to be treated as a stack.

5.2.3 Graphics

The execution environment has text-based graphics operations support, so simple commands such as `print_w`, which prints a `word` of data as a number, or `print_string`, which prints an ASCII String, are fully supported. Other operations are supported, such as relocating the cursor on the screen, and clearing the screen.

5.2.4 Input

SGJ's GUI allows for the detection of any keyboard key press (including modifiers), however this is more than is needed for the execution environment. Instead, an arbitrary mapping of keys to gamepad buttons is made, with a reasonable mapping for up/down/left/right on the keyboard to correspond to the same directional codes in SGJ. Translated programs can discover the key press(s) performed by the user in the same manner as the original program.

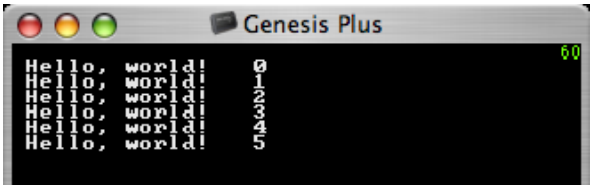
5.3 System Tests

This section looks at some of the test programs successfully translated by SGJ, including the level of support for various libraries.

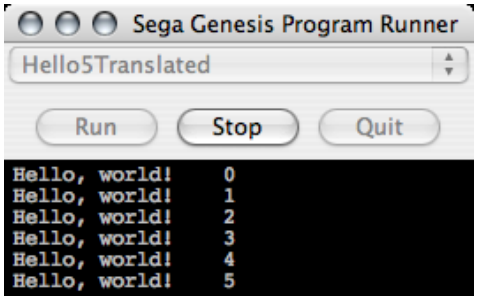
5.3.1 `Hello, world!`

The first example program demonstrates the support for looping, as well as memory access, constants and printing of strings. It writes the string "Hello, world!", followed by a tab, followed by an incrementing index from 0 to 5. The index is stored in memory, and the address of it is a definition.

The first screenshot shows the original program running on the Genesis Plus [2] emulator.



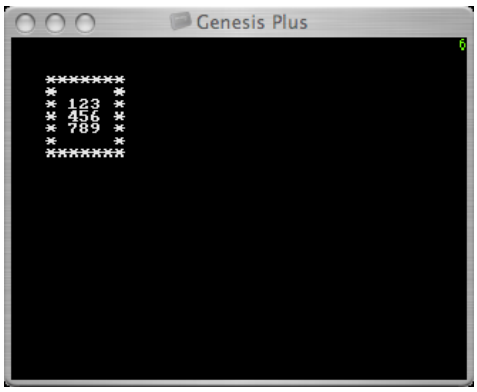
This is the screenshot of the program running on SGJ's GUI, after being translated by SGJ.

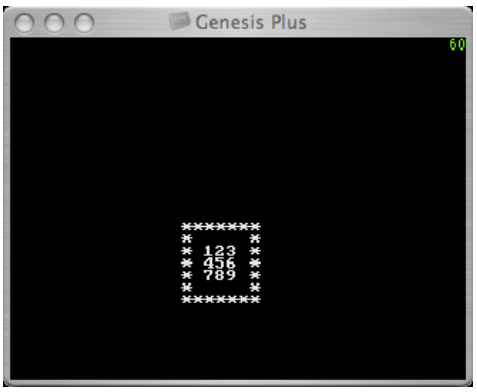


5.3.2 Cursor Movement

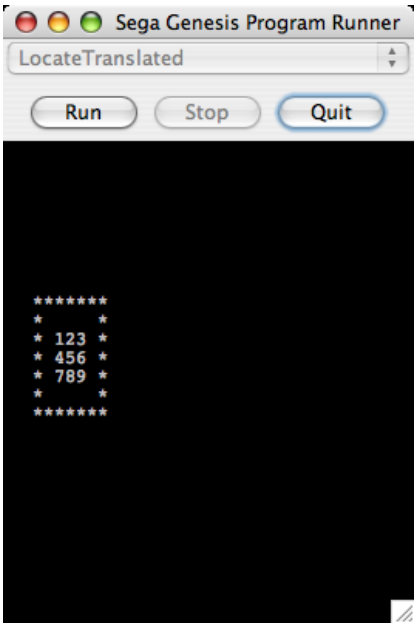
This is an example of the ability to move the cursor on the screen. It starts the cursor on the top-left of the display, progressing diagonally down to the bottom-right. The screen is cleared before every relocation of the cursor. The example shows the ability to locate the cursor and to clear the screen.

