

Motorola Simulator Project Report

The 68k/Sim Team

May 9, 2002

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 1.1 | Background and Problem Statement | 3 |
| 1.2 | Technical Approach | 3 |
| 1.2.1 | Representation of the Microprocessor and Memory | 4 |
| 1.2.2 | S-Record parsing | 4 |
| 1.2.3 | Instruction decoding and execution | 4 |
| 1.2.4 | Instruction translation | 5 |
| 1.2.5 | Implementation of the help system | 5 |
| 1.2.6 | GUI Implementation | 5 |
| 1.2.7 | Undo functionality | 5 |
| 1.3 | Division of Tasks | 6 |
| 1.3.1 | Phase 0—Initial Code Framework | 6 |
| 1.3.2 | Phase 1—Execution and Translation Code | 6 |
| 1.3.3 | Phase 2—Graphical User Interface | 7 |
| 1.3.4 | Phase 3—Context Sensitive Help System | 7 |
| 2 | Requirements | 8 |
| 2.1 | Functional Requirements | 8 |
| 2.2 | Non-functional Requirements | 8 |
| 2.3 | User Interaction | 8 |
| 3 | Design | 9 |
| 3.1 | Architecture | 9 |
| 3.1.1 | Core Emulation Engine | 9 |
| 3.1.2 | GUI | 9 |
| 3.2 | Algorithm and Class Design | 10 |
| 3.2.1 | Class Design | 10 |
| 3.2.2 | Algorithm Design | 11 |
| 3.2.3 | RestoreState (Undo) | 12 |
| 4 | Implementation | 13 |
| 4.1 | Tools Used | 13 |
| 4.1.1 | QT GUI Libraries | 13 |
| 4.1.2 | CVS Code Management | 14 |
| 4.1.3 | Mailman Mailing List | 14 |
| 4.1.4 | Pair Programming | 14 |
| 4.2 | User Interface Implementation | 14 |
| 4.2.1 | QMTable | 14 |
| 4.2.2 | Opcode template viewer | 15 |
| 4.2.3 | CCR flags | 15 |

| | | |
|----------|--|-----------|
| 4.2.4 | Updating the GUI | 15 |
| 4.3 | Implementation of Algorithms | 15 |
| 4.3.1 | Execution and Translation | 15 |
| 5 | Conclusions | 16 |
| 5.1 | Design and Implementation Problems | 16 |
| 5.2 | Completion of Project Objectives | 16 |
| 5.2.1 | Supervisor's Requirements | 16 |
| 5.2.2 | Additional Objectives | 16 |
| 5.3 | How we worked as a group | 16 |
| 5.4 | Future Enhancements | 17 |
| 5.4.1 | Solve some remaining issues | 17 |
| 5.4.2 | Make the tool useful for 2BA4 students | 17 |
| 5.4.3 | Provide assembling functions | 17 |

Chapter 1

Introduction

This document forms the Final Group Report for **Group B** and details the requirements, design and implementation of **68K/Sim**, a cross-platform Motorola 68008 Simulator written as part of the 2BA7 course.

The group consisted of Diarmuid Power, Annie Bedford, Gerard Whyte, Alan Donnelly, Laura Redmond and Edsko de Vries and was supervised by Michael Manzke.

1.1 Background and Problem Statement

"Design and implement a simulation and animation that emulates the 68008-instruction set." Particular emphasis was to be placed on ensuring that the finished application could be used as a teaching tool for beginners to Motorola assembly programming, particularly first-year Computer Science students studying Computing (1BA3).

We decided that the program should be able to read programs compiled to Motorola S-Record format (this format is used by the 1BA3 development environment), allow the user to through their program and give a visual representation of the actions taking place in the entire system. We also felt that a help system that would flag errors and give a useful explanation of the error should be implemented. Undo functionality we thought would be an extremely useful addition to the application, allowing the user to undo the effects of an instruction, change the value of a register or the status register and execute the instruction again. After consulting with our supervisor we decided that animation of CPU operations should be omitted in order to provide a more functional program for students (such as providing an undo function and implementing the context sensitive help system).

1.2 Technical Approach

Having formed a clear definition of the task, we identified the following key problems which would need to be addressed:

1. Representation of the Microprocessor and Memory
2. S-Record parsing
3. Instruction decoding and execution
4. Instruction translation
5. Implementation of the help system
6. Undo functionality
7. GUI Implementation

1.2.1 Representation of the Microprocessor and Memory

We decided to use separate classes to represent each part of the microprocessor system (the CPU and attached memory in the current implementation). Within the CPU class, a specialised structure to represent the registers would also be required. The memory would be implemented in a separate class and would provide access to any given amount of virtual memory in the microprocessor system (currently the application provides 1MB of memory).

