# Generator

## A Sega Genesis emulator

*Executing Sega Genesis cartridges on foreign systems*

3rd year project report 1997-1998

Written by
James Ponder

Supervised by
Sara Kalvala

# Contents

# 1 - Introduction

## *1.1 The Sega Genesis*

The Sega Genesis, which is named The MegaDrive in the UK, was produced in 1990 and was the sequel to Sega's very popular Master System. Previous systems, such as the Master System, were 8-bit and had very rudimentary graphics and sound - this new 16-bit system had advanced FM[i] stereo sound, much improved graphics and a faster processor allowing for more dynamic games.

The system was purchased by consumers throughout the world and had a huge following, much like the Sony PlayStation has today. The system plugs into the television and was hence easily accessible to the home user. The games for systems such as these are purchased separately and when the Sega Genesis ended its commercial life there were many hundreds of games available.

## *1.2 Emulation*

Emulation is providing the ability to run a piece of software on a system that was not designed for the software. This project is concerned with allowing a user to play the games that were originally provided on cartridges for the Genesis on their modern computer system.

Why would we want to emulate the Sega Genesis when there are new and far more advanced systems available?

**Historical**
Many people feel that we should preserve the games for the historical benefit of generations to come. Just as we keep records of all television programs and films by transferring them onto modern materials, it is possible to play these games through emulation on computers of the present and future. The whole computer era of this century will be examined by historians of the future, and through the ability to play the games that children experienced on a daily basis there will be a better understanding of the culture that we have in the 90s today.

**Nostalgia**
Millions of people played the games that are available on the Sega Genesis. The demand is already present to enable people to be able to play their favourite old games on hardware that they have access to now. Indeed, so much so that a whole cult has formed around the process of emulation. From newsgroups to web pages, there is information on the subject for everyone[1]. The Genesis is not an exception and there are many pages of information, not only on how to

[i] Frequency Modulation - a form of sound modulation used to generate tones and hence music.

write games for the now obsolete system[2], but also text on the inner workings and how it operates[3]. Quite a substantial part of the Research chapter is based on this information.

**Practical**
The original hardware of the Genesis is unlikely to fail mechanically for some years to come, but it is nonetheless a fact that eventually units will start to wear out and stop working. Many of the systems sold will be used be small children and will be damaged easily and others will be thrown out, considered worthless. The cartridges themselves are of more concern as most will contain EPROMs which have a much shorter lifetime than proper ROMs. Rather than to try and maintain some working machines it would seem far more practical to emulate the software. It is also true to say that all games ever produced can easily be stored on a few ZIP disks (100MB floppies) whereas the cartridges for those games would take a few bookshelves.

## *1.3 Existing emulators*

Emulators are not a new concept in themselves. Virtually all machines, whether for work or recreational activities, have been emulated in some fashion by someone, somewhere. One of the earliest examples of emulation was for commercial reasons by non-PC vendors. In order to attract people who needed a PC-compatible at work but didn't want to stick with a cumbersome architecture, suppliers from the late 1980s provided software emulations of a PC running DOS. This allowed people to run applications such as Word 5 but let them stay in the native environment of their chosen system (e.g. Acorn or Apple machines).

The Sega Genesis had already been emulated when this project was started. The emulation available was solely for PC-compatibles and was written in assembler, a non-portable language. At the close of this project there were an additional two emulators available - still solely for PCs. All three versions had shortcomings in their emulation that I believed could be overcome with a properly managed project.

## *1.4 Project Aims*

The aim is to design and implement an emulation of the Sega Genesis in software. There were several key design goals.

**Platform independence**
A major problem of the current emulators is that they are not platform independent. They will only emulate a Sega Genesis if you run the emulator on a PC-compatible computer. This means that they will not run on any non-PC computer, e.g. UNIX, Apple or Acorn machines, or indeed any machine that might be designed in the future.

**Emulation accuracy**
There is no point writing an inaccurate emulator. If what you see on the screen does not match what a user on a real Genesis would see then it is inaccurate. This would obviously not be good for historic reasons or nostalgic - the users would complain. It is important to remember that no official information from Sega is available to emulator authors, and as such, it is almost never possible to emulate with absolute precision. It is however important to ensure that the emulation is not needlessly inaccurate.

**Fast emulation**
The system we are emulating is a games machine. It is hence necessary to put a significant amount of effort into the speed of emulation. Although it is always good to have efficient code, it was known from the outset that it would be necessary to explore new and advanced techniques in the quest for speed.

## *1.5 Report outline*

This project concentrates on the 68000 processor at the heart of the system that runs the programs originally found on cartridges. The reason for this is that it offers the most scope for improvement and is where an emulator's heart is.

The following three chapters form the bulk of the research involved in this project. The first of these chapters, chapter 2, presents an overview of the system as a whole. This should help form an understanding of what the Genesis is all about and lay the foundation to the whole project.

Chapter 3 has a more detailed look at the part of the Genesis that is responsible for the graphics output including the Video Display Processor.

Lastly chapter 4 looks at the other major part of the Genesis, the main processor, and how it can be emulated.

Unfortunately to really appreciate this project would take a report 5 times the size of this.

# 2 - The Sega Genesis

Writing emulators has always been difficult because the companies who build and design the systems do not usually give out information on how they are constructed and how you would emulate them. Sega are no exception and have not released any information on the internal workings of the Genesis - all information here was found in public domain documents available on the Internet ***.

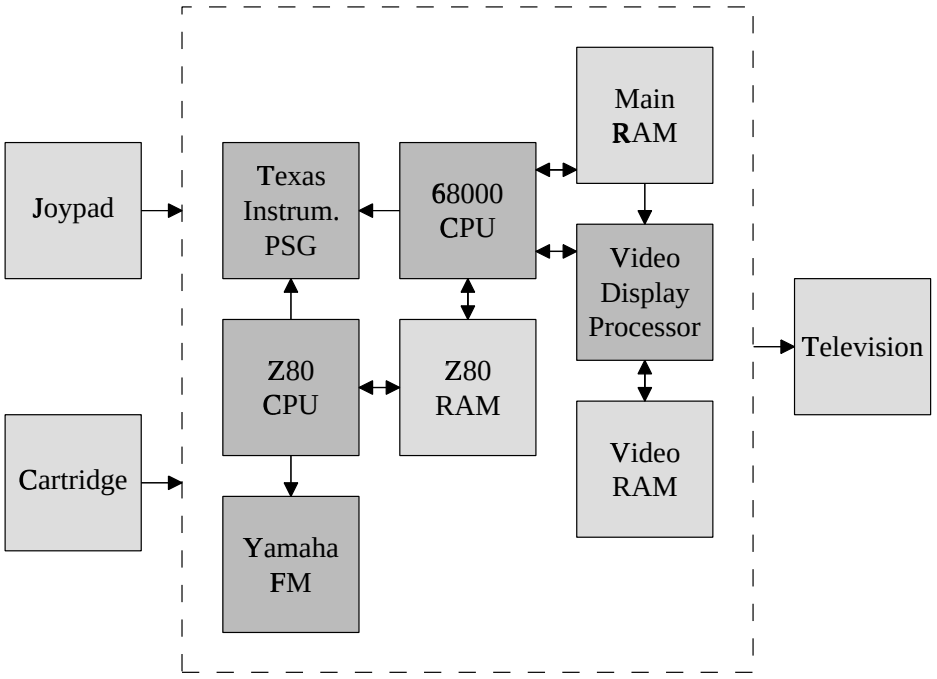


Figure 2.1 - Block diagram

## *2.1 User operation*

The Sega Genesis connects directly into the user's television set. In order to play a game a cartridge must then be connected to the system. The cartridge is internally a ROM of data and there are little or no electronics apart from this ROM inside. The cartridge has many pins at the bottom of it that connect to the main system and these are data and address lines that connect directly into the main buses of the system. The cartridge becomes a part of memory - from memory location 0 upwards.

The game is played through the use of a joypad. This has 3 buttons marked A, B and C plus a start button used for starting the game or pausing the game once in progress. The joypad also has an 8-way directional thumb-pad which is usually used to control your character.

## *2.2 Central Processor*

As with all computer systems there is a processor at the heart of the Genesis. The CPU chosen by Sega was the 16-bit 68000 Motorola microprocessor running at 8Mhz. This was presumably for cost reasons as this chip, and others of the family, were used in many computers of that time. Even today the 68000 processor is still popular.

The 68000 can address up to 24bits of data, meaning that the Genesis has an address range of 16 Megabytes. The Genesis communicates to the outside world through the use of memory mapped I/O. By writing to a location in memory an external device can read what data is being sent and perform actions. In the same way, by reading from a location in memory an external device can present data for the processor to read, and hence provide input to the processor.

The 68000 processor treats the data in locations 0 to 255 specially as it usually belongs to the operating system. These are special locations that tell the processor where to branch to when an interrupt occurs, when certain exceptions happen and where to start when the power is first turned on. Since the game on the cartridge includes the data for locations 0 to 255 the game itself performs all the functions of an operating system. The game (at manufacture time) can choose this data and as we shall see later the locations for interrupts are very important. It also makes emulating this system easier than some others as there is no copyright Operating System required by us.

## 2.3 68000 Memory layout

The memory map of the 68000 in the Genesis is presented below:

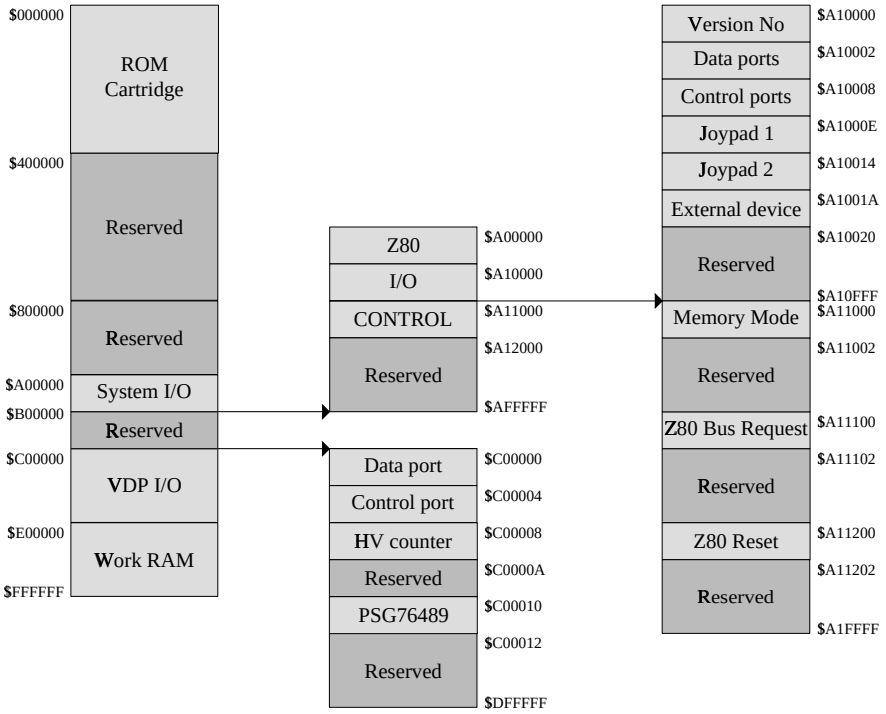


Figure 2.2 - Genesis memory map

The ROM and RAM are banks of memory in the usual way, however, the System I/O and VDP memory areas are memory mapped methods of communicating with different parts of the Genesis system. Due to the complexities involved the precise method of communication and syntax is not presented here but can be found in one of the Internet Genesis documents[3].

## 2.4 Z80 Secondary Processor

The 68000 processor performs most of the work in making the game function, however, music and sound are performed in the background by a Z80 processor. The Z80 processor can be setup and managed from the 68000 through the memory mapped area shown above. The Z80 works concurrently with the 68000 system and has its own memory area. Once the 68000 has presented it with a program to run it operates concurrently without interference. This method of background processing is advantageous in that the sound and music do not consume valuable processor time or cause timing problems. The two devices that the Z80 can communicate with are:

### Texas Instruments PSG Processor

This processor is a low-capability programmable sound generator. It can produce simple tones and white noise. It is not used to produce music and most games do not use it for much.

### Yamaha FM Sound Processor

This processor is a high-capability frequency modulation device that produces a particular range of tones and voices that are suitable for creating music from. The style of music this produces is very distinctive and popular.

### Z80 and Sound/Music

It was decided that the sound part of the emulation fell outside the scope of this project and the emulation of it was not attempted. It was felt that the project was already on the verge of being too complicated without adding FM synthesis, envelope generation and Z80 synthesis to the list of functions to implement.

## *2.5 Video Display Processor*

This processor is responsible for generating the signal that is output to a television screen and is quite a complicated part of the system. Up until recently memory has been very expensive and it was difficult to have a large chunk of memory devoted to representing the screen, as is common on today's desktop computers. It was also difficult for a game to update the entire screen to reflect the game-play between each frame - as we shall see the processor only has a few thousand instructions between each frame to do any updating.

The designers chose to have an intelligent processor that could create the screen from a series of layers and priorities that could be manipulated by changing only a few bytes at a time. An 8Mhz processor should has as much time available for the real game-play as possible and as such putting as much effort onto the VDP as possible was probably seen as a good idea. Today we see this in the production of 3d graphics cards for high-end workstations.