These first screenshots are of the original program on the emulator:





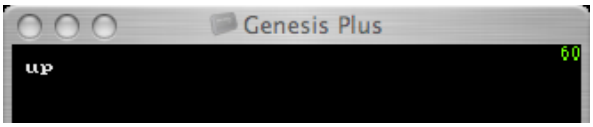
These screenshots are of the translated program on SGJ's GUI:



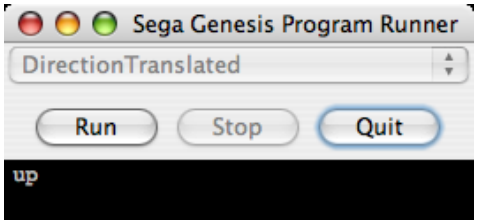
5.3.3 Input detection

This final example is a program which reads the user input and displays the direction on the screen. The example shows the ability to detect the key direction pressed and relay the information to the user program.

The first screenshot is the original program on the emulator:



Finally this screenshot is the translated program running on SGJ's GUI:



Chapter 6

Summary

This chapter recaps the major findings in the project, makes some recommendations for future work, and reiterates the contributions.

6.1 Major Findings

There were a number major findings in this project. One finding is the difficulties in converting from a low level assembly language into a high level language, such as Java. There are many intricacies in the conversion, stemming from the language differences listing in Section 2.5, including typing of data, control flow and primitive data types.

Another finding is the complications that arise from converting a language written natively for an architecture into a general purpose multi-architecture language. Hardware supported constructs, such as condition checking and direct memory manipulation, are missing from Java, and at best can only be emulated.

6.2 Future Work

An obvious improvement to SGJ would implement the remaining instructions to give it support for the full instruction set. This would give it the ability to support a wider range of programs.

Full graphics support would also be a great improvement, meaning that SGJ can be tested properly. It would also bring SGJ closer to the original look of the programs. A number of options are available for implementation, including the J2ME [16] framework, which provides a number of common graphics operations, such as sprites, which in turn would make the graphics emulation in SGJ much easier.

Sound support in SGJ would also bring it closer to the goal of the original look and feel. Java provides a sound API, and this can be leveraged to provide sound support in SGJ.

Interrupt support is a needed feature missing from SGJ at the moment. It will be required to support graphics, and will allow for a more complete translation of user programs.

Finally, while SGJ translates assembly for programs compiled from BEX, adding support for decompilation of binaries would be a great step towards supporting original programs for SGJ.

6.3 Contributions

There are two main contributions of this project, a methodology for program conversion, and a prototype conversion tool. The methodology outlines the main issues in dealing with a conversion from a low-level assembly to a high-level language, such as Java. It also suggests approaches to solving the issues.

The second contribution is SGJ, a prototype tool to convert Motorola 68000 assembly code into Java source code. SGJ implements some of the solutions proposed by the methodology, and the Java source code executes on the standard Java virtual machine, version 1.5.