The main requirement of the CPU class was to provide functionality to execute a single instruction at any given memory location. In addition, the class needed to be able to translate an instruction independently of execution and to provide undo functionality. The execution process was to consist of decoding the data at the location specified by the program counter, updating the operands involved in the instruction according to the function the instruction performs (e.g. addition in the case of `add.b d0,d1`) and setting the status register according to the result. Errors and potential errors also needed to be detected during execution. To implement translating, the CPU needed to be able to generate a string containing the assembler syntax for the instruction. Finally, in order to provide undo functionality, the CPU needed to store the changes to system after executing each instruction.

Slightly less complicated than the CPU class, the class representing the attached memory needed to provide functionality to load s-records, return a byte, word or longword from a given location in memory and write any memory access errors to the help stack.

(See **Design: Algorithm and Class Design** for more information on the implementation of these classes and execution, translation and undo algorithms.)

1.2.2 S-Record parsing

We decided to use Motorola's S-Record format as the file format of our application since this is the file format used by the majority of compilers for the Motorola microprocessor platform. Furthermore, our supervisor had requested that the application should display the instructions that the compiler used, rather than those originally used by the student (see Functional Requirements). This was only possible by parsing the S-Record file. S-Records encode instructions in an ASCII string which contains information such as the type of S-Record, where it should be located in memory and its length followed by the instructions encoded, or an initialisation value for the Program Counter. These ASCII strings needed to be decoded into a binary representation which could then be further decoded and 'executed' by the application.

1.2.3 Instruction decoding and execution

The binary data resulting from parsing the s-records needed to be further interpreted as Motorola instructions and then executed on our emulated microprocessor. We decided that it should be possible for the application to interpret any part of memory as an instruction. Given a starting point in memory, initial decoding would then take place where the instruction is identified according to its unique signature. (For usability, this starting point could be automatically set when opening compiled S-Records.)

Having identified the instruction, the instruction word then needed to be further decoded to extract the registers and memory involved in the instruction along with the addressing mode being used and any immediate data following the instruction. This would be done by separate methods for each instruction which would update the system according to the function of the instruction (e.g. add, subtract, clear etc...), set the status flags accordingly and update the program counter according to the size of the instruction and any immediate data. Given the updated program counter, the application would now have a starting point to begin decoding and executing the next instruction. Due to the possibility of immediate data following the instruction, we could not assume a fixed size for instructions and this had to be accounted for during decoding and execution

(See **Design: Algorithm and Class Design** for further information on the implementation of the decoding and execution engine)

1.2.4 Instruction translation

As well as executing the an instruction at a given memory location, we also needed to decode the instruction data into human readable form. This was quite important, since there would otherwise be no easy way of knowing what instruction was currently executing.

As with instruction execution, we wanted to have the ability to translate any address in memory and interpret it as an instruction, so we used the same identification and decoding process as used for instruction execution. Since we would know registers, memory and addressing mode involved, it would just be a matter of returning the appropriate to return a string representation of the instruction word at the current memory location. It was important that translation did not update the emulated system in any way, so in order to know where the next instruction began, we decided translation methods should return the address of the next instruction, rather than updating the program counter.

(See Design: Algorithm and Class Design for further information on the implementation of the translation engine)

1.2.5 Implementation of the help system

One of the most confusing parts of the 1BA3 course for students was the way in which errors were reported to the user. Typically, an erroneous program would crash and output an error message that didn't always give an immediate indication of what the problem was or where it occurred (students would have to inspect the program counter and work out from that where the error happened). We wanted to improve upon the error reporting system already in place for 1BA3 students by providing a more verbose message as well as an indication of where the error occurred. We also wanted to report potential errors (such not popping all operands from the system stack when returning from a subroutine) and allow the user to continue executing the erroneous code (this of course would only be possible in the emulator).

Resulting from class feedback at the first Progress Report Presentation, we decided to use a stack data structure (since we would frequently need to have access to the last errors generated) to store any generated errors. We proceeded to modify the methods used to read and modify our virtual memory so that they checked for address errors and bus errors and pushed the appropriate error message to the 'help stack.' The methods used to execute instructions that altered the system stack (mostly move and movem) or were involved with subroutines (bx and rts) or any other instructions that would be sources of potential errors, were also modified to push errors to the help stack.

1.2.6 GUI Implementation

The main requirement of the GUI was to demystify the operation of the M68008 microprocessor system by providing an easy to understand visualisation of the components of the microprocessor system. This would include the contents of memory, the status register, instruction register and the program counter and the system stack. In addition we needed to show clearly what the emulated system was currently executing and for this we included a 'program viewer' which showed the memory address of instructions in memory, the hex number representing the instruction's opcode and the translated string associated with the instruction. The help system would also need to be visible at all times and we decided it would be useful to show the opcode template of the instruction currently being executed (the template describes the meaning of each bit in an instruction word).