## *2.6 Summary*

Although a mysterious black box to the user, the Genesis is, on the surface at least, quite a straightforward set of components.

A quick look at the Genesis' system design should have given you the feel that the project is clearly divided into two sections.

- 68000 processor, a known CPU - how do we emulate/simulate it?
- Video Display processor, an unknown - what is it and is it easy to perform its role?

In some respects a CPU is a CPU, its function is not an unknown quantity and does not require further immediate research. On the other hand it was felt that the Video Display processor needed a lot of research - not only to find out what work lay ahead but also so that there was a deeper understanding of the system as a whole.

# Video Display

## *3.1 Television output*

The Sega system is designed to be used in countries that have either the PAL and NTSC standards. The software can find out from the system whether the unit is being used in a PAL or NTSC country and hence alter the output mode. The Genesis supports a 224 pixel high mode for both NTSC and PAL and a 240 pixel high mode for only PAL. To simplify the software almost all games use just the 224 pixel high mode - when this is used on a PAL TV the Genesis centers the picture, leaving a small blank portion similar to that found on wide-screen movies.

All timing in the software is triggered in response to interrupts. These are ultimately generated by the PAL/NTSC standard. There are Vertical Interrupts, once per screen refresh at the end, and Horizontal Interrupts, once per line at the start.

A television electron beam cannot move from the end of one line to the start of the next instantaneously and this delay is called the Horizontal Blanking period. Likewise there is a Vertical Blanking period between each frame. The diagram below should help to picture this:

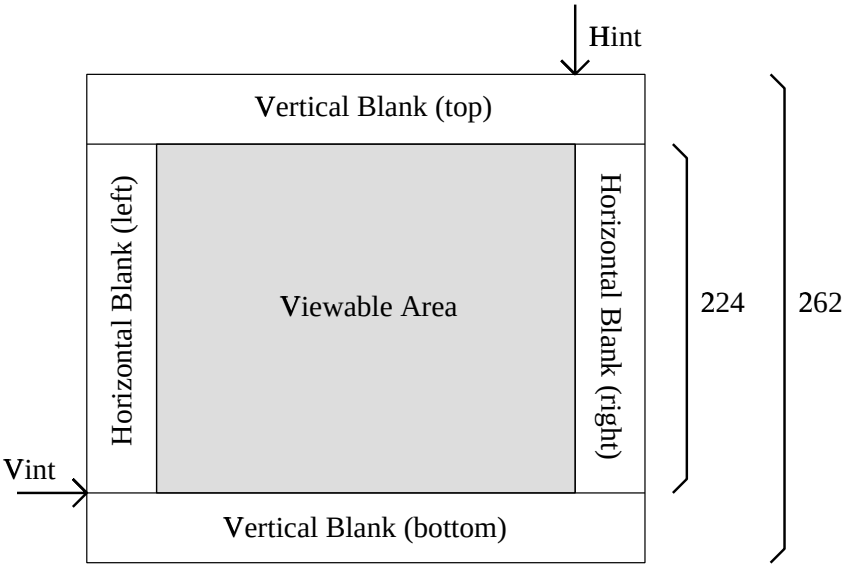


Figure 3.1 - Television frame

Most games do not need to use the Horizontal Interrupts as it is sufficient to update the video information once per frame, however, they can be used to perform interesting raster effects such as palette switching half way down the screen.

The table below shows both the PAL and NTSC timing information for non-interlaced operation (the normal mode for the Genesis).

| | Total lines | Frequency | μs/frame | μs/line |
|---|---|---|---|---|
| NTSC | 262 | 30Hz | 33333 | 127.2 |
| PAL | 312 | 25Hz | 40000 | 128.2 |

We can use this information, and the fact that the CPU runs at 8Mhz to calculate how many clock cycles we should emulate per line and per frame:

| | Clks/frame | Clks/line | V blank clks |
|---|---|---|---|
| NTSC | 266666 | 1018 | 38634 |
| PAL | 320000 | 1026 | 90176 |

The game is told through interrupts and a memory mapped location what the current line that the display hardware is currently outputting. When in NTSC mode, after the 224 visible lines are drawn a Vertical Interrupt occurs and the game then has 38634 CPU clocks to update the screen for the next frame. We must ensure that we emulate the CPU as accurately as possible, performing the correct number of instructions corresponding to 38634 clocks.

## *3.2 Video processing*

As already mentioned in chapter 2, the video processor in the Genesis creates the output frame through a series of graphical planes. The VDP provides for a background colour, two scrolling planes of graphics and independent sprites. One of the scrolling planes also has a 'window' facility which allows for a different scrolling plane to be used in a particular area of the screen.

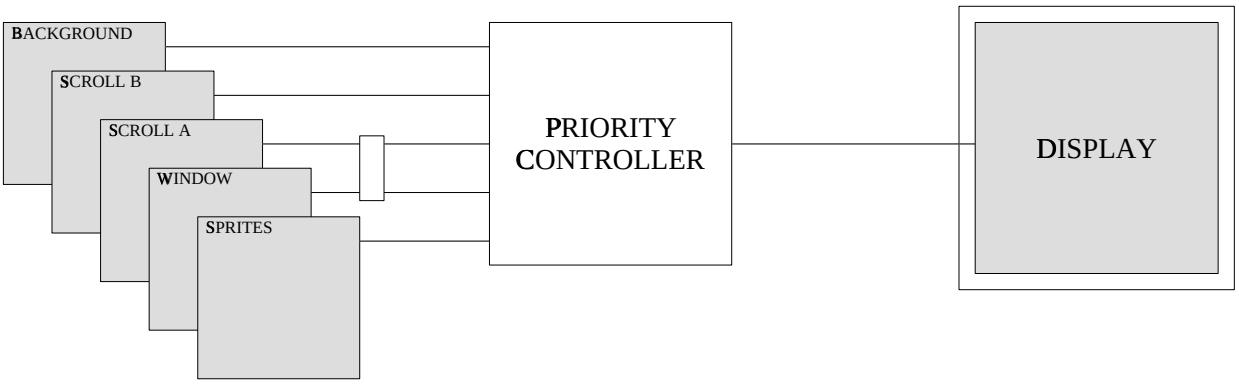


Figure 3.2 - Frame composition

Each layer can have a mask so that the layer below it shows through. If all layers are masked in a particular place then the background layer will show through. This is just a single colour that

the game can choose. If the game decides to alter this colour, perhaps to indicate a different mood to the game, it can do so with just a few instructions. If you compare this to altering an entire bank of memory-mapped video ram then you can see this method has a lot of potential, and is why the Genesis can get away with a processor that only goes at 8Mhz.

The two SCROLL planes, A and B, are managed through a two-dimensional array of cells. Each element of the array contains a sprite number, and the sprite data is held in a different part of the memory. Hence, by changing one byte of data in the array, a part of the screen can change - in this case each cell is 8 by 8 pixels. This reduces the work load of the processor considerably and allows for quite complicated changes each frame without any shifting of data.

Each scroll plane is bigger than the visible screen area and the video processor allows it to be shifted about using an X and Y offset. The game can alter these offsets and move the planes independently of each other. In a platform game, where the player moves a character along a landscape moving from right to left, by changing one value the game can shift the entire landscape and not even need to update the newly revealed area (as in hardware scrolling) as the scroll areas are many times larger than the screen is.

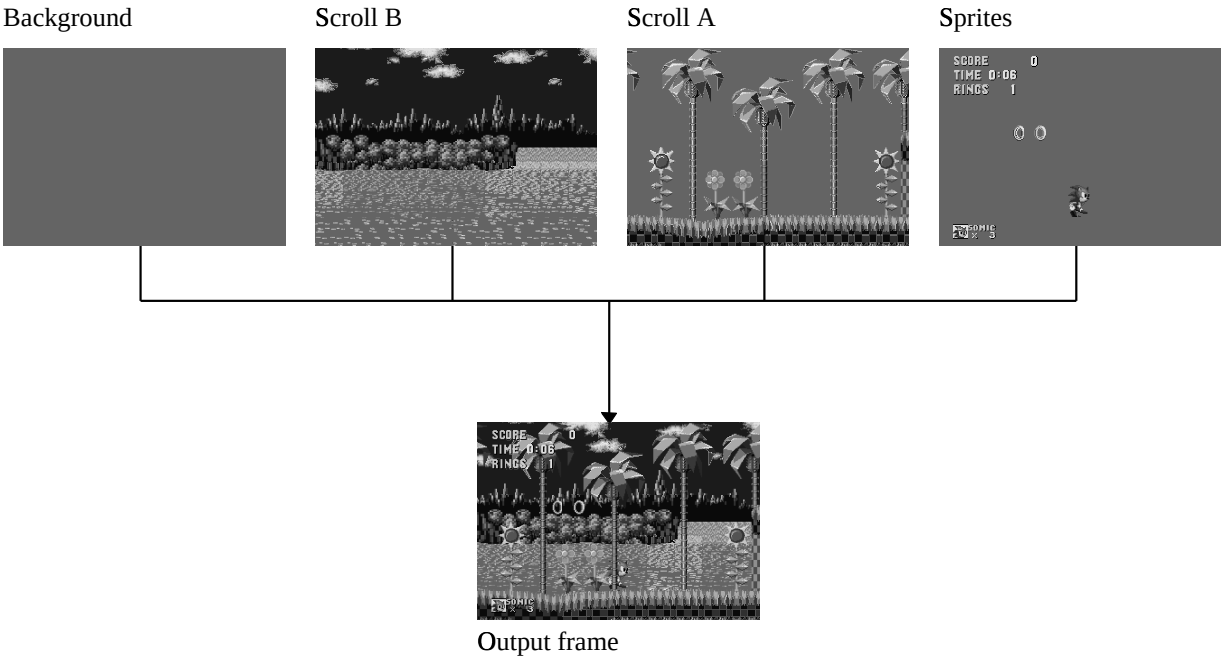


Figure 3.3 - Example frame composition

The screen shots above should clarify what is happening. The Scroll B frame in the game above is being used as a parallax movement playfield. This will be moved from right to left at a slower rate than Scroll A in order to create a 3d-effect. Scroll A moves from right to left in order to produce movement for the character, in this case Sonic The Hedgehog, who stays in the center of the screen. Sonic himself is plotted in the Sprites layer which allows the game to have variably sized images placed on the screen. The sprites are again made up of cells, but instead

of filling the entire screen a sprite is an array of just a few cells, e.g. 2x4.

## *3.3 Priorities*

Each cell on each layer, and the cell-components of the sprite, can have a priority, 0 or 1, attached to them. If you look at figure 3.3 you will notice that the Sonic character appears behind a tree that is in Scroll A. This is explained because the tree is at priority 1, meaning that it goes above anything that is priority 0, such as Sonic. Figure 3.4 shows everything in Scroll A that is at priority 1.

Scroll A - priority 1

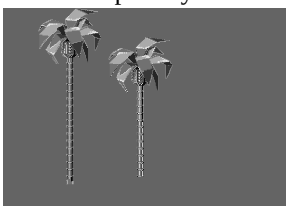

Figure 3.4 - Frame using priorities

You will notice that the tree does not go behind the score information, this is not surprising when you learn that the score information is at priority 1 too. There is an order of precedence when plotting:

1 Background priority 0
2 Scroll B priority 0
3 Scroll A priority 0
4 Sprites priority 0
5 Scroll B priority 1
6 Scroll A priority 1
7 Sprites priority 1

It should amaze you to see how many effects can be produced by setting priorities. This means that almost all of the sonic game-play is performed in hardware and allows for the game to do much more important activities during the Vertical Blank period between each frame, for example collision detection and animation.

## 3.4 Colours

Each cell in the scroll layers and the sprites can have a palette number, 0-3. This indicates which palette to use. These 4 palettes are stored in a special area of memory called the Colour RAM (CRAM), located inside the VDP. There are 16 entries in each palette definition and each entry can indicate a colour composition of Red, Green and Blue, 3 bits each. This allows the Genesis to have the possibility of 512 colours, however, you will note that there are only 4 palettes of 16 colours, hence there is a maximum visible number of 64.

This can be increased by palette switching (changing the palette definition) when a Horizontal Interrupt occurs. This allows the game to have a different set of colours for one half of the screen to the other, for example.

## 3.5 Other features

The VDP also has a Hi-light and Shadow facility. Each cell can be set to either of these and the palette used for that cell will either be brighter or darker. The Hi-light is accomplished by the correct colour values being halved and added to a half-intensity white, and the Shadow by just halving the colour values to produce a maximum of half-intensity.

The VDP can also flip each cell horizontally and/or vertically. This is typically used to save space in a sprite or large layer-based image.

Although not used by many games the VDP also has the ability to detect collisions in sprites on the sprite layer. This was a very useful function on early consoles but for most games it is too crude to use this as almost always there is a sprite overlapping with another sprite somewhere on the screen. Instead, most games do the detection in software themselves.

## 3.6 Operation

The VDP has electronics designed to output PAL or NTSC information. When it detects that it is about to start a line of a frame it raises the logic level on an output pin and informs the CPU via the Horizontal Interrupt that a line is due for plotting.