Appendix A

68000 Grammar

The following is the complete EBNF grammar file for the Motorola 68000 developed as part of the project. It is compatible with ANTLR [15] version 3.

```
grammar Genesis;

tokens {
    ADD='add';
    ADDQ='addq';
    AND='and';
    BRA='bra';
    BEQ='beq';
    BNE='bne';
    BSR='bsr';
    BTST='btst';
    CLR='clr';
    CMP='cmp';
    DIVU='divu';
    JMP='jmp';
    JSR='jsr';
    LEA='lea';
    LSL='lsl';
    LSR='lsr';
    MOVE='move';
    MOVEM='movem';
    MULU='mulu';
    NOP='nop';
    OR='or';
    ROR='ror';
    RTS='rts';
    SEQ='seq';
    SCS='scs';
    SHI='shi';
    SNE='sne';
    SUB='sub';
    SUBQ='subq';
    SWAP='swap';
    TST='tst';
```

```
    OPT='OPT';
    INCLUDE='include';
    EVEN='EVEN';
    END='END';

    EQU='EQU';
}

@lexer::header {
        package sgj.core;
    }

    @header {
        package sgj.core;
        import sgj.parser.commands.*;
        import sgj.parser.operands.*;
        import sgj.parser.misc.*;
        import java.util.Vector;
    }

    @members {
        Stack<Label> labels = new Stack<Label>();
        Vector<ParserConstant> constants = new Vector<
           ParserConstant>();
        Vector<ParserData> dataTable = new Vector<ParserData>()
           ;
    }


    prog:   {labels.push(new Label("START"));} line* cmd? EOF
  ;

line:   (cmd|lbl|equ|data)? EOL
    ;

lbl     :       label ':'? {labels.push(new Label($label.text));}
    ;

equ     :       label EQU effectiveaddress {constants.add(new
   ParserConstant($label.text, new LiteralAddress("#" +
   $effectiveaddress.text)));}
    ;

data
    :        label ':' 'dc'.size '\''datastring'\',0' {
       dataTable.add(new ParserData($label.text, $datastring.
       text)); }
    ;
```

```
datastring
        :       (label|dataregister|addrregister|literalinteger|
           effectiveaddress|','|'!'|'>'|'*')+
        ;

cmd
@init {
    String sz = null;
    String al = null;
}
    :   c=cmd_2_1 ('.' s=size)? o1=readaddress ',' o2=
           readwriteaddress {if (s!=null) sz=$s.text; labels.peek().
           addCommand(new Command_2_1(c, sz, o1, o2));}
        |       c=cmd_1_x '.' size o1=readaddress ',' addresslist {
           Operand op = new AddressList($addresslist.text); labels.
           peek().addCommand(new Command_1_x(c, $size.text, o1, op)
           );}
        |       c=cmd_x_1 '.' size addresslist ',' o2=writeaddress
           {Operand op = new AddressList($addresslist.text); labels
           .peek().addCommand(new Command_x_1(c, $size.text, op, o2
           ));}
        |       c=cmd_n_1 (effectiveaddress|label) ',' addrregister
            {Operand op1 = new EffectiveAddress($effectiveaddress.
           text); Operand op2 = new AddressRegister($addrregister.
           text); labels.peek().addCommand(new Command_n_1(c, op1,
           op2));}
        |       c=cmd_1_1 o1=readwriteaddress {labels.peek().
           addCommand(new Command_1_1(c, o1));}
        |       c=cmd_1_0 o1=readaddress {labels.peek().addCommand(
           new Command_1_0(c, o1));}
        |       c=cmd_0_1 ('.' s=size)? o1=writeaddress {if (s!=
           null) sz=$s.text; labels.peek().addCommand(new
           Command_0_1(c, sz, o1));}
        |       c=cmd_0_0 {labels.peek().addCommand(new Command(c))
           ;}
        |       c=cmd_extra a=arg? {if (a!=null) sz=$a.text; labels
           .peek().addCommand(new Command_Extra(c, sz));}
        ;

cmd_2_1 returns [int i = 0]
        :       ADD {i = ADD;}
    |   ADDQ {i = ADDQ;}
    |   AND {i = AND;}
    |   BTST {i = BTST;}
    |   CMP {i = CMP;}
    |   DIVU {i = DIVU;}
    |   LSL {i = LSL;}
    |   LSR {i = LSR;}
    |   MOVE {i = MOVE;}
    |   MULU {i = MULU;}
```

```
    |   OR {i = OR;}
    |   ROR {i = ROR;}
    |   SUB {i = SUB;}
    |   SUBQ {i = SUBQ;}
        ;

cmd_1_x returns [int i = 0]
        :       MOVEM {i = MOVEM;}
        ;

cmd_x_1 returns [int i = 0]
        :       MOVEM {i = MOVEM;}
        ;

cmd_n_1 returns [int i = 0]