(See **Implementation: User Interface Implementation** for further information)

1.2.7 Undo functionality

Being able to undo the effect of executing an instruction was a major feature of our program. In order to be able to undo an instruction, the entire state of the system before executing the instruction would have to be saved, and then restored again when undoing that instruction. Due to limited nature of computer resources, we could not of course save the contents of the *entire* system, since this would require little over a megabyte of memory per instruction in order to save the contents of the virtual memory. A compromise needed to be reached and we decided to save

enough changes so that the maximum amount of memory changeable by an instruction (16 longwords by movem) would be saved. This would of course would have to exclude any changes made directly in the memory viewer. Similarly the number of changes to the system stack would also have to be limited. For convenience, a specialised data structure was also needed to store the changes to the system state.

1.3 Division of Tasks

As we opted to apply the pair programming technique promoted by Extreme Programming, the tasks within the project were not divided up individually but per pair (this was cleared by both the project coordinator). The team was split up in three pairs:

Team A/G Alan Donnelly and Gerard Whyte

Team D/A Diarmuid Power and Annie Bedford

Team L/E Laura Redmond and Edsko de Vries

Below the division of tasks is outlined per phase of the project. Note that phase 2 (GUI) and phase 3 (help system) were merged in the end; phase 3 turned out to be smaller than phase 2; hence two teams worked on phase 2 and one team worked on phase 3. Code is annotated in parts but this is by no means complete; this section should be the final guide.

1.3.1 Phase 0—Initial Code Framework

This phase consisted of defining the framework for the project; the CPU class, the memory class, instruction decoding (but, obviously, not executing) and S-record parsing. This phase was completed by **Team L/E**.

1.3.2 Phase 1—Execution and Translation Code

This phase was by far the largest phase (approximately 22,000 of code in the files which constitute this phase alone!).

The Execution Engine

The execution engine was divided up as follows:

Team A/G: abcd(), add(), addi(), addq(), addx(), and(), andi(), andiCcr(), asl_rReg(), asl_rMem(), bxx(), bchgStat(), bchgDyn(), bclr(unsigned char Bit), bclrStat(), bclrDyn(), bkpt(), bra(), bset(unsigned char Bit), bsetStat(), bsetDyn(), bsr(), btst(unsigned char Bit), btstStat(), btstDyn(), chk(), clr(), cmp(), cmpa(), cmpi() and cmpm().

Team D/A: dbxx(), divsWord(), divsLong(), divuWord(), divuLong(), eor(), eori(), eoriCcr(), exg(), ext(), illegal(), jmp(), jsr(), lea(), linkWord(), linkLong(), lsl_rReg(), lsl_rMem(), move(), movea(), moveFromCcr(), moveToCcr(), movem(), movep(), moveq(), mulsWord(), mulsLong(), muluWord(), muluLong(), nbcd(), neg(), negx(), nop() and not().

Team L/E: or(), ori(), oriCcr(), pea(), rol_rReg(), rol_rMem(), roxl_rReg(), roxl_rMem(), rts(), sbcd(), sxx(), sub(), suba(), subi(), subq(), subx(), swap(), tas(), trap(), trapv(), tst() and unlk().

Instruction Translation

Likewise for the instruction translation:

Team A/G: tS_abcd(), tS_add(), tS_addi(), tS_addq(), tS_addx(), tS_and(), tS_andi(), tS_andiCcr(), tS_asl_rReg(), tS_asl_rMem(), tS_bxx(), tS_bchgStat(), tS_bchgDyn(), tS_bclrStat(), tS_bclrDyn(), tS_bkpt(), tS_bra(), tS_bsetStat(), tS_bsetDyn(), tS_bsr(), tS_btstStat(), tS_btstDyn(), tS_chk(), tS_clr(), tS_cmp(), tS_cmpa(), tS_cmpi(), tS_cmpm() and tS_bits().

Team D/A: tS_dbxx(), tS_divsWord(), tS_divsLong(), tS_divuWord(), tS_divuLong(), tS_eor(), tS_eori(), tS_eoriCcr(), tS_ext(), tS_ext(), tS_illegal(), tS_jump(), tS_jsr(), tS_lea(), tS_linkWord(), tS_linkLong(), tS_lsl_rReg(), tS_lsl_rMem(), tS_move(), tS_movea(), tS_moveFromCcr(), tS_moveToCcr(), tS_movem(), tS_movep(), tS_moveq(), tS_mulsWord(), tS_mulsLong(), tS_muluWord(), tS_muluLong(), tS_nbcd(), tS_neg(), tS_negx(), tS_nop() and tS_not().

Team L/E: tS_or(), tS_ori(), tS_oriCcr(), tS_pea(), tS_rol_rReg(), tS_rol_rMem(), tS_roxl_rReg(), tS_roxl_rMem(), tS_rts(), tS_sbcd(), tS_sxx(), tS_sub(), tS_suba(), tS_subi(), tS_subq(), tS_subx(), tS_swap(), tS_tas(), tS_trap(), tS_trapv(), tS_tst() and tS_unlk().