The CPU has about 36 clocks before it is taken off the bus and the VDP begins to transfer the data from the RAM areas. The location of where to transfer the information is communicated to the VDP beforehand - scrolling information and colour maps are also known already. This data transfer is to pick up the array information for the layers and also the sprite information and cell data itself. Once this is complete the CPU regains control.

The information that was transferred is used for the generation of the single output line currently required. This allows the program to change the data after each Horizontal Interrupt in order to affect the information produced on the next line - this procedure is however rare as it is normally unnecessary to fiddle in this way, the VDP does most that a game could want already.

## *3.7 Summary*

The Video Display Processor forms a substantial amount of work to plot the frame. This chapter should have enabled the reader to get a feel of what will need to be written in order to emulate the functions of the VDP.

A lot of the features described can be omitted in any initial design if it is just necessary to demonstrate a working system as quickly as possible. For example, the hi-light, shadow and priority of graphics does not have to be correct for some games. If we wish to have an accurate emulation, one of our design goals, it would be better if we simulated as many functions as possible in the time available.

The next chapter addresses the problem of how we could emulate the 68000 processor in the system. The 68000 is the focus for the project but it is important not to underestimate the complexities involved in the other parts.

# 68000 emulation

## *4.1 Motorola 68000 processor*

The 68000 processor is a 16-bit CISC processor that can address up to 24-bits worth of memory. It has 8 data registers and 8 address registers which are all 32-bit. It is quite straightforward and has lots of documentation available that eased the development of this project[4]. The emulation requires that we execute in software the behaviour of a 68000 processor as exactly as we can. As with all processors it is geared around the execution of each instruction at the current Program Counter, in this case a virtual register containing where we are next to execute from. Once an instruction has been executed the Program Counter is updated and the emulation proceeds with the next instruction. The 68000 has variable length instructions - although always a multiple of 2 bytes (16-bits) they vary up to 10 bytes in length. Usually the updated PC will simply be the old PC plus the length of the current instruction but there are circumstances, such as branches, that make the PC move to an arbitrary point in memory.

## *4.1 Types of emulation*

There are three standard ways of performing emulation that are detailed below. In each case the emulator has a loop that will perform the desired action of the instruction or instructions that are currently at the virtual Program Counter.

### Interpretive emulation

Interpretive emulation is the most common method used in emulation. The instruction at the PC will be decoded by looking it up in a table. The table will contain the address of a routine that will perform the desired action of this instruction. Usually each routine can cope with a group of similar instructions, such as ones where the destination CPU register is encoded in the bit pattern of the instruction.

Advantages

- The main advantage over the other two forms of emulation is that it can cope with self-modifying code. This occured quite often in games on console systems like the Genesis because it is the only way that a game can dynamically change itself. All of its own code is in ROM and unchangeable and hence often code is created in the RAM area.
- There are no output structures and hence is easy to debug and maintain.
- It is simple and easy to write.

Disadvantages

- This form of emulation is very slow. No attempt to cache the lookups is made and hence the emulator may decode an instruction repeatedly and unnecessarily.

### Binary to binary translation

Rather than do lookups and interpret the code as you go, this method takes the whole program and attempts to compile it into native code, all at once. This is usually done by converting the object code to a language, such as C, which can then be compiled efficiently by a dedicated compiler.

Advantages

- Produces probably the fastest code possible.
- The code is optimised using current compiler technologies and can hence make the code even better than it originally was when those compiler technologies were not available.

Disadvantages

- Cannot cope with self-modifying code at all. Without putting in an interpretive engine it cannot emulate any code that wasn't originally in the program - in this case any code that wasn't in the cartridge ROM.
- Requires all programs to be compiled before they can be used which can be a time consuming process.
- Data gets converted as well as code as we don't know at compile time what is code and what is data.
- 68000 instructions are variable-length and since we don't know where the start of a routine is we will have to produce output for each 16-bit start address. The output files are going to be very big.

### Dynamic recompilation

This is a mixture of binary to binary translation and interpretive emulation. Execution proceeds as with the interpretive method but after a block of code has been executed enough times to warrant compilation the block is compiled to native code. When this block is encountered again it will be executed directly except when a write to the block means it needs to recompile again.

Advantages

- Almost as fast as binary to binary translation.
- Can cope with modifying code.
- Testing can proceed on the interpreting section before having to switch to the development of the dynamic compiler.

Disadvantages

- Much more complex than the other methods.
- Requires the writing of a compiler and back-end for each target processor.
- Dealing with self-modifying code will cause a performance penalty in order to detect it.

## *4.2 Current 68000 emulators*

Here is a small selection of emulators and the type that they use:

**GenEm**[5] **/ KGen**[6] **/ Genecyst**[7]
These are the 3 existing Genesis emulators for MS DOS machines. All of them are *interpretive* 68000 emulators but manage quite admirable performance on standard Pentium machines due to their construction in assembler.

**Executor8**
This is an Apple Macintosh emulator for unix and MS DOS machines. It is the only *dynamic recompiler* that I know exists and is said to be impressive in terms of its performance on x86 machines. It is commercial and has had lots of development over the years.

**Tibbit9**
This is a timing insensitive *binary-to-binary* engine designed for use in embedded controllers. It was constructed as part of a thesis and performs the conversion by outputting a C source file that will be optimised by a modern compiler to produce very fast code.

## *4.3 Type chosen*

All the emulators listed in the previous section were looked at to see how well they performed and what the benefits of each were. It was apparent that a binary-to-binary engine would not be satisfactory as it requires each game to be pre-converted. This would require either a compiler to be part of the project or that another compiler be used. Either way, it would mean the end to portability which was one of the key goals of this project. The disadvantage that it cannot cope with code created in RAM was also seen as a decisive factor.

Interpretive emulation seemed to be too safe an option, in that it is the slowest. This combined with a portable language, rather than assembler, seemed to be against the requirement of a 'fast emulation' as outlined in the Project Aims.

On the other hand a dynamic recompiler was seen to be almost beyond the scope of a project such as this. It is by far the most complicated route to take as it involves compiling into assembler to be called from our chosen platform-independent language. However, this was the

option taken as it was definitely the most applicable option in line with the aims of this project - to use advanced techniques in the pursuit of speed.

Dynamic recompilation is by its very definition the compiling of object code on the target processor. This is obviously a non-portable activity and it was seen as unlikely that a back-end for many processors would be written. As such, it was decided that the interpreter in the project should aim to be as fast as possible when it is unable to dynamically recompile. It was felt good that dynamic recompiling requires an interpretive emulation engine as it meant that if sufficient time could not be found to implement it, there was always the option of falling back on only normal interpretive methods.

## *4.4 Interpretive emulation*

The chosen method, dynamic recompilation, requires an interpretive engine as well as a compiler. This chapter describes the different ways the interpretive engine could be written. Emphasis, as always, is placed on making it as fast as possible.

The main goal for a 68000 interpretive engine is to decide, as quickly as possible, what the 16-bit CPU instruction that located at the Program Counter is meant to do. To illustrate this, in the following examples, the instruction $4E92 will be used to show how it would be recognised. In actual fact $4E92 is a varient of the JSR instruction which jumps to a given location in memory after storing a return address on the stack:

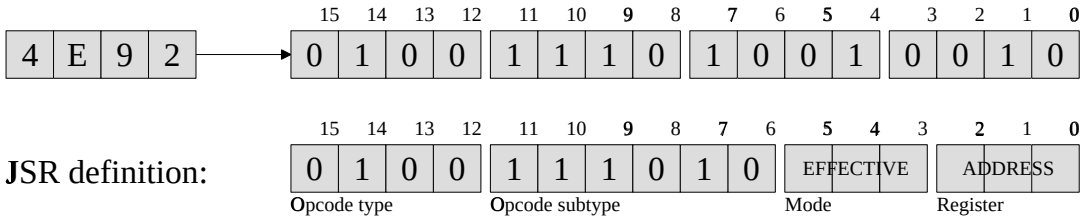


Figure 4.1 - Instruction composition and JSR definition

The effective address section of the JSR instruction is not unique and is used in most instructions in the 68000. It is important to remember the 68000 is a CISC processor and as such there can be dozans of varients of a single instruction.

The 'effective address' is a term used to mean an address that may require a calculation in order to find it.

The different types of effective address, indicated by the Mode and Register fields in an instruction definition, are shown below in figure 4.2.

| Addressing Mode Name | Syntax | Mode | Register |
|---|---|---|---|
| Data Register Direct | Dn | 000 | Data Register |
| Address Register Direct | An | 001 | Address Register |
| Address Register Indirect | (An) | 010 | Address Register |
| Address Register Indirect Postincrement | (An)+ | 011 | Address Register |
| Address Register Indirect Predecrement | -(An) | 100 | Address Register |
| Address Register Indirect with Displacement | w(An) | 101 | Address Register |
| Address Register Indirect with Index | b(An,Rx) | 110 | Address Register |
| Absolute Short | w | 111 | 000 |
| Absolute Long | l | 111 | 001 |
| Program Counter with Displacement | w(PC) | 111 | 010 |
| Program Counter with Index | b(PC,Rx) | 111 | 011 |
| Immediate | #x | 111 | 100 |
| Status Register | SR | 111 | 100 |
| Condition Code Register | CCR | 111 | 100 |

Figure 4.2 - 68000 effective address modes

Quite a lot of 68000 instructions also take varying sizes of data - JSR always takes a 32-bit address. Instructions such as ADD, OR etc. can affect 8-bits, 16-bits or 32-bits of information. For example it is possible to add the lower 8-bits of a data register to the location of an effective address. In this case the effective address is referencing one byte of information.

This background information should be helpful in understanding the problems that were faced - each different size and type of ADD requries a different method of implementation. Each size requires a different memory access type (byte, word and long) and each type requires a different way of locating the memory we need to access.

## Switch/case statements

There are two ways 68000 interpreters work - the simplest and most common method is to have switch/case statements to create simple jump tables, as shown below in figure 4.2. This is used in a variety of emulators including the *68k-simulator*[10] and EMU68[11].

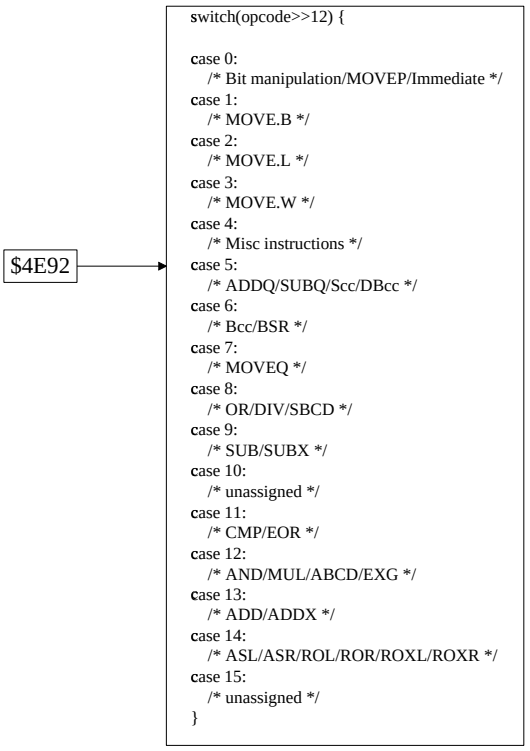


Figure 4.3 - Typical 68000 interpretive emulation decode table

This requires only a small amount of memory and quite quickly distinguishes the instruction set into 16 separate groups, determined by the value of their top nibble - 4 in the above example of $4E92. The only problem with this method is that there is still a long way to go before we work out what $4E92 really is.

The next step is to look at how the instructions in group 4 are arranged - all groups are different. In this particular case we must first check bits 6-8 to identify a few instructions which are orientated slightly differently than the rest. Once that has been done we can then do another jump table on bits 8-11 in the same way as before. At this stage we know that the instruction corresponds to about 15 different instructions, however, only the bottom 8 bits are left undecoded. Checkings bits 6-7 will reveal that this is a JSR instruction. The bottom 6 bits are an effective address indicator as shown previously.

To decide what must actually be done to find the address to jump to, some more decoding must happen to decide on the effective address. $4E92 is actually an 'Address Register Indirect' operation on address register 2. The function to implement JSR will have to work this out and then load the address register into memory.

We've already demonstrated how we'll need at least 2 jump tables and a couple of comparisons before we know what the instruction is, plus another jump table for the effective address computation. In most programs this wouldn't matter, but since we're performing millions of these instruction decodes per second, every instruction counts.

## One jump table

There is an alternative. Rather than have a series of jump tables and checks to work out which routine can cope with the next instruction, we can have just one jump table that has 65536 entries - one for each CPU instruction, from $0000 to $FFFF.

Hence, to decode $4E92 we just look-up the address contained in the $4E92th entry of the table and branch to it. The function that we end up with could be an all-purpose JSR routine that works out the effective address and then does the jump - however, if we wanted ultimate speed we could write a function for all different types of JSR. You should be able to see that this would allow for functions that have no decision making in them - once we've looked up the $4E92th entry we have a function that deals with specifically JSR 'Address Register Indirect'. We could go further and have a function that deals with just 'Address Register Indirect from register 2' for just $4E92 which would remove an 'instr AND $0007' to decode the register required. We have to be careful however that we don't go too far as otherwise we run the risk of having 65536 different functions taking a not inconsiderable amount of memory. Another consideration not mentioned before is that the bigger the program the less can be cached in the target processor's data and instruction caches.