        :       LEA {i = LEA;}
        ;

cmd_1_1 returns [int i = 0]
        :       SWAP {i = SWAP;}
        ;

cmd_1_0 returns [int i = 0]
        :       BRA {i = BRA;}
    |   BEQ {i = BEQ;}
    |   BNE {i = BNE;}
    |   BSR {i = BSR;}
    |   JMP {i = JMP;}
    |   JSR {i = JSR;}
        ;

cmd_0_1 returns [int i = 0]
        :       CLR {i = CLR;}
    |   SCS {i = SCS;}
    |   SEQ {i = SEQ;}
    |   SHI {i = SHI;}
    |   SNE {i = SNE;}
    |   TST {i = TST;}
        ;

cmd_0_0 returns [int i = 0]
        :       NOP {i = NOP;}
    |   RTS {i = RTS;}
        ;

cmd_extra returns [int i = 0]
        :       OPT {i = OPT;}
    |   INCLUDE {i = INCLUDE;}
    |   EVEN {i = EVEN;}
    |   END {i = END;}
```

```
        ;

arg     :       LABEL '+' | LABEL '.' (LABEL|size)
        ;

size:   'b'|'B'|'w'|'W'|'l'|'L'|'s'|'S'|'d'|'D'|'x'|'X'|'p'|'P'
        ;

readaddress returns [Operand o]
        :       dataregister {o = new DataRegister($dataregister.
            text);}
        |       addrregister {o = new AddressRegister($addrregister
            .text);}
        |       address {o = new Address($address.text);}
        |       effectiveaddress {o = new EffectiveAddress(
            $effectiveaddress.text);}
        |       constant {o = new Constant($constant.text);}
        |       l=literal {o = l;}
        |       label {o = new sgj.parser.operands.Label($label.
            text);}
        ;

writeaddress returns [Operand o]
        :       dataregister {o = new DataRegister($dataregister.
            text);}
        |       addrregister {o = new AddressRegister($addrregister
            .text);}
        |       address {o = new Address($address.text);}
        ;

readwriteaddress returns [Operand o]
        :       dataregister {o = new DataRegister($dataregister.
            text);}
        |       addrregister {o = new AddressRegister($addrregister
            .text);}
        |       address {o = new Address($address.text);}
        |       effectiveaddress {o = new EffectiveAddress(
            $effectiveaddress.text);}
        |       constant {o = new Constant($constant.text);}
        ;

addresslist
        :       readwriteaddress(SLASH(addressrange|
            readwriteaddress))+
        |       addressrange(SLASH(addressrange|readwriteaddress))*
        ;

addressrange
        :       (ADDRREG|DATAREG)'-'(ADDRREG|DATAREG)
        ;
```

```
dataregister
        :       DATAREG
        ;

address
        :       '(' (ADDRREG|ADDRESS|composite) ')'
        |       '-' '(' ADDRREG ')'
        |       '(' ADDRREG ')' '+'
        ;

composite
        :       dataregister '.' size ',' addrregister
        ;

effectiveaddress
        :       ADDRESS
        ;

addrregister
        :       ADDRREG
        ;

ADDRREG
    :   ('a'|'A') '0'..'7'
    ;

ADDRESS
        :       '$' HEX+
        ;

DATAREG
        :       ('d'|'D') '0'..'7'
        ;

constant
        :       '(' label ')'
        ;

literal returns [Literal l]
        :       '#' (
                literallabel {l = new LiteralLabel("#" +
                   $literallabel.text);}
        |       literalinteger {l = new LiteralInteger("#" +
           $literalinteger.text);}
        |       effectiveaddress {l = new LiteralAddress("#" +
           $effectiveaddress.text);}
                )
        ;
```

```
literallabel
        :       label
        ;

label
    :   LABEL
        ;

LABEL
    :   ('b'..'c'|'e'..'z'|'B'..'C'|'E'..'Z'|'_') ('a'..'z'|'A'..'Z
        '|'_'|'0'..'9')*
    |   ('a'|'A'|'d'|'D') ('a'..'z'|'A'..'Z'|'_'|'8'|'9') ('a'..'z
        '|'A'..'Z'|'_'|'0'..'9')*
    |   ('a'|'A'|'d'|'D') ('0'..'7') ('a'..'z'|'A'..'Z'|'_
        '|'0'..'9')+
        ;

fragment DIGIT
    : ('0'..'9');

literalinteger
        :       INT
        ;

INT : ('-')? (DIGIT)+
    ;

fragment HEX
        :       '0'..'9'|'a'..'f'|'A'..'F'
    ;

SLASH
        :       '/'
        ;

EOL     :       (('\r')? '\n')
        ;

WS      :       (' '|'\t'|'\u000C') { $channel=HIDDEN; }
        ;
```