Note that **Team L/E** were given fewer instructions than the other two teams as they had set up the code framework (phase 0).

1.3.3 Phase 2—Graphical User Interface

This phase was mainly done by two teams. Note that though the number of lines required for the GUI and the number of items listed below is far smaller than those for phase 1, this phase was much less straight forward and it took more time than expected.

Team D/A: set up the basic framework for the GUI. They designed the system stack display. Implemented Open, Reset, Step and Run. Inherited and overloaded QTable, to be able to highlight rows or individual cells.

Team A/G: refined the GUI. Automatic scrolling for the program viewer, window resizing, hex/decimal display in the program viewer. Help stack ‘flashing’. Besides they added “What’s This?” functionality.

Team L/E: implemented Undo, ‘Go!’, and breakpoints.

1.3.4 Phase 3—Context Sensitive Help System

This phase was completed by **Team L/E**. It resulted in the ‘help stack’ of the GUI. Internally, it consisted of the development of a generic stack class, and more important insertion of code in the individual instruction methods (see phase 1) to detect possible erroneous situation and issue warnings; also in the case of actual errors (bus error, address error, etc.) a clear error message was issued explaining why the error occurred. **Team A/G** have also added a number of help stack messages in their development of the display of the system stack.

Chapter 2

Requirements

From the problem statement, discussions with our supervisor, and our own ideas we were able to define the requirements that our product needed to fluffed.

2.1 Functional Requirements

1. The most important requirement that it was to fulfil, was to simulate the Motorola 68008 instruction-set.
2. It should be able to read in and decode S-Records for execution of the program and also to print out the instructions in their correct format as the assembler would have output them.
3. The product should be able to step forwards and backwards through a program.
4. We needed to provide context sensitive help depending on the state of the CPU and what was happening in the program

2.2 Non-functional Requirements

1. We wanted the GUI to be friendly and easy to get to grips with, if it was to succeed as a teaching tool.
2. To further this aim of creating a teaching tool, it was necessary to create a visual representation of the CPU and what was happening in the supplied program. In this way the user would earn a better understanding of what the program was doing and the product would fulfil it's purpose.
3. We wanted the product to be cross-platform, that is to be able to run under various Operating Systems, such as the Windows and UNIX Operating Systems, and also on different architectures.

2.3 User Interaction

User opens a program previously compiled by the RoboDev Development Environment. The instructions are loaded into the virtual memory space of the simulated MC68008, awaiting simulated execution.

The user can then step through the program's execution. Step, goes through the program by instructions, stopping after each one.

After each instruction, execution is suspended and the user can step forward and backwards through the program from that point. The state of the system is updated at each step (i.e. after the execution of each instruction).

When the simulator encounters an error, the user is clearly notified about the cause of the error and the origin of this error (e.g. 'This address error has occurred because you have tried to write a word to odd memory location \$1001. See line xxx.').

Chapter 3

Design

This sections describes the architecture of the application along with a discussion of class and algorithm design.

3.1 Architecture

Our representation of the M68008 microprocessor system adheres quite rigidly to Object Orientated Design principles with each constituent part represented by it's own class. Our choice of QT for GUI development meant that the entire system would be completely Object Orientated. The application was split into two distinct components from the start, representing the Core Emulation Engine and the GUI. This primarily allowed us to begin work on the task of decoding the M68008 instruction set without worrying about GUI functionality.

3.1.1 Core Emulation Engine

Having made this division, we designed the Core Emulation Engine so that it could be easily interrogated by the GUI architecture that we would later implement on top of it. Within the Core Emulation Engine, we needed a central CPU class which would deal with the decoding and emulated execution of instructions. This class would also need to provide translation functionality to present the encoded instructions in human-readable form as well as providing undo and system stack functionality. To complete the Core Emulation Engine, several supporting classes were required. Chief among these was a class to represent the memory attached to the CPU, which would have to provide client classes with access any byte, word or longword in the available memory space and deal with any memory access errors (e.g. address outside memory range). The registers needed a specialised data structure too in order to deal effectively with byte, word and longword operations. A generic stack structure was required to implement the help and undo subsystems and another class was required to save the state of the entire system.