The question that presented itself now was how should we construct the 65536-entry jump table and do we really want to write potentially thousands of functions by hand?

Quite simply it is impossible to construct a 65536 entry table by hand without introducing errors. Not only that, if we want to have a different routine for every type of instruction (there are only 7 different types of JSR, but there are 158 types of AND and 298 types of ADD!) then it would seem very difficult to construct the functions required by hand. It was to this end that it was decided that there must be a way to construct both the table and the functions automatically using a template.

The differences between different types of instructions are quite minor. There is always the actual action, e.g. 'output = sourcedata AND destdata' for the AND functions, but where to get sourcedata and destdata are different, plus where to put the output varies. A template for each set of instructions therefore seemed possible, and a program could process this template and add on the necessary bits of code at the start and end so that the function could get the data and store the answer in the correct places.

## *4.5 Function generation*

Warning: This section is quite complicated. If you have an understanding of the 68000 it is a great help to the understanding of what we have to do.

The first step in generating the functions was to describe all the different instructions. Each instruction has a set of bits that must be set to a particular value and other bits that can be set to specify a different type. For example, the JSR instruction contains 10 bits that have to a particular value in order for it to be a JSR instruction, and the other 6 bits relate to how to locate data for the instruction (the effective address).

The JSR instruction cannot use a data register to get the address to jump to, it must be an address register. As such the effective address Mode 000 (Data Register Direct) is not valid. Unfortunately for us the designers of the 68000 decided that they would use these single unused instructions dotted about the address space for other instructions. This means that it is not good enough to describe a JSR instruction as being able to be any combination in the lower 6 bits. As a result of this we must specify each and every valid type for the JSR instruction so that there are no two descriptions for the same instruction - our jump table builder will throw an error if it detects this situation.

### Instruction definition file

The format of the description file changed a lot during the design and development as new things needed to be added, but the entry for JSR ended up being a single entry:

```
0100 1110 10 eee EEE    0 1     -----   -----   JSR.L      e(*,-Regs,-Amod,-Imm)
```

This needs describing. The first 16 characters indicate which bits can vary. In this case we have 10 bits that must be a particular value plus 6 bits that are special. The *eeeEEE* corresponds to an effective address - the lowercase 'e' represents the Mode field and the uppercase 'E' represents the Register field. There are then some flags which do not need explaining now and then there is a description of what the instruction really is. Firstly 'JSR' indicates the name of this instruction and hence what template will emulate it. The '.L' indicates *Long*, meaning that all 32-bits are used in this instruction, so any data required from the effective address will be fetched in full. Lastly we have the parameters of the instruction, in this case *e()* specifies that the parameter comes from the *eeeEEE* bits and the modes accepted are all (represented by the *) minus the Register Direct modes (-Regs), the modes that modify an address register (-Amod) and the Immediate mode (-Imm).

From this information the program that scans it can deduce which bit patterns are JSR instructions and what parameters will be required by the function that implements it.

A more complicated example is the ADD instruction family - as already mentioned these rules expand to 298 different types of instruction:

```
0000 0110 zz eee EEE    0 0     -----   XNZVC   ADD.z      #z,e(*,-Areg,-Imm,-PC)
0101 iii0 zz eee EEE    0 0     -----   XNZVC   ADD.z      #I,e(*,-Areg,-PC,-Imm)
0101 iii0 01 eee EEE    0 0     -----   -----   ADD.W      #I,e(Areg)
0101 iii0 10 eee EEE    0 0     -----   -----   ADD.L      #I,e(Areg)
0101 0000 zz eee EEE    0 0     -----   XNZVC   ADD.z      #8,e(*,-Areg,-PC,-Imm)
0101 0000 01 eee EEE    0 0     -----   -----   ADD.W      #8,e(Areg)
0101 0000 10 eee EEE    0 0     -----   -----   ADD.L      #8,e(Areg)
1101 nnn0 zz eee EEE    0 0     -----   XNZVC   ADD.z      e(*),n(Dreg)
1101 nnn1 zz eee EEE    0 0     -----   XNZVC   ADD.z      n(Dreg),e(*,-Regs,-PC,-Imm)
1101 nnn0 11 eee EEE    0 0     -----   -----   ADDA.W     e(*),n(Areg)
1101 nnn1 11 eee EEE    0 0     -----   -----   ADDA.L     e(*),n(Areg)
1101 nnn1 zz 000 NNN    0 0     X-Z--   XNZVC   ADDX.z     N(Dreg),n(Dreg)
1101 nnn1 zz 001 NNN    0 0     X-Z--   XNZVC   ADDX.z     N(Adec),n(Adec)
```

Lots of different types of ADD may come as no surprise to a seasoned CISC user. Infact many of these ADDs are infact generalisations. The 68000 actually has ADD, ADDQ (add quick), ADDI (add immediate), ADDA (add from address register) and ADDX (add with extend). If we describe the instructions generically enough we can perform most of these with the basic ADD template - you will notice that there is no ADDQ function type in the definitions above. The ADDQ instructions are those that have 'iii' in the bit-pattern indicating that the source operand can be fetched from the bits themselves. When we come to generate the functions we can detect this case automatically and hence no longer need to write a seperate template for ADDQ.

Introduced above are a number of different operand types. Add instructions have two operands separated by commas, each has an initial letter that generally corresponds to a particular section in the bit-pattern. For example if the source operand is always a particular effective address Mode then the letter 'n' corresponds to the register number location in the bit-pattern. The operand description will specify how this register is to be interpreted, for example n(Adec) in the last entry means that the register 'nnn' is to be interpreted as Address Register Indirect Predecrement mode and that the register is in bit positions 9-11.

You will notice that some entries have a size field of '.z'. This indicates that the size is specified by the 'zz' bit-pattern.

```
1101 nnn0 zz eee EEE    0 0     -----   XNZVC   ADD.z      e(*),n(Dreg)
```

This descriptor expands to all 12 different effective address modes, plus each mode has 3 different sizes (byte, word and long) making 36 different types. Although it is possible for the destination Data Register, marked by 'nnn', to be expanded too the penalty for not doing so is only an AND instruction in each function. If we did expand them then we would end up with 288 different functions for this one descriptor alone! This would be a waste of memory and there are no control-flow issues, so expanding this is not considered a good idea.

## Creating functions

It was anticipated at this stage that after parsing the descriptors that a program would need to be written that would cycle through all possible valid instructions and output source code to all the different functions. The 'template' mentioned wouldn't actually be a template but rather a bit of code that calls functions to output source to the file being generated. The process would be:

- Output code to get source effective address
- Output code to get source data (if required)
- Output code to get destination effective address
- Output code to get destination data (if required)
- Output code to perform action on source and/or destination
- Output code to put output data to destination effective address, if applicable.

Everything above depends on the actual instruction type. For 'JSR' there is no destination information - JSR would get the source effective address, as this is where we want to jump to, and then set the PC to that address. The routine to output the source-code for getting the effective address would see what particular effective address method this function is trying to provide support for, and output different source depending on this. In this manner all different types of JSR will end up being supporting.

More information on how this is actually accomplished is presented in the Implementation section.

## Disassembly

One of the requirements of the user interface, and of course debugging, is to have a disassembler so that the actions of the emulator can be traced.

The beauty of the function generation and the descriptors of each instruction is that there is a textual equivalent of the instruction built into the description. To do a disassembly all that is required is that the bit-pattern is looked up and the information presented in the descriptor can be used to generate a disassembly.

This means that any disassembled instruction indicates exactly what the interpretive section will do when it encounters the instruction. There being an integrated disassembler will aid in the debugging, allow detection of errors at a glance and mean a trivial amount of work to write the disassembling routines. This was considered a real bonus.

# System Design

## *5.1 Language*

The language chosen for this project was C. There were a number of reasons for this:

**Portability**
C is a very portable language and can be compiled on nearly every computer in existence. Although there are a wide number of extensions in the different compilers available it was decided that in most circumstances a portable alternative can be used.

**Low-level**
Although C is sometimes considered high-level it is infact as much low-level as you can get in a portable manner. C allows a programmer to directly manipulate structures and bytes in a way that is very advantageous in a emulator.

**Not object orientated**
Although OO languages are normally considered better for most applications an emulator does not need the facility of objects. The only advantage would be the strong enforcing of entry/exit conditions on functions, but these can be overcome through the proper use of the language and header files. OO languages normally have more enforced rules on typing which may prohibit the low-levelness required of this project.

**First language for unix**
The best market for a portable application is the unix range of workstations. Most operating systems that are not unix will require applications to be substantially modified even if the C source itself does not need to be altered. Since unix operating systems are written in C themselves it makes it very easy programming in C on them.

## *5.2 Operating system*

Although in some respects this goes hand-in-hand with the choosing of a language, the operating system chosen was unix. We shall see that during the course of this project it was ported to a non-unix operating system, Acorn's Risc OS, which shows the level of portability attained in this project.

Unix has a number of facilities that aided the development of this project:

**Pre-emptive multitasking**
Although common in operating systems these days, the pre-emptive multitasking in unix is perhaps the best available anywhere. Since we will be doing some very low-level activities in the emulator it is important that the machine will not crash due to the actions of one program. The pre-emptiveness also aids development and debugging as many things can occur at once.

**Make**
Compiling C programs is done through the use of the CC program, however, rather than issue a lengthy command each time we want to compile something a Makefile can be used to work out what needs compiling. This allows a programmer to worry more about the content rather than worrying about compilation, dependencies etc.

**Imake**
This project originally used Imake to create the Makefiles necessary for compiling the programs. Imake allows you to specify a portable description of what is to be compiled and how. This allows you to take the this portable file to another unix machine and get it to make a new Makefile using the correct syntax for that particular variant of unix. Although the project didn't end up using Imake it was considered a worthwhile exercise to learn about it.

**Autoconf**
It soon became clear that choosing Imake was limiting. Switching to autoconf improved the portability of the project and allowed for more detection of the underlying environment. Autoconf works in much the same was as Imake except that it produces a shell script that can be run without any installed output processor, which aids portability. New functions to detect different variants of unix can be written much more easily than Imake. One of the facilities offered by Autoconf was the ability to easily declare unsigned integer types that we can definite the size of, e.g. *uint8* for unsigned integer of 8 bits, *sint16* for signed integer of 16 bits, etc.

## *5.3 X Toolkits*

X Windows is the windowing environment common on unix workstations. C programs talk to X through the xlib library which provides very low-level calls for drawing objects in the windows.

Rather than having to draw everything by hand it is more common to use a toolkit that can do it for you. These toolkits can normally support different styles of, what are known as, widgets. Widgets can be any object in a window, from simple lines and text to complex sliders, scrollbars and buttons.

**Xt/Athena**
This is the standard combination for X and comes as standard. Xt is a toolkit that can support different styles of widgets, in this case Athena is a 2d widget set that is common to see on early X applications. Although it serves its purpose it is out of date in both style and functionality and is best avoided. Since it was desired that the application that would fit in with today's current applications this combination was not chosen.

**Xt/Motif**
Motif is a newer widget set that provides 3d windows and buttons that fit in with today's desktop environments. This would definitely have been a good choice for this project but unfortunately Motif is a copyright package and requires that the user has bought a copy. Since it is not freely available and as such Motif had to be dropped from consideration.

**Xt/Lesstif**
Lesstif is a free version of Motif. Unfortunate it was in beta at the time of this project and was not considered stable enough. It is not as widespread as other toolkits here and it was felt difficult to rely on being present on a user's machine.

**Tcl/Tk**
Tcl is a scripting language that has a library called Tk that adds on commands that can be used to control the appearance of the application. Tcl was originally written for the purpose of having Tk so they are well partnered. Tcl/Tk has the benefit that the script can be distributed with the application and hence is alterable by the user. The only downside is that a new language has to be learnt in order to use it.

**Perl/Tk**
This is the same as Tcl/Tk except that the scripting language is the well-known Perl. This is a lot easier than having to learn Tcl but has the downside that it is not as supported a combination. Most people who use Tk expect that the underlying language is Tcl and all documentation and support is for Tcl/Tk, which is a problem.

Due to the availability of books and lots of documentation, Tcl/Tk was chosen over Perl/Tk. The problem of having to learn a new language was considered negligible compared with the lack of documentation that may cause difficulties later if Perl/Tk was used.

## *5.4 Modules*

This project can be divided into modules in much the same way as the system was constructed in hardware by Sega. This section explains the strategy used to create the modules and how the project was broken down to make it easier to implement.

The initial design is presented here and all changes are presented in the next chapter.