Appendix B

Hello World Program (Assembly)

This is the assembly code for the example "Hello world" program used throughout the document.

```
        move.l  #0,d0
        move.w  d0,(__INTEGER_i)
__FOR_JUMP_0
        move.l  #__DATA_0,a0
        jsr     push_string
        jsr     popdisplay
        jsr     print_tab
        clr.l   d0
        move.w  (__INTEGER_i),d0
        jsr     print_w
        jsr     print_cr
__FOR_JUMP3_0
        move.l  #5,d0
        cmp.w   (__INTEGER_i),d0
        beq     __FOR_JUMP2_0
        move.l  #1,d0
        add.w   d0,(__INTEGER_i)
        bra     __FOR_JUMP_0
__FOR_JUMP2_0
forever
        bra     forever

        EVEN

__INTEGER_i     EQU     $FF002E
HEAPSTART       EQU     $FF0030
__DATA_0        dc.b    'Hello, world!',0
        include basicstr.s
        EVEN
        include basicdat.s
        EVEN

__DATA_Pendcode
        END
```

Appendix C

Hello World Program (Java)

This is the example program "Hello world" used throughout the document. This version is the Java source code produced by SGJ.

```
package sgj.machine.translation;

import java.util.Stack;

import sgj.machine.instructions.DataMovement;
import sgj.machine.instructions.IntegerArithmetic;
import sgj.machine.library.Display;
import sgj.machine.library.Input;
import sgj.machine.library.Machine;
import sgj.machine.resources.Conditions;
import sgj.machine.resources.Constants;
import sgj.machine.resources.DataTable;
import sgj.machine.resources.LiteralPointer;
import sgj.machine.resources.Registers;
import sgj.machine.resources.Size;

public class Hello5Translated extends Thread {
  private Display display;
  private Input input;
  private Stack<String> currentLabel;

  public Hello5Translated(Display d, Input i, String l) {
    this.display = d;
    this.input = i;
    this.currentLabel = new Stack<String>();
    this.currentLabel.push(l);
    Constants.set("__INTEGER_i", Integer.parseInt("FF002E", 16));
    Constants.set("HEAPSTART", Integer.parseInt("FF0030", 16));
    Constants.set("CURSORX", Integer.parseInt("FF0000", 16));
    Constants.set("CURSORY", Integer.parseInt("FF0001", 16));
    DataTable.set("__DATA_0", "Hello, world!");
  }

  @Override
```

```
public void run() {
  try {
    gotoLabel();
  }
  catch (InterruptedException ie) {
  }
}

public void gotoLabel() throws InterruptedException {
  while (true) {
    Machine.machine().step();
    try {
      Thread.sleep(100);
      this.getClass().getMethod("method_" + currentLabel.peek()).
          invoke(this);
    }
    catch (InterruptedException ie) {
      throw ie;
    }
    catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}

public void method_print_tab() throws InterruptedException {
  display.print_tab();
}

public void method_popdisplay() throws InterruptedException {
  display.popdisplay();
}

public void method_START() throws InterruptedException {
  DataMovement.move(0, Registers.d0, Size.l);
  DataMovement.move(Registers.d0, new LiteralPointer(Constants
      .get("__INTEGER_i")), Size.w);
  method___FOR_JUMP_0();
}

public void method___FOR_JUMP2_0() throws InterruptedException {
  method_forever();
}

public void method_push_string() throws InterruptedException {
  display.push_string();
}

public void method_forever() throws InterruptedException {
```

```
    currentLabel.pop();
    currentLabel.push("forever");
    return;
  }

  public void method_print_w() throws InterruptedException {
    display.print_w();
  }

  public void method___FOR_JUMP3_0() throws InterruptedException {
    DataMovement.move(5, Registers.d0, Size.l);
    IntegerArithmetic.cmp(new LiteralPointer(Constants.get("
        __INTEGER_i")),
         Registers.d0, Size.w);
    if (Conditions.isSet(Conditions.Z)) {
      currentLabel.pop();
      currentLabel.push("__FOR_JUMP2_0");
      return;
    }
    DataMovement.move(1, Registers.d0, Size.l);
    IntegerArithmetic.add(Registers.d0, new LiteralPointer(
        Constants
          .get("__INTEGER_i")), Size.w);
    currentLabel.pop();
    currentLabel.push("__FOR_JUMP_0");
    return;
  }

  public void method___DATA_Pendcode() throws InterruptedException
      {
    System.exit(0);
  }

  public void method_screen_cls() throws InterruptedException {
    display.screen_cls();
  }

  public void method_joypad() throws InterruptedException {
    input.joypad();
  }

  public void method_delay() throws InterruptedException {
    Machine.machine().delay();
  }

  public void method___FOR_JUMP_0() throws InterruptedException {
    DataMovement.move(Constants.get("__DATA_0"), Registers.a0, Size
        .l);
    method_push_string();
    method_popdisplay();
```

```
    method_print_tab();
    IntegerArithmetic.clr(Registers.d0, Size.l);
    DataMovement.move(new LiteralPointer(Constants.get("__INTEGER_i
        ")),
         Registers.d0, Size.w);
    method_print_w();
    method_print_cr();
    method___FOR_JUMP3_0();
  }

  public void method_print_cr() throws InterruptedException {
    display.print_cr();
  }
}
```