3.1.2 GUI

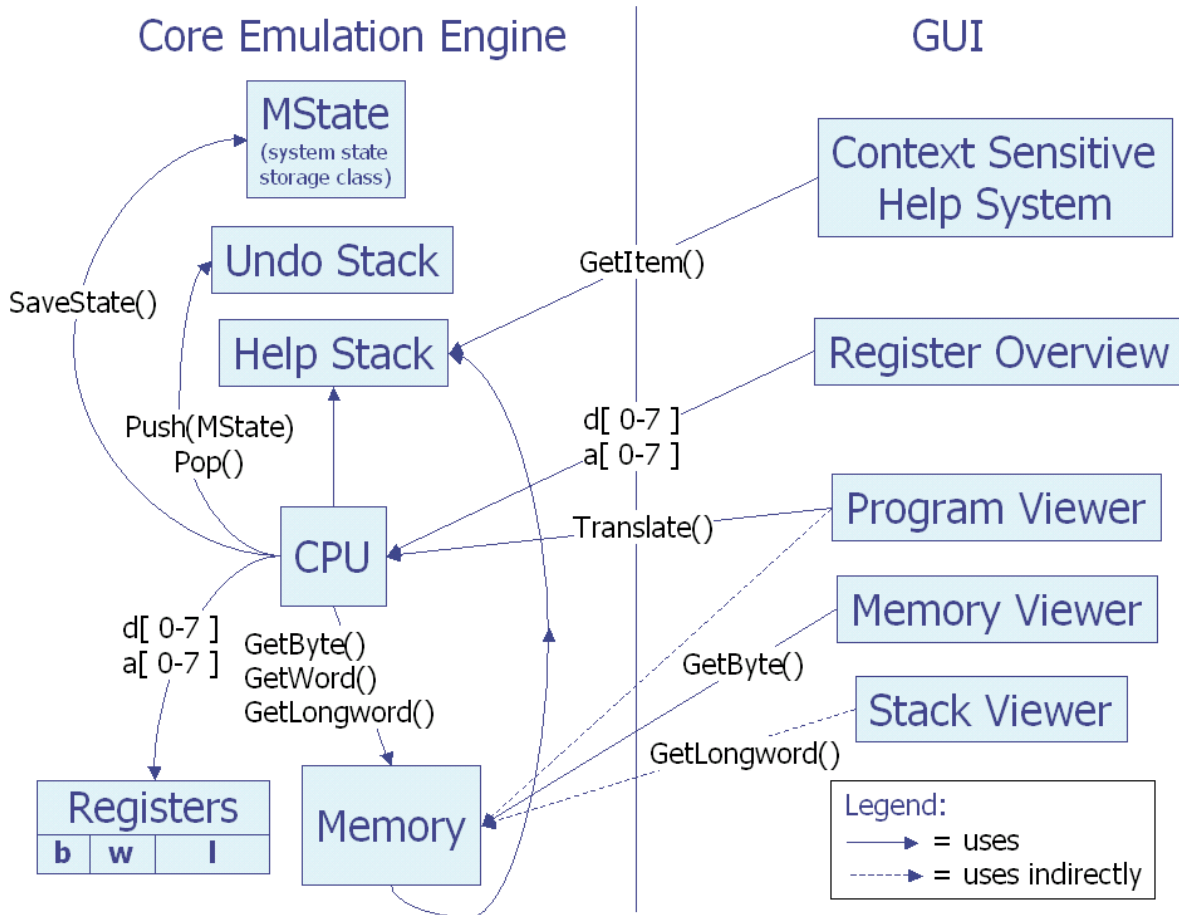
The GUI was required to give an easy to understand representation of the workings of the M68008 and so, as a result, we had to come up with an intuitive interface to represent the state of the Core Emulation Engine at any given time. We decided tables would best represent most parts of the system since we would mostly need to list the values of various parts of the system (e.g. the contents of registers, memory etc...).

As the primary function of the application was to run pre-assembled M68008 programs, the central part of our application was the program viewer. Here we decided we should list the contents of a small block of memory (approx. 100 instructions after the current point of execution), showing the location in memory, the encoded value (or opcode) and the translated 'human-readable' assembler syntax of the current instruction. The other parts of the system had a direct mapping to a GUI component, with the help system, registers, memory and stack viewers having their own table. Our job in implementing the tables was made somewhat easier by QT, which provides a

standard table widget. However, we needed to be able to highlight the row in the program viewer that was about to be executed and any row containing a breakpoint. This functionality we had to implement ourselves.

In addition to showing the hex and binary values of the Status Register, we also thought it would be useful to show their status more visually, so we decided to implement an LED widget that could show the current state of each status bit and would allow the state to be toggled when clicked.

3.2 Algorithm and Class Design



3.2.1 Class Design

CPU

The CPU class forms the central part of the Core Emulation Engine, providing instruction decoding, execution and translation functionality. The instruction engine was completely encapsulated in this class to maximise ease of integration with the GUI. As in the real world, the CPU and memory were implemented separately, mostly to provide a somewhat modular system which could include other devices in the future.

Memory

The primary function of the memory class was to provide the CPU with data to read and execute. The memory also had to deal with parsing S-Records from a source file. Since the M68008 can access bytes,

words and longwords from memory, functionality to implement each of these types of memory access was necessary. In addition, the memory access routines needed to check for any access errors (address and bus errors) and update the help stack accordingly.

Two virtual memories objects are required in order to implement state saving, and hence undo functionality, correctly. One memory holds the current state of the system, while the other holds the previous state. Comparing the two determines the memory to save to the undo stack.

Registers

Not strictly a class in itself, the registers were implemented in a C-style struct to allow byte, word and longword operations to be emulated correctly and efficiently.

MState - Saved system state

MState stores the changes in the emulated system's state between the execution of each instruction using as efficient a storage mechanism as possible. It contains the contents of the CPU registers and status registers, whether the CPU was halted or not and the changes to memory and the system stack. To avoid using excessive amounts of memory the amount of undo information had to be limited in some way. This was done by limiting the number of changes to memory and the stack. Memory changes were limited to 20 longwords (movem alters the most amount of memory, 16 longwords), and while this meant that the effect of any instruction could be undone, excessive changes made in the memory viewer would not be undone. Such a trade-off was unfortunately necessary to avoid excessive memory usage, while providing the best possible undo functionality.

Generic Stack

In addition to the system stack, we also needed to implement an undo and help stack. In order to avoid having classes for each of these that would essentially all be doing the same thing, we implemented a generic stack template class that could be used to represent any kind of stack. This meant that we had only to modify one class definition (rather than three) to modify the way all our stacks functioned.

3.2.2 Algorithm Design

Instruction Decoding, Execution and Translation

Implementing and testing the decoding, execution and translation engines formed the largest part of the project, forming approximately 80% of the code. This was most time consuming part of the project too, and consisted mostly referring to the Motorola Reference Manual, checking what part of the system is affected when this addressing mode is used and then writing code to change the state of the system accordingly.

When executing or translating instructions, decoding first takes place where the instruction is identified by its unique signature (this is effectively the non-variable of the instruction). Once the instruction has been determined by inspecting the non-variable parts, the parts that do vary (i.e. the operands, type of operation (byte, word, longword) and addressing mode) are inspected and appropriate action taken. This might be to load one register with the contents of another in the case of a move instruction, or add one register to the contents of a particular memory location. Each possibility (and there are more than 50 for some instructions) must be checked for and appropriate action taken. As well as updating the operands, the program counter needed updating, both with the size of the instruction (always a word) and the size of any immediate data following the instruction (variable). The CCR posed its own problems with calculations of conditions like overflow being particularly nasty.

Instruction translation was somewhat easier as the program counter and CCR did not need to be updated. In addition, the code used for executing the instructions could be used with the functionality stripped out, since the execution engine checks for the same conditions as the translation engine. Translation was implemented by writing a string containing the assembler syntax to a string buffer and returning it and the address of the next instruction (rather than updating the program counter).

Step

Step simply saves the current state of the system, invokes the decoding and execution engine and then pushes the current state to the undo stack.

3.2.3 RestoreState (Undo)

RestoreState pops the last MState from the undo stack and updates the system state with the state stored in the MState popped.

Chapter 4

Implementation

This sections discusses the tools used in the project's development and the implementation of the GUI and algorithms used.

4.1 Tools Used

Several tools were used to aid the project's development, both for developing the application code and for managing input from team members.

Due mostly to familiarity, we decided to write the application in C++. Our development environment consisted of a mixture of Microsoft's Visual Studio and VIM/GCC, again due to familiarity and also the ability to test across multiple target platforms. In addition, we chose Trolltech's QT libraries for GUI development and CVS (Concurrent Versions System) was used to manage development code. We used a centralised mailing list to share ideas and record decisions made. Programming tasks were distributed according to Pair Programming methods.

4.1.1 QT GUI Libraries

We based our decision to use QT on the grounds of portability (QT runs on a variety of platforms) and usability. Despite familiarity with Microsoft's MFC, neither of our requirements for portability or usability could be fulfilled by MFC. While it is possible to compile an MFC application under non-Microsoft platforms, it is far from straightforward. QT libraries on the other hand have been tried and tested across most modern, mainstream operating systems.

QT's signal/slot mechanism for event driven programming is a case in point of the usability benefits of QT. Whereas MFC requires a somewhat limited message map declaration and call-back function mechanism, QT allows widgets (any GUI component e.g. buttons, menus, dialogs) to emit signals which are then connected to functions called slots. The slot functions execute in response to a signal from the widget. Multiple signals can be connected to one slot and multiple slots (can be connected) to one signal. It is also very easy to define custom signals which can be generated in response to existing signals or some custom widget function. Signals and slots were of particular use in updating the GUI , allowing us to create one signal which updated the contents of each table.

QT also sticks rigidly to object orientation and this was particularly of use when customising widgets. This allowed us to easily extend QTable (QT's basic table widget) to provide row highlighting and shading.

4.1.2 CVS Code Management

Since we could have potentially had up to six different versions of the project code at any one time, we decided to use a version tracking system to keep track of changes to the code and allow us to undo any changes that would cause the application not to function correctly. In addition it proved to be an easy way to keep track of each group's contribution to the project and to monitor the progress of the project as a whole.

The ability to undo the effects of a particular revision of the code was used successfully on two occasions when erroneous code was committed. Attempting to do this manually would have taken many frustrating hours, if not days, to complete, so we were quite pleased to be able to roll back the effects in a matter of minutes.

CVS itself was quite easy to use and for each team member/group, involved updating the working version of the code with the version on the CVS server, making changes and then committing the changes back to the CVS server. In conjunction with Horde's Chora (a web interface for CVS) it proved to be a very useful tool.

4.1.3 Mailman Mailing List

Our main motivation for using a mailing list was the ability to keep an archive of past group discussions, ideas and progress. It made sending group-wide e-mail somewhat easier, with just one e-mail address to remember (rather than 5), and meant all members could be sure of receiving e-mail sent to the group.

4.1.4 Pair Programming

We decided to work on code in pairs, both to improve the application's reliability and to make coding somewhat more interesting. Pair programming worked well in most cases, however it sometimes failed as a result of conflicting schedules. Working in pairs was also useful for sharing ideas and programming techniques and many errors were prevented while coding and other errors more quickly resolved by it's use.

4.2 User Interface Implementation

4.2.1 QMTable

As discussed already, the majority of the application's GUI consists of tables. In order to implement the program viewer correctly, we needed to re-implement the default QT table widgets to provide row highlighting functionality. To improve the appearance of the table, we decided to shade alternate rows. Implementing row highlighting mostly involved re-implementing the cell painting methods for the standard QTable widget so that the cell background was painted with an appropriate colour.

Each time a table is updated, the contents of each of it's cells are rewritten and the visible cells redrawn. It was necessary to reread all the data for each table on each update since instructions can modify any part of memory and so any part of the table could in theory be modified. Updating the program viewer primarily concerns invoking the CPU's translate method for the number of rows in the table. This gives the assembler syntax and address of each instruction, and the opcode is simply obtained by repeatedly reading bytes from the memory until the size of the opcode is reached. The memory viewer is updated by reading a byte from a starting address until all the cells in the table are filled. The stack table simple interrogates the system stack object's contents.

4.2.2 Opcode template viewer

The opcode template viewer was implemented using QT's default label widget. QT allows the background of any widget to be set to a number of graphics file formats, so we simply changed the bitmap files (saved as PNGs) to the correct one for the instruction about to be executed. It just remained to modify the decoding engine to save the name of the instruction that was about to be executed and naming our files according to this. The opcode template is updated by the program viewer update method due to compatibility issues.

4.2.3 CCR flags

As mentioned previously, we decided to implement the CCR flags as clickable LEDs for increased usability. When the LED button is clicked, the value of the associated status flag is toggled using bitwise inversion. Each LED button emits an `updateTables` signal when the button has been clicked to update the colour of the LED.

The `registerUpdate` slot responds to the `updateTables` signal, and is responsible for setting the button to the appropriate color. This is achieved by checking the value of the associated status flag and setting the background colour of the button accordingly (this is implemented as standard in QT).

4.2.4 Updating the GUI

The current state of the GUI can be update at any time by calling `emit updateTables()`. This emits the `updateTables` signal which is connected to each of the update routines of each GUI component discussed above.

4.3 Implementation of Algorithms

Instruction Decoding

Decoding the instruction begins with the `Link` method which identifies the instruction according to its signature. This is done using a macro which is supplied with the method to call the bit mask corresponding to the instruction signature and the value of the signature. The macro then ANDs the mask with the instruction register and compares it to the instruction signature, invoking the subroutine which will perform the execution if necessary. The decoding engine is effectively repeated with some modifications for translation with values such as storing a string containing the last instruction executed (this is used by the opcode template image to ensure the image is only updated for new instructions).

4.3.1 Execution and Translation

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----------|----|---|--------|---|---|-------------------|---|---|----------|---|---|
| 1 | 1 | 0 | 1 | REGISTER | | | OPMODE | | | EFFECTIVE ADDRESS | | | | | |
| | | | | | | | | | | MODE | | | REGISTER | | |

The exe-

cutation and translation engines expand upon the functionality of the initial decoding by inspecting the parts of the instruction opcode containing the operands and addressing mode for the instruction. Compiler macros are used again to get the value of particular bits in the operand. The value of each part of the instruction is first extracted from the opcode using the compiler macros (i.e. register, opcode, effective address mode and register in the above instruction template). The remainder of the method is then essentially a large switch or if construct which checks first of all for the mode in use (opcode) - this indicates the size of the operation (byte, word or longword) and the order of the operands (effective address as the source or destination). The mode field then indicates what the operands are. One (usually the destination) is of fixed type for the instruction (usually a data register, but can be an address register for `adda`, `movea` etc...). The other will indicate the effective address (a register, indirect addressing mode (e.g. `a0`) or immediate data). Knowing this information, the relevant calculation can then take place and the PC and CCR are updated.

Chapter 5

Conclusions

This section details the conclusions that were made with regard to our objectives, any problems that we faced, how well we functioned as a team and future possibilities for the product.

5.1 Design and Implementation Problems

For the most part we had very few serious problems, however there were one or two.

The first problem turned up when we realised that there were some problems with the Motorola manual that were using. This caused some of our code to be redundant and this was fixed.

The biggest problem that we had was implementing the system stack satisfactorily. We were having numerous problems with it which required that the pop and push functionality be completely rewritten towards the end of the development.

5.2 Completion of Project Objectives

5.2.1 Supervisor's Requirements

Overall, we came very close to fulfilling all of the requirements specified by our supervisor. We have produced a program that could be used as a teaching tool with the Introduction to Computing (1BA3) module in mind. The only shortcoming of our end product is that we have not fully implemented all of the instructions.

5.2.2 Additional Objectives

As regards our other objectives, we fulfilled all of them that we knew we could get done in the time provided. Our code is completely cross-platform and has been tested on various UNIX and Windows operating systems, as well as on two processor architectures, Sun SPARC and IBM compatible PCs. This does not mean that more work could not be done, however all in all, all the feasible and planned objectives that we had for the project were completed.

5.3 How we worked as a group

As a group we functioned quite well. We stuck more or less to our own assigned deadlines and indeed had the core development of the project finished well before the official deadline. This provided time to test the product.

Communication within the group worked well most of the time with a few minor failures.

All in all, the team was much like any other team, people assumed their roles and for the most carried out their responsibilities to advance the group along the development.

5.4 Future Enhancements

5.4.1 Solve some remaining issues

Despite having a fully functioning Motorola 68008 emulator, some issues remain with the existing code:

1. **Stack issues:** The operation of the system stack is not entirely bug free and does not work in every cases. It does however function in most common cases.
2. **Extend undo functionality:** Allow *all* changes to be saved and not a limited subset, or increase the subset changes that will be saved. If undo is pressed, it should undo all changes and not just a subset.
3. **Program Viewer scrolling:** The automatic scrolling mechanism of the program viewer is not completely effective and sometimes fails after opening files.
4. **Application window:** The application runs in a dialog window and not a main window. This excludes functionality such as a button bar, menu bar and status bar. This can easily be modified and would greatly improve the program aesthetically.
5. **Instruction decoding engine:** The instruction decoding engine can currently correctly decode and execute approximately 90% of the Motorola 68008 instruction set. This includes all of the most commonly used instructions, however some instructions have been omitted in whole or in part.
6. **Undoing the help stack:** Errors pushed to the help stack are currently not undone. It is therefore somewhat confusing when the an instruction which caused an error is undone and the error remains. Since the help system is implemented as a stack, integration of the stack and undo systems would require recording the number of errors generated in each saved state and then popping this number of items from the help stack.

5.4.2 Make the tool useful for 2BA4 students

The application currently only supports user instructions and functions such as privileged instructions and interrupts are not implemented. These are both fundamental aspects of the software design part of 2BA4 (Microprocessor Design), the follow on course to 1BA3, and so the application is not particularly useful for 2BA4 students. Proper memory mapping would also need to be implemented to allow features such as serial IO and LCDs to be included and access to the PIT and interrupt handling would also be useful.

Given the current architecture of the application it would be relatively easy to add more instruction decoding/translating functionality. The Step() and Translate() methods would need updating and methods for each instruction would need to be implemented. Once proper memory mapping is implemented, adding more devices would be quite straightforward due to the modularisation that memory mapping would provide. A basic form of IO is already implemented, and extending this would mostly involve developing and integrating a GUI widget. Emulating an LCD device would again be mostly GUI related, and would probably be best implemented as an extension or re-implementation of the output functionality in a serial IO system. Implementation of interrupts would be somewhat more challenging and would probably have to be written from scratch.

5.4.3 Provide assembling functions

Currently to rectify errors in assembled code being emulated by the application, the user must modify the original source file elsewhere, reassemble the program and load it into the emulator again. It would be extremely useful, therefore if the user could modify the program directly in the program viewer and have the program reassemble the changes made. There are a number of ways of achieving this. Simplest among them would be to include a compiler with the program that the compiler could execute automatically and then reopen the output from the compiler. However, this solution is far from ideal and would require either the entire memory space (1MB) to be recompiled each time just one line was changed or that the bounds of the program in memory were known.

A more suitable solution would be to have the ability to recompile each instruction in the program viewer as it is changed. This could be achieved by using some pre-existing libraries or by using the knowledge gained in completing the project to extend the execution/translation engine to include an assembling engine.