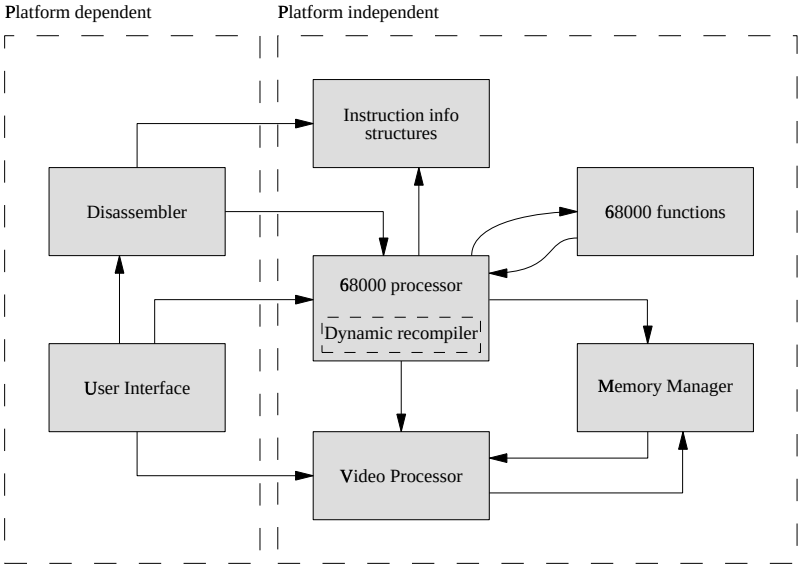


Figure 5.1 - Modules

The above model is vastly simplified but tries to show the flow of information, the left hand grouping of modules consists of the platform dependent elements and the right hand grouping the platform independent elements. In this way it is possible to port the application to another architecture with the minimum of fuss.

As shown above, the User Interface starts operations either through the 68000 emulation module or through video display and hence will talk to the 68000 processor or the video processor. The memory manager is somewhat at the heart of the project in that the 68000 will almost always communicate with other parts through memory mapped i/o.

Each module is detailed below in the manner that I believed it would operate at the design stage, and in most respects was what was implemented.

# 5.5 User Interface

**Purpose**
To provide the platform-dependant features required for user interaction.

**Description**
The user interface is very important to a user and in most areas of computing can decide on whether an application is successful or not. Although the main responsibility of the user interface is to display the output from the emulated system, it was important to have an easy-to-use program that could be extended easily.

Quite early on it was decided that Tcl/Tk would be the best way to achieve these design goals. Tcl/Tk allows the programmer to test user interface design ideas quickly through modifications on a non-compiling script, plus it allows the user to alter the application's appearance once it has been distributed.

A number of facilities were seen as required in addition to the obvious display output:

- To be able to load the game images.
- To allow the user to change the parameters of the emulation.
- To have debugging facilities including a disassembly, memory dump and register window.

The ability to load the games was of course the first step, and I envisaged a loading window much like that of Netscape's, as this is the most common 3d user interface around. The loading window and window with a menu-bar from Netscape are shown below, as examples of the style.

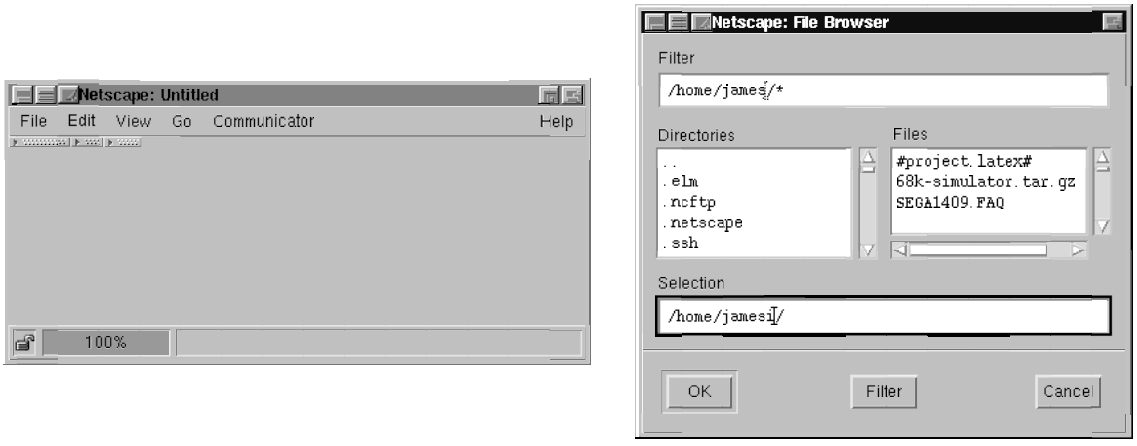


Figure 5.2 - Netscape dialogue windows

The debugging environment is of utmost importance as without it there would be no hope in fixing any problems encountered during development. A great deal of emphasis was placed on its design to ensure it was a help and not a hindrance.

Any additional facilities would enhance the user's opinion of the application and hence would be welcomed if time permitted. One such facility that was hoped could be added was the ability to resize the emulated screen. The standard 320x224 is a little small on today's high-resolution monitors.

## *5.6 Memory manager*

**Purpose**
To provide a fast method of lookup for the 68000 memory map.

**Description**
A processor, and of course the emulation of it, makes many memory lookups during operation - infact there is always at least one memory read per instruction to read the data at the current Program Counter.

Many emulators perform this function through the use of C switch/case statements which allow the C compiler to build up a series of comparisons to work out which part of memory a lookup is occurring at and hence perform different actions - don't forget, most memory read/writes are not writing to a block of memory but are instead causing some action.

This method of memory accesses can be improved upon considerably and it was decided that an array of function pointers, in a similar fashion to the emulator, would be much faster. In this manner when we want to read from a location of memory we can index into an array the function that can deal with this memory access.

The 68000 memory map can address up to 16.8MB of information: $000000 to $FFFFFF. It would be feasible to have an array of function pointers this size but most workstations would have problems with this amount of memory. Instead of this we notice that the memory map is in sections (see figure 2.2). Initially it seems a good idea to have 16 of these sections so that $000000 to $0FFFFF goes to one function, $100000 to $1FFFFF goes to another and so on. However, if you look at the memory map you will notice that an increased resolution of $1000 will allow us to distinguish between a lot more - for example the I/O and the Control areas:

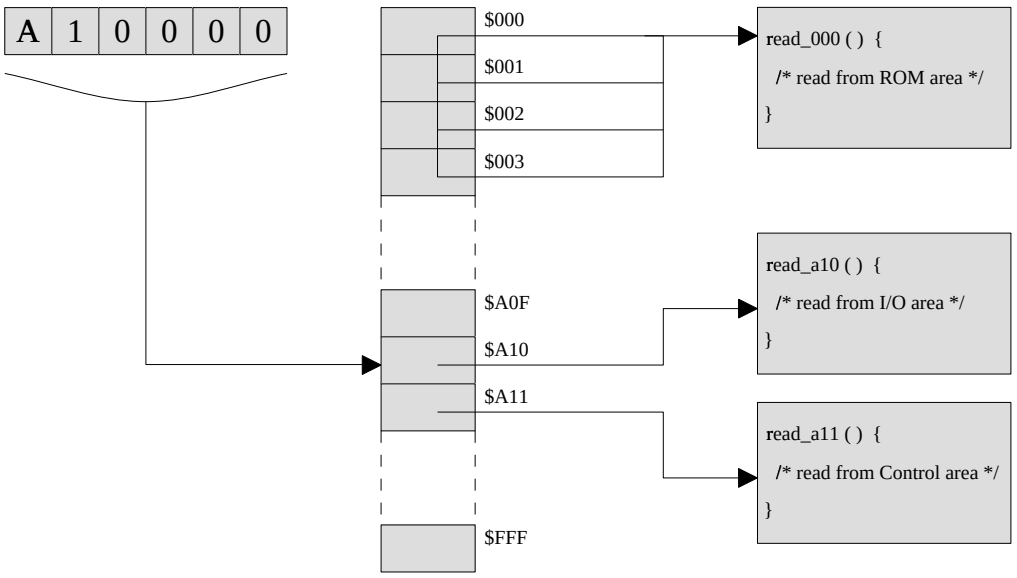


Figure 5.3 - Memory fetch decode

As you can imagine it becomes a trade-off between memory and speed. The $1000 resolution depicted above is nice because it distinguishes between every major location and consumes only 16k of memory (12-bit index means 4096 entries of 4-byte function pointers). This resolution takes a 24-bit address and takes the top 12-bits as an index and the bottom 12-bits as an offset.

This function-array will require constructing at run-time using a structure describing what functions should be called for which blocks of memory. For speed a different routine for reading and writing should be constructed.

## *5.7 68000 processor*

**Purpose**
To emulate the functions of a 68000 CPU interpretively.

**Description**
A method of how the 68000 core will be written was outlined in the previous chapter. The emulation will consist of the following:

- Description file of all 68000 instructions: *def68k.def*
- Program to convert the descriptors into a machine readable C structure: *def68k*
- Program to take the structures and generate C source for each function type: *gen68k*
- Main source that will contain the virtual machine engine and call the functions.

We split the action of going from the textual descriptions to source code functions into two programs, *def68k* and *gen68k*. This helps to modularise but also means that the C structure generated by *def68k* can be used in the main source. This is important for the disassembler which needs access to this information to generate a textual disassembly. It is also used for the dynamic recompiler which needs to know the end of blocks, but this is discussed later.

We also need to be able to keep an accurate record of how many clocks are emulated for each instruction so that we can accurately simulate the interrupts required for the Video Display Processor. This can be incorporated into *def68k* - each instruction takes a specific number of clock cycles dependant on its basic type (AND, ADD etc), excluding data fetches and stores. As the program outputs the structure describing the instruction type it can work out how many clocks it takes and record this into the structure for the main program to read later.

The main source file will have at its disposal a structure for each instruction type and a function that it can call to emulate it. It is responsible for:

- Creating the 65536-entry function table.
- Creating another table to convert from the instruction to the information structure for that instruction - used by the disassembly routines, for example.
- Maintaining the virtual state of the CPU, including registers, clock counts, PC etc.
- Executing the code by fetching the instruction and calling the correct function.
- Causing interrupts when required by the PAL/NTSC protocol.

## *5.8 Video Processor*

**Purpose**
To emulate the functions of the Video Display Processor.

**Description**
The 68000 processor keeps track of how many clocks it has emulated. When the number of clocks corresponds to the length of time it takes to output one line of PAL/NTSC it will generate an interrupt for the program running on the 68000 to detect. The Video Processor will then transfer the data, from the 68000's RAM, required to draw the image for the next line. This proceeds until all 224 lines have been drawn. When this has completed the Video Processor will inform the User Interface that it can display the frame to the user and passes information such as the frame buffer and the palette required.

The Video Processor is responsible for:

- Plotting each line of the frame as quickly as possible.
- DMA transfers between the 68000 RAM and the Video RAM areas.
- Maintaining the 25 video registers.
- Collision and overflow detection.

More details on the above can be found in Chapter 3.

## 5.9 Disassembler

**Purpose**
To disassemble 68000 instructions.

**Description**
The disassembler is necessary for debugging the project. It would not be very prudent to assume that the project will work first time and hence it is vital that there are the proper development tools available to trace the actions of the emulator.

The disassembler will enable instructions to be decoded and shown to the user. Firstly this should allow for spotting of simple mistaken identity, e.g. when an instruction is being incorrectly decoded. Secondly using single-step debugging through the 68000 code and watching what happens it should be easy to check with the disassembly output whether or not the desired action is being performed correctly.

This module should be detachable from the rest of the project as it would not be required in a non-development environment.

## 5.10 Dynamic Recompiler

**Purpose**
To provide dynamic recompiling on support architectures

**Description**
The beauty of a dynamic recompiler is that it is designed to be separated from the rest of the project so that on unsupported architectures the program as a whole continues to operate.

This part of the project was implemented last, but it was by no means added on design wise at the last minute. There are many key requirements for dynamic recompilation that could not be ignored during the design and implementation of the other parts.

Dynamic recompilation is simply taking a 68000 instruction and creating some machine code that would perform the same function. It was decided that the implementation of the dynamic recompiler would be carried out for the ARM series of target processors. The reasoning behind this is simply that there was previous experience of the assembly language and thus this made it a more sensible choice. ARM's processors are within many computing devices, from Psion 5's to Network Computers, however, it is the fact that they are at the heart of Acorn computers that made it a good choice. Acorn's are quite easy to program on and the native operating system being single-user meant that we could access very low-level calls to simplify the process.

**Blocks**

The interpretive engine is going to be relatively fast for an interpreter, as we are going to generate specific functions for each separate type of instruction. As such, the dynamic compiler will have to be extra special in order to improve upon the speed. Of course, hand compiling to native code is always going to be faster than running C-compiled code - the question is, how much faster, and will it be worth the effort?

Compiling a single instruction is at the heart of a dynamic recompiler but in order to be effective the compilation must proceed on the basis of blocks.

A block of instructions, where the flow of execution is linear (i.e. there are no branches out of the block), has a lot of scope for optimisations before we go ahead and actually create code for it. For example, redundant flag calculation removal, as presented below.

## Redundant flag calculation removal

There are 5 status flags in the 68000 processor: X, C, N, V and Z. These flags represent what the state of the last CPU instruction was. Nearly all instructions affect these flags but most of the time the result is never checked. The one time they are used is when there is a control-flow decision to make. This usually takes the form of a branch instruction that will only take effect when certain flags are set. For example, the 68000 instruction 'BNE $1000' means that we should only branch to location $1000 when the Z flag is clear, as this indicates *Not Equal to*.

Branch instructions do not occur after every instruction, as such, there are flags that are needlessly calculated that will never be looked at. This is a very serious issue for an emulator: an instruction that adds two registers together involves just a few assembler instructions on the target processor, but to set the flags will require dozens.

If you recall, when we discussed the descriptors for each instruction, you will remember that we had some unexplained flags. These flags are actually for this optimisation and list which flags each instruction needs to look at and which flags the instruction may or may not alter.

Every time a new section of code is encountered we will set about identifying it as a block. The start of the block is just the start of the unknown code we encounter, and it ends with the first instruction that might change the PC. The instructions that do this are flagged as such in the descriptor.

An example section of code, which could form a block, is shown below in figure 6.2.

| Instructions | | Flags affected | Flags checked |
|---|---|---|---|
| move.l | #5, d1 | NZVC | none |
| move.b | d2, d3 | NZVC | none |
| beq | block2 | none | Z |

Figure 6.2 - Redundant flag calculations

The first instruction in the block moves the value 5 into data register 1. This is not a pointless instruction, it is doing something that might be used later, however, the flags N, Z, V and C are altered to reflect this operation. The next instruction moves the lower 8-bits of data register 2 to data register 3. The branch instruction will cause the execution to go to block 2 if the Z flag is set. This will actually occur when the lower 8-bits of D2 represent 0, however, this is not important. What is important is that if the emulation engine were to calculate what the four flags should be set to after the first instruction, it would be a waste of time.

The technique, after a block has been identified, is to start at the end and see what flags are required. We then work backwards and record in our cache what will be needed later. Once we have found an instruction that will create the flags required we stop looking for those flags as they are no longer required to be generated by instructions earlier.

The optimisation of the block in figure 6.2 would result in no flags at all needing to be generated by the first instruction, and just the Z flag being generated by the second instruction. As you can see, the number of flag calculations has the potential to drop from 8 to 1. This is in actual fact not quite correct - we don't know what will happen after the block we are analysing has finished and hence we must make sure all the flags are correct at the end. Hence, in reality the second instruction would need to update all 4 flags. The bigger the block, the more scope for redundant flag removal, though, and a 10 to 1 reduction should not be unusual.

## Dynamic recompilation of a block

Once we have executed a block a specified number of times, which could be no times at all, we will want to compile it. Obviously we only have this option on supported architectures and the 68000 interpretive engine would use the recompiler only when appropriate.

The compilation will proceed much better now that we know what flags should be generated by each instruction in the block. The compiler will also have some sort of register allocation policy that would allow 68000 registers to be cached in real target processor registers. This should cause a significant speed-up as we would not need to store and re-load a register value needlessly, as we would during the interpretive engine.

# Implementation

## *6.1 The Plan*

The 68000 ROM data on the cartridge is read using a special device that sequentially reads the data out and stores it to a file. Many places on the Internet have the cartridge images available for download and hence there was no reason to construct or purchase a reader for this project.

Several of these images are public domain, written by authors who distribute the ROM images, including 68000 source code, to show how a game can be constructed for the Genesis. This form of input data is an invaluable tool and it was decided that it should be the primary goal to emulate them. The simplistic nature of these images and easy to follow source (and comments) enables easy debugging of an emulator, and once working with these images more complex and real-life games could be tried and tested.

The complexity of this project meant that no results could be seen until very near to the end of the project's life. There was unfortunately no way around this - just one bug is likely to prohibit everything from working.

The first step was the development of the compile-time programs for the CPU emulation - these programs output files for the main program to use.

Once the source for the emulation functions had been generated the next step was to develop the virtual machine that would call them. At this stage it would be difficult to proceed without a user interface and this was seen as the next logical step. This would allow the virtual machine to be stepped through and the results viewed by looking at the memory and register contents presented.

Stepping through code and observing the results and how memory is affected will only get you so far. Small non-obvious bugs are likely to be observed only when the output is incorrect. Since all output is based on the Video Processor it would then be important to develop that.

The goals for this project were to demonstrate, working:

- A basic 68000 interpreter.
- A test image showing the Video Processor and interpreter working together.
- Commercial cartridge image demonstrating a functioning game.
- Dynamic recompiling 68000 engine with the test image.
- Commerical image using the dynamic recompiling engine.

## *6.2 Generating Instruction Information Blocks*

The first step was to take the instruction descriptors and convert them into a machine readable structure. This structure was actually implemented as an array of Instruction Information Blocks (IIBs). The C structure for an IIB is presented below:

```
typedef struct {
  uint16 mask;              /* mask of bits that are static */
  uint16 bits;              /* bit values corresponding to bits in mask */
  t_mnemonic mnemonic;      /* instruction mnemonic */
  struct {
    int priv:1;             /* instruction is privileged if set */
    int endblk:1;           /* instruction ends a block if set */
    int imm_notzero:1;      /* immediate data cannot be 0 (if applicable) */
    int used:5;             /* bitmap of XNZVC flags inspected */
    int set:5;              /* bitmap of XNZVC flags altered */
  } flags;
  t_size size;              /* size of instruction */
  t_datatype stype;         /* type of source */
  t_datatype dtype;         /* type of destination */
  unsigned int sbitpos:4;   /* bit position of source imm data or register part of EA */
  unsigned int dbitpos:4;   /* bit position of destination register part of EA */
  unsigned int immvalue;    /* if stype is ImmS this is the value */
  unsigned int cc;          /* condition code if mnemonic is Scc/Dbcc/Bcc */
  unsigned int funcnum;     /* function number for this instruction */
  unsigned int wordlen;     /* length in words of this instruction */
  unsigned int clocks;      /* number of external clock periods */
} t_iib;                    /* instruction information block */
```

This structure contains everything about the instruction that can be gathered from a descriptor. The first two entries, the *mask* and *bits* are unsigned 16 bit numbers that correspond to the bits that must be set in order for this IIB to apply to an instruction. Note that the program that outputs these IIBs has already split each instruction descriptor into the different sizes and effective address modes. One IIB covers one group of instructions, where a group can differ no more than just the register or immediate value that is used. To remind you this means that each function will have no control-flow decisions. The *mask* and *bits* are used to construct the 65536-entry tables.

The *mnemonic* specifies which template will be used, e.g. ADD, OR, etc. The scanning program ensures that the descriptor contains a valid mnemonic type. This is used by the function creating program to determine which template to use, it is also used by the debugger to indicate to the user what instruction this represents.

The *flags* structure contains such miscellaneous information that, for the most part, are necessary for the dynamic recompiler - e.g. whether this instruction ends a block.

The size, source type and destination type are simply read from the descriptor. The *sbitpos* and *dbitpos* contain where in the 16-bit instruction the bits are that give the varying part of the bit-pattern - normally the register number but sometimes immediate data.

The other important entries to note are the *funcnum*, which indicates what function number this IIB is for (useful for debugging to indicate what routine it is going to call to emulate this instruction), *wordlen*, which indicates how many words this instruction uses (so that the CPU can increment the PC) and *clocks*, which indicates how many clock periods to add to the counter after this instruction has been executed.

## *6.3 Program: def68k*

**Source files**
def68k.c

**Purpose**
Responsible for reading in the descriptor file that describes all possible 68000 instructions and for producing machine-processable output files useful for other programs.

**Input files**
def68k.def

**Output files**
def68k-iibs.h
def68k-funcs.h
def68k-proto.h

**Description**
The input file is listed in appendix *** including full syntax information. This program processes each line and ensures it is valid. It then expands each descriptor so that there should be no control-flow decisions in the implementation of the function to emulate the corresponding instruction. This involves outputting Instruction Information Blocks that have a particular size, source effective address type, destination effective address type and if applicable condition code type.

The def68k program is also responsible for calculating the length in 16-bit words of each instruction and now many clock cycles it takes to execute. These values are stored alongside the information from the descriptor and stored in the IIB - detailed in the previous section.

Two additional files, def68k-funcs.h and def68k-proto.h are produced. They do not contain any information as such, just structural information required by other parts of the project.

**def68k-funcs.h**
Contains an array of function pointers so that the IIB *funcnum* number can index to a function pointer.

**def68k-proto.h**
Contains the prototypes for the functions that will be created by the gen68k program.

## *6.4 Generating the functions*

After constructing the Instruction Information Blocks it was necessary to write a program to generate a function that would emulate each of them.

Each IIB specifies the source effective address type and the destination effective address type. This information is used to determine how each function will find the operands to emulate the instruction and possibly how to store the result.

The 68000 works using one or two effective addresses. These are called the source and destination effecitve addresses and the instruction may use either the address or the data located at that address.

Figure 4.2 detailed the different types of effective address. For our purposes we extend the different types so that we can combine certain types of instructions together - e.g. it has already been shown how ADDQ can be incorporated into ADD. ADDQ's source data comes from immediate data encoded into the bit pattern itself (hence why it is called Add Quick). We therefore include an effective address type 'Immediate 3 bits' which means the value comes from 3 bits worth of information in the bit-pattern.

The structure *t_datatype*, as used in the IIB, contains an enumerated list of the different source and destination types, you should recognise most from the 68000's effective address modes. A description of each is presented below:

| Type | Name | Effective Address | Effective Address Value |
|---|---|---|---|
| dt_Dreg | Data Register Direct | n/a | Data Register contents |
| dt_Areg | Address Register Direct | n/a | Address Register contents |
| dt_Aind | Address Register Indirect | Address register | Memory location contents |
| dt_Ainc | Address Register Indirect Postincrement | Address register | Memory location contents |
| dt_Adec | Address Register Indirect Predecrement | Address register - size | Memory location contents |
| dt_Adis | Address Register Indirect with Displacement | Address register + Displacement | Memory location contents |
| dt_Aidx | Address Register Indirect with Index | Address register + Index register + Displacement | Memory location contents |
| dt_AbsW | Absolute Short | Given 16-bit sign extended address | Memory location contents |
| dt_AbsL | Absolute Long | Given 32-bit address | Memory location contents |
| dt_Pdis | Program Counter with Displacement | PC address + Displacement | Memory location contents |
| dt_Pidx | Program Counter with Index | PC address + Index register + Displacement | Memory location contents |
| dt_ImmB | Immediate Byte | n/a | 8-bit value at PC+2 |
| dt_ImmW | Immediate Word | n/a | 16-bit value at PC+2 |
| dt_ImmL | Immediate Long | n/a | 32-bit value at PC+2 |
| dt_ImmS | Immediate Specified | n/a | Specified value in IIB |
| dt_Imm3 | Immediate 3-bit | n/a | 3-bit value in bit-pattern |
| dt_Imm4 | Immediate 4-bit | n/a | 4-bit value in bit-pattern |
| dt_Imm8 | Immediate 8-bit | n/a | 8-bit value in bit-pattern |
| dt_Imm8s | Immediate 8-bit signed | n/a | 8-bit signed value in bit-pattern |

Figure 6.1 - Source and Destination data types

For each IIB there is a source and destination type from the above. Note that the destination type cannot be one of the immediate data types as there is no address to store data to. This and other invalid situations are detected.

The generation of each function proceeds by looking at what mnemonic is indicated in the IIB and then using the appropriate template routine. This routine will then call a small number of functions to output the desired source to locate the data.

```
generate_ea(output, iib, tp_src, 1);          /* generate srcaddr */
generate_eaval(output, iib, tp_src);          /* load srcdata from srcaddr */

generate_ea(output, iib, tp_dst, 1);          /* generate dstaddr */

generate_outdata(output, iib, "srcdata");     /* outdata = srcdata */

generate_eastore(output, iib, tp_dst);        /* store outdata in dstaddr */
```

The code above is all that is required to generate all the different types of MOVE, i.e. all 3 different sizes (byte, word and long) and all the different types of source and destination effective address modes (301 in total).

The procedure is:

- Output code to generate *srcaddr*, the address of the source operand
- Output code to generate *srcdata*, the data in the address of the source operand
- Output code to generate *dstaddr*, the address of the destination operand
- Output code to generate *outdata*, the output data
- Output code to generate a store of *outdata* to *dstaddr*, the address of the destination

Let's look at the first two lines:

```
generate_ea(output, iib, tp_src, 1);          /* generate srcaddr */
generate_eaval(output, iib, tp_src);          /* load srcdata from srcaddr */
```

The *generate_ea* function writes to the file handle *output* using information from the given IIB. This function can output either the address of the source or destination operand, and in this case *tp_src* indicates that we want to get *srcaddr*. There is also a flag that indicates whether or not we want to perform any updating, e.g. for a postincrement whether or not to increment the address register afterwards. The *generate_eaval* function then outputs source-code to take the *srcaddr* and produce *srcdata*, which usually involves a memory fetch.

The *generate_ea* function is clever and will not output anything if it is not required for the current mode given in the IIB, for example if the value is not in an address but immediate data.

Some examples of what these two functions will output:

**Source type** Address Register Indirect
**Mnemonic** MOVE
**Size** Word (16-bits)

```
int srcreg = (instr >> 6) & 7;
uint32 srcaddr = ADDRREG(srcreg);

uint16 srcdata = fetchword(srcaddr);
```

*generate_ea* has first produced source-code that will find out what the register is that the effective address is contained in. It allocates a C variable for convenience - it will not actually get stored anywhere as the C compiler will optimise it out. The source address is then loaded from the register and stored in a 32-bit unsigned integer (uint32) *srcaddr*.

*generate_eaval* then calls *fetchword* which reads a 16-bit value from *srcaddr* and places it in a 16-bit unsigned integer (uint32) *srcdata*.

**Source type** Immediate 3-bit
**Mnemonic** MOVE
**Size** Byte (8-bits)

```
/* no address for Immediate 3-bit */

unsigned int srcdata = (instr >> 6) & 7;
```

The functions have generated completely different code for Immediate 3-bit. generate_ea doesn't output anything in this case as there is no address associated with this mode. generate_eaval fetches the effective address value from the instruction bit-pattern itself. The 6 is from sbitpos in the IIB which if you remember is itself worked out from the position of the descriptor 'iii' bits, discussed in section 4.5.

### Summary

The descriptions above are perhaps difficult to understand at first glance but the following key points can be derived from them:

- A small section of code can generate hundreds of variants of a single instruction.
- The code is easily understood and almost high-level in nature.
- The functions that generate the code are small and perform simple actions.
- Any unnecessary code that is generated will be optimised out by the compiler.
- Whilst writing the templates emphasis can be placed on function and not implementation.

## *6.5 Program: gen68k*

**Source files**
gen68k.c

**Purpose**
Generates the function definitions for each Instruction Information Block.

**Input files**
def68k-iibs.h

**Output files**
cpu68k-0.c
cpu68k-1.c
...
cpu68k-e.c
cpu68k-f.c

**Description**
This program takes the IIBs that were generated by the def68k program and creates the functions necessary to implementation each instruction's behaviour.

Rather than creating one big source file this program creates 16, the cpu68k-0.c output file corresponds to all instructions of the form $0xxx, the cpu68k-1.c output file corresponds to all instructions of the form $1xxx and so on. This makes compiling easier and reduces memory usage.

Each function generated has the following features:

- The source effective address and/or effective address value are generated.
- The destination effective address and/or effective address value are generated, if applicable.
- The output data is calculated using the above values.
- The output data is stored in the destination effective address location.
- The PC is updated to point at the next instruction.
- Any processor flags are updated.

## *6.6 68000 Processor emulation*

**Source files**
cpu68k.c

**Description**
At this stage of the implementation all the functions to implement each instruction have been created and during initialisation tables will be constructed using the IIBs so that we can go instantly from a 16-bit instruction to an IIB or to a function.

The CPU maintains a virtual state of the 68000 processor including:

- The currently location of the Program Counter.
- The state of the status flags.
- The 8 data registers and 8 address registers.
- The number of clocks since the last event.
- The next event (e.g. interrupt).

The source for the nitty gritty of the emulator is not worth showing here, however pseudo-code for the emulation is shown below:

```
main() {
  while(true) {
    load the word at the PC
    fetch from our table the function to call for this word
    call the function
    update the number of clocks we have emulated
    if (event) {
      generate interrupt if necessary
    }
  }
}
```

The loop is very tight and is designed to be as fast as possible. Several improvements were made during the development to increase the speed beyond even this.

Although we mentioned this in the previous chapter with reference to dynamic recomplication, it is actually possible to use the block marking techniques to our advantage with an interpretive system.

The operation of working out what flags are redundantly calculated might seem not to be useful to the interpreter - there is no way to alter the pre-compiled functions that handle each instruction. We could, memory permitting, create several versions of each function, one for each permutation of flag-setting requirements - however, with 5 flags this would result in 32 different varients of each function.

A compromise was reached. We would have just 2 variants of each function - one that

calculates all the flags required and one that calculates none of them. This is actually quite a good idea as most often this is exactly what will happen, we have require no flags to be created for any instruction in the block, apart from the last one.

The function generating program, *gen68k*, was altered so that it outputs two different versions of each implementation: one that performs flag calculations and one that doesn't. As such our 65536-entry table actually turns out to be twice as big, but still quite manageable.

### Instruction Parameter Cache

The idea of an Instruction Parameter Cache list came about because we need to cache the information about which instructions will require the flag-processing function and which instructions can be emulated without flag-processing.

Each instruction in the block has a *t_ipc* structure filled in. The definition of this structure is:

```
typedef struct _t_ipc {
  void (*function)(struct _t_ipc *ipc);
  uint8 used;               /* bitmap of XNZVC flags inspected */
  uint8 set;                /* bitmap of XNZVC flags altered */
  uint16 opcode;            /* 16-bit instruction corresponding to this IPC */
  uint16 wordlen;           /* the length in words of this instruction */
  uint32 src;               /* any source data cached */
  uint32 dst;               /* any destination data cached */
} t_ipc;
```

The *used* and *set* fields are for the redundant flag optimisation previously discussed. The other fields are all present in order to speed up the emulation:

The *function* pointer is a pointer to the function that will emulate this instruction. It is no longer true that the opcode entry can be used alone to index into our function table as there are now two different variants of implementation for each instruction opcode. It is a better idea to work out which function we need to call during the construction of the IPC rather than when we want to call it, for speed. Every optimisation counts.

The *opcode*, *src* and *dst* entries are all easily found through other methods, but they are here because the 68000 processor is a big-endian machine and some target processors will be little-endian. What this means is that on the 68000, the first byte of a 32-bit word is the most significant part, whereas on other machines, such as x86 PCs, the first byte of a 32-bit word is the least significant. If at all possible we would like to cache the correct endianness, for the target processor, so that the conversion occurs only once, when the IPC is constructed.

Another optimisation comes when we are able to detect that a source operand for an instruction will be fetched from ROM, always. There is no reason to do the fetching every time, since ROM cannot change, so the *src* cache can be used here too, in order to store the actual value.

The *wordlen* entry is present so that when we wish to dynamically recompile this block we are able to quickly identify how much to increment the PC by after each instruction. For normal interpretive emulation this is not necessary as each function updates the PC itself (this too is for performance reasons as it enables functions to add an integer constant that requires no memory fetching - to add the value in this structure would require a memory access to read it).

The IPC lists could have been stored in a standard linked list and searched linearly to find out if there is an IPC list for the current PC, however, this would have been inefficient. Instead the lower bits of the PC are used as the index to a hash table that is big enough that there should be very few occurrences at each location. The implementation was found to work well with a 16k table which means that the bits corresponding to $xxx7FE of the PC are used to reference a linked list. The chances of there being multiple blocks starting at the same offset in each 2k page are slim, to say the least.

## Debugging

Marking blocks, performing optimisations and executing blocks rather than individual instructions makes this interpretive emulator very fast. The downside is that it makes debugging difficult. As such there are different methods of executing the code. The fastest will try and perform every optimisation known to it, but the slowest will use the safest option. Tests have shown that the different in speed between the two is about 10x as slow.

## Busy-wait loop

Most games do not require the available number of CPU instructions in order to update the state of the game between each frame. It became apparent during development that games go into a loop that checks a location in RAM and jumps back to the checking instruction if it is not a particular value - without doing anything inbetween. Of course, this loop would be unhelpful if there were not interrupts. An interrupt forces the PC to a particular sub-routine which sets the location in RAM so that the foreground routine then knows that the next frame is due.

Obviously, for an emulator, going around in a loop executing instructions that don't need to be executed is not a good thing to do. As such, a further optimisation was added that involves checking for blocks that are 2 instructions in length and that just loop around. If this occurs we will mark the block as a special block that, when executed and it does indeed loop, it will also force the next interrupt to happen. The affect this had was startling and often made the difference between a playable and unplayable game.

## Final procedure

The 68000 interpretive emulation therefore proceeds as follows:

- Use the current PC to lookup in the hash table the linked list of IPC blocks
- If the current PC does not have a corresponding IPC block entry then:
  Create IPC block
  Eliminate redundant flag calculations and get function pointer
  Check for special busy-wait looping block and flag if appropriate
- Execute instructions listed in IPC block:
  Call function
  Loop around IPC list until all instructions emulated

Once the block has been emulated the new PC is looked up in the same manner.

## Functions

The major functions provided by this module are:

**cpu68k_init**
This is called when the program starts up and initialises the virtual machine including:
- Building the Instruction to IIB table
- Building the Instruction to Function table
- Clearing the Data and Address register states
- Building support structures for interpretive routines

**cpu68k_ipc**
This is called internally during the execution cycle - it takes an IIB and a location of where the instruction came from within the 68000 address space and fills in an IPC structure.

**cpu68k_makeipclist**
Given a 68000 address this uses cpu68k_ipc to build an IPC list containing IPC information for the block starting at the given location.

**cpu68k_autovector**
This is how an interrupt is actioned. This takes as a parameter the autovector number which indicates where to fetch the location of the interrupt routine from. This updates the processor level, moves to supervisor mode and updates the PC and status flag registers.

**cpu68k_step**
This simple executes a single instruction and does not use any cached information. It looks up the correct IIB using the Instruction to IIB table. It then calls cpu68k_ipc to create an IPC entry and looks up the function to call using the Internet to Function table. It then passes the IPC to

this function so that it can emulate it. Once this has returned the number of clocks emulated is updated using the clock information in the IIB and if appropriate causes an interrupt by calling cpu68k_autovector.

**cpu68k_framestep**
This does an entire frame's worth of cpu68k_step actions, but uses caching and, if possible, dynamic recomplication, to do it. It will not check for breakpoints set in the debugger and will use IIB/IPC entries to their maximum potential.

**cpu68k_reset**
This resets the 68000 CPU and returns it to the default state.

## *6.7 Video Display Processor*

**Source files**
vdp.c

**Description**
The Video Display Processor builds each line of the frame up individually in exactly the same was as the hardware had to do due to the television protocols. This was seen as appropriate as it enabled the implementation of our software emulation to follow closely what must happen in a line-based drawing processor.

This means that our implementation can support changes of the data after every scanline. It also correctly supports collision detection, overflows, and palette changes (platform dependencies permitting), all of which can only be accomplished using a line-based VDP emulation.

A complete run-down of features, implemented in this simulation:
- DMA copy between VRAM, VSRAM and CRAM.
- DMA copy of 68000 RAM to VRAM, VSRAM and CRAM.
- DMA fill of VRAM, VSRAM and CRAM.
- Background layer.
- Scroll layer A and scroll layer B.
- Sprite layer.
- Priorties on scroll and sprite layers.
- Colour - including hi-light and shadow facilities.
- Horizontal scrolling of screen at screen, cell and line resolutions.
- Vertical scrolling of screen at screen and column resolutions.
- Vertical and Horizontal flip.
- All sizes of sprite and scroll layers.

## Functions

The major functions provided by this module are:

**vdp_init**
This is called when the program starts up and is responsible for the initial reset of the system. It does not perform any once-only initialisation and just calls vdp_reset.

**vdp_reset**
Resets the state of the VDP, including:

- Clearing the 25 VDP registers.
- Setting the PAL/NTSC and language modes.
- Clearing flags such as collisions, overflows and interlace.
- Clearing the CRAM, VSRAM and VRAM memory buffers.

Plus it also calculates the number of clocks that the CPU should emulate between each line, so that the emulation is in accordance with the television protocol selected.

**vdp_status, vdp_storectrl, vdp_storedata, vdp_fetchdata**
Functions for returning the status and performing DMA and register setting. All for the memory manager to use when the game requests it through memory mapped i/o.

**vdp_sprites**
Used by vdp_renderline to draw the sprites component of a line in the current frame.

**vdp_layer**
Used by vdp_renderline to draw a scroll layer component of a line in the current frame.

**vdp_renderline**
This is called by the processor to render a line at the correct moment. It calls vdp_layer and vdp_sprites to build up the line in the correct manner outlined in section 3.3.

**vdp_showregs**
Displays the current status of the VDP system for debugging.

## *6.8 Memory Manager*

**Source files**
mem68k.c

**Description**
The implementation of the memory manager did not differ from that envisaged in the design stage.

A memory map structure was defined that would detail a set of functions that would be appropriate for fetching and storing data of different sizes for particular regions of the 68000 address space.

### Functions

The major functions provided by this module are:

**mem68k_init**
This function is called when the program is first started up. It goes through the memory map structure and builds the tables required by the system. The tables created:

- mem68k_memptr returns a pointer to the memory. This is used by the disassembler.
- mem68k_fetch_byte fetches a byte from the given location.
- mem68k_fetch_word fetches a 16-bit word from the given location.
- mem68k_fetch_long fetches a 32-bit long from the given location.
- mem68k_store_byte stores a byte to the given location
- mem68k_store_word stores a 16-bit word to the given location
- mem68k_store_long stores a 32-bit long to the given location

These tables are used in the 68000 emulation directly so as to perform the desired action as quickly as possible.

The other functions in this module are specific for dealing with the cases is the different memory areas. These memory areas are:

- 000 - 3FF ROM
- 400 - 9FF Invalid
- A00 - A01 Z80 RAM
- A04 Z80 Yamaha i/o
- A06 Z80 Bank register
- A10 System I/O
- A11 System CTRL
- C00 VDP
- FF0 - FFF RAM

The memory manager also has several public variables that can be used by the User Interface module to communicate keyboard actions:

- mem68k_cont1_a, mem68k_cont1_b, mem_cont1_c
- mem68k_cont1_left, mem68k_cont1_right, mem68k_cont1_up, mem68k_cont1_down
- mem68k_cont2_a, mem68k_cont2_b, mem_cont2_c
- mem68k_cont2_left, mem68k_cont2_right, mem68k_cont2_up, mem68k_cont2_down
- mem68k_cont1_start, mem68k_cont2_start

The variables start mem68k_cont and then indicate 1 or 2 for either the player 1 controller or the player 2 controller. Each controller has three buttons marked A, B and C and these correspond to the variables above. The left, right, up and down variables correspond to holding down one of the directional arrows on the controller. The start variables indicate holding the start button down on each controller.

## *6.9 User Interface*

**Source files**
ui.h

**Description**
This is a platform dependent source file that contains a set of specific functions that must be within any User Interface implementation. The functions required are given in *ui.h*, and are listed below for completeness:

**ui_init**
Called with the standard C argument arguments.

**ui_setinfo**
Called in response to loading a Sega Genesis image file. This gives the UI chance to display to the user the name and copyright of the image.

**ui_loop**
Called after initialisation and passes control to the User Interface module.

**ui_endframe**
The 68000 emulation module will call this so that the User Interface module can display the latest complete frame to the user.

**ui_run**
Called after ui_setinfo in response to loading a Sega Genesis image file. The usual action for the User Interface module is to call cpu68k_framestep() repeatedly, whilst checking for keyboard input etc. Keyboard input should be stored in the public controller input variables.

**ui_final**
If the program must quit due to some external stimulus (e.g. a signal on UNIX) then this gives the User Interface a chance to closedown cleanly.

## 6.10 Tcl/Tk User Interface

**Source files**
ui-tcltk.c
gen.tcl

**Description**
This file implements the User Interface header file. Tcl allows us to create the desired motif-style look, as can be seen from the screen shot below.


Figure 6.2 - The main window

In order to understand what features are available we shall proceed through the menus shown above.

**File menu**
The file menu contains the two options: Load image and Quit. The image loaded is shown below and was chosen out of the available options. It was intended to look similar to the Netscape loader in figure 5.2.

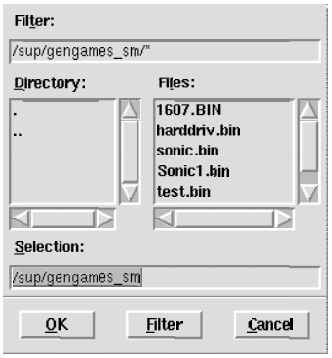


Figure 6.3 - The image loader

The loader can accept the two standard file formats for Genesis games, those ending in .bin and those ending in .smd.

**Open menu**
This menu only contains a single option - to open the Info window. This presents information on the current image that is running:

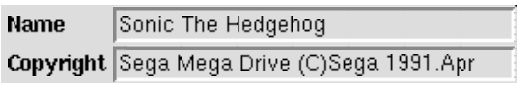


Figure 6.4 - The Information window

**Display menu**
This menu allows the user to select what the display looks like - some functions control elements of the User Interface module and some involve communication with the Video Display Processor to alter its parameters.

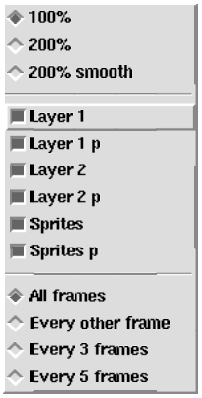


Figure 6.5 - The Display menu options

The first section lets the user enlarge the main window so that it scales the emulation up. There is also a 200% smooth option which attempts to interpolate and enhance the image.

The section section allows the user to turn off the plotting of various layers at different priorities. This can speed up the game when the target processor is having difficulty in keeping up.

Lastly there is a frame-skip option so that instead of plotting every frame, we reduce the frame-rate and only plot every other frame. This is quite effective as much of the speed is lost through the creation of the image and the transfer through X to the screen.

**CPU menu**
The CPU menu configures what %age of a real 8Mhz 68000 should be emulated. For example setting the CPU to 50% makes it act like a 4Mhz processor.

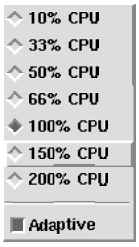


Figure 6.6 - The CPU menu settings

The Adaptive function, which has been previously mentioned, tries to work out when the busy-wait block occurs and when it does, it forces an interrupt which is effectively the same as reducing the number of CPU cycles.

**Debug menu**

The debug facilities are quite extensive and certainly helped ensure the project was finished on time.

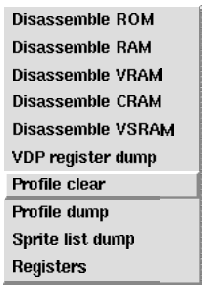


Figure 6.7 - The Debug menu

The debug option on the menu-bar is configured in the Tcl/Tk script to vanish when the project is compiled in non-debug mode. However, with debug turned on, instead of running an image when it is opened from the loader window, it is suspended and the Registers window is used instead:


Figure 6.8 - The Register window

To start executing an image as fast as possible the Frame Step button is clicked on. However, single stepping and continue-until-breakpoint can be performed from here too. Pressing Ctrl-C will stop the current execution and display an updated register dump in the window. Clicking on a register value will open up a disassembly window, as shown below:

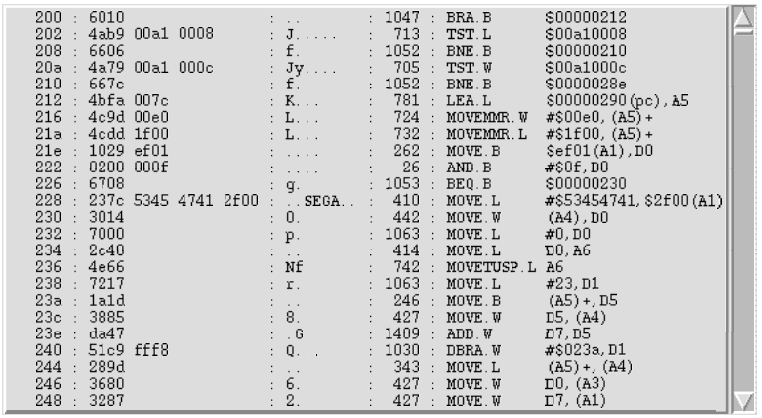


Figure 6.9 - An example disassembly window

The disassembly window indicates what address each instruction is at, the data that it comprises, the function number that it is emulated using and an assembly string. All this information allows for easy debugging of any problems that might occur.

The other debug facilities output text to the console, but include valuable tools such as viewing the VDP registers and profile analysis to see which instructions are called the most - or, which instructions might be causing a problem at a given point in a game.

## *6.11 Main program*

**Source files**
generator.c

**Description**
This is where the program contains the main() function and were execution starts. It initialises all the other modules and contains the loading routines for the image files. It is a small file that is platform dependent, although most parts could be used on most platforms.

## 6.12 ARM recompiler

**Source files**
compile-arm.c

**Description**
The recompiler was not implemented until the interpreting 68000 emulation was tried and tested. The implementation strategy used was to create a callable function that would nominally branch to the existing C interpretive functions. In this way the recompiler can implement small parts of the 68000 instruction space instead of everything all at once. This allowed for a much better development environment as it was clear what was going wrong.

A simple algorithm was used to allocate the registers required - an array of registers was kept that contained a description of the contents of each register, e.g. what 68000 Data or Address register it contained, plus an indication of how long ago it was last used. In this way we intelligently keep registers that are often used and therefore reduce memory loads and stores.

This file is very complicated and it is suggested that you view appendix *** for the source code to it.

## 6.13 Other work

It was found that the C compilers used during this project produced some very good code - the only disadvantage that the interpretive functions had over dynamic recompilation was that information used in one function could not be passed on to the next, as there was no way to know in the case of the interpreter what function would be called next - at least, not to the C compiler at compile time.

The source file reg68k.c contains a wrapper that tries, and does succeed, to do away with this advantage to some extent. It attempts to allocate target processor registers to C variables. We allocate 3 registers, two for the different types of 68000 register banks and one for the Program Counter. This allows us to compile the interpretive functions assuming that these three things are placed in a particular target processor register.

At the end of every instruction we must increment the PC. Without these global registers this would mean a load instruction, an add instruction, and a store instruction. Now, we just have an add instruction.

The speed increase was definitely noticeable, especially on the Acorn platform where the it was found that the dynamic recompiler did not have a quantifiable effect on performance after this modification was made.

# Conclusions

The project has been demonstrated, and it works. The aims that were initially proposed, and that are detailed in section 1.4 have definitely been accomplished. Portability has been demonstrated, speed can be seen in the playability of the games and accuracy is self evident.

## *7.1 Evaluation*

There is no better way to evaluate an emulator than by playing it. Although it would be untrue to say every game works, it is fair to say that the majority do. The following screen shots show a selection of the games in the order of difficulty.

**Test Image**

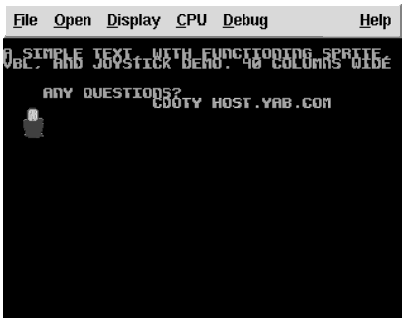


Figure 7.1 - Test Image screenshot

This is the public domain image that was used to test the most basic functions of the system. It was the first target to accomplish and uses a small number of facilities provided by the Genesis. The number of 68000 CPU instructions it uses are also small and it was easy to identify each routine and what was supposed to be going on.

### Hard Drivin'

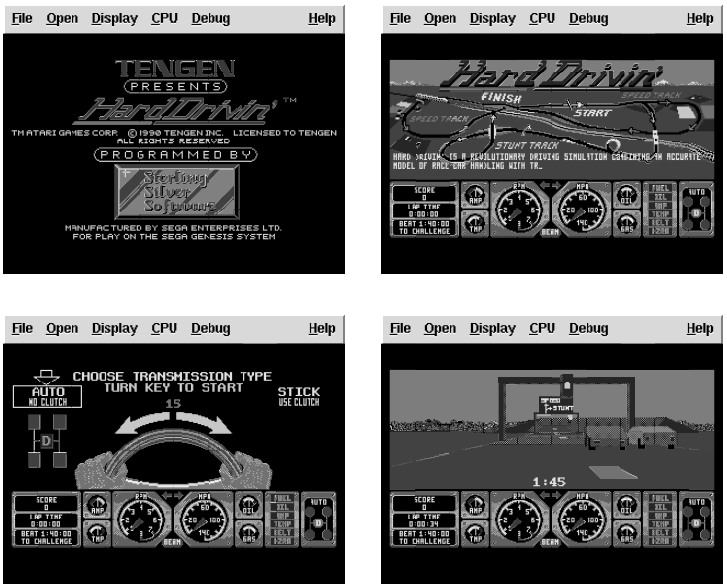


Figure 7.2 - Hard Drivin' screenshots

This game was recommended to me by the author of GenEm, the original PC emulator. The reason for this is because it does not require any DMA from the Video Processor, and was hence an ideal image to test out the other functions of the system.

### Sonic The Hedgehog


Figure 7.3 - Sonic The Hedgehog screenshots.

This game is the most famous of all games for the Sega Genesis and was my personal reason for wanting to do this project. Sonic The Hedgehog is a cute character, of course, but the game play, graphics and levels in this most classic of games is something to be cherished. It uses

virtually every function of the Sega Genesis and as you can see for the most part it is emulated correctly. Unfortunately there a couple of glitches. The top-left screenshot in figure 7.3 shows the bottom of Sonic above the logo, which is incorrect. Interestingly this error is found on all the PC emulators and hence leads me to believe this is not an error on my part but some misinterpretation of the available documents or undocumented feature of the VDP.

**Sonic The Hedgehog III**

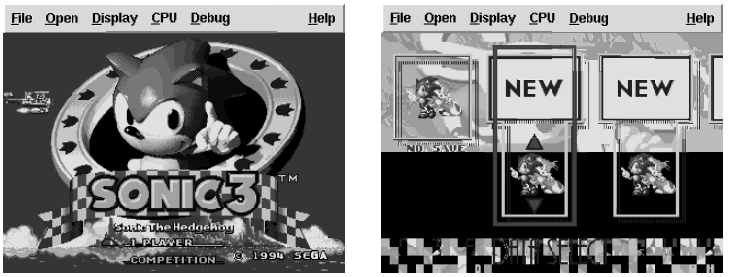


Figure 7.4 - Sonic The Hedgehog III

Sonic was so popular that he had dozens of sequels. Sonic III is shown here because there are errors that I think are due to the execution of code in RAM that is not detected quickly enough to invalidate the IPC lists for the RAM area. As you can see, the right hand screenshot is corrupt, and the actual game will not run.

**Ecco the Dolphin II**


Figure 7.5 - Ecco the Dolphin II

This shows another game in a completely different style. It actually emulates quite well, the only problem is when Ecco, the lead character, hits the water on his way down from a jump. This results him stopping instead of smoothly gliding into the water. This is probably due to an

arithmetic error, possibly on one of the CPU flags. The problem is that this sort of