Bibliography

[1] ANCKAERT, B., MADOU, M., AND BOSSCHERE, K. D. *Information Hiding*, vol. 4437/2007 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, ch. A Model for Self-Modifying Code, pp. 232–248.

[2] BANNISTER, R. F. Genesis plus. `http://www.bannister.org/software/gplus.htm`. Accessed 12th October 2007.

[3] CAI, H., SHAO, Z., AND VAYNBERG, A. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), ACM Press, pp. 66–77.

[4] CIFUENTES, C. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[5] CIFUENTES, C., SIMON, D., AND FRABOULET, A. Assembly to high-level language translation. In *ICSM '98: Proceedings of the International Conference on Software Maintenance* (Washington, DC, USA, 1998), IEEE Computer Society, p. 228.

[6] DICKENS, T. Migrating legacy engineering applications to java. In *OOPSLA '02: OOPSLA 2002 Practitioners Reports* (New York, NY, USA, 2002), ACM Press, pp. 1–ff.

[7] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] KÅGSTRÖM, S., GRAHN, H., AND LUNDBERG, L. Cibyl: an environment for language diversity on mobile devices. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM Press, pp. 75–82.

[9] LINDHOLM, T., AND YELLIN, F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] LOVEMAN, D. B. Program improvement by source-to-source transformation. *J. ACM 24*, 1 (1977), 121–145.

[11] MARTIN, J. *Ephedra - A C to Java Migration Environment*. PhD thesis, Northern Illinois University, 1996.

[12] MOTOROLA INC. *Motorola M68000 Family Programmer's Reference Manual*, 1992.

[13] NATIONAL LIBRARY OF AUSTRALIA. Digital preservation strategies. `http://www.nla.gov.au/padi/topics/18.html`. Accessed 12th October 2007.

[14] NORMAN, J. Basiegaxorz. `http://devster.monkeeh.com/sega/basiegaxorz/`. Accessed 12th October 2007.

[15] PARR, T. Antlr. http://www.antlr.org/about.html. Accessed 12th October 2007.

[16] SUN MICROSYSTEMS. Java ME. http://java.sun.com/javame/index.jsp. Accessed 12th October 2007.

[17] VAN DEN BRAND, M. G. J., KLINT, P., AND VERHOEF, C. Reverse engineering and system renovation – an annotated bibliography. *SIGSOFT Softw. Eng. Notes 22*, 1 (1997), 57–68.

[18] VMWARE. VMWare. http://www.vmware.com/. Accessed 12th October 2007.

[19] WELLS, D. Extreme programming. http://extremeprogramming.org/rules.html. Accessed 12th October 2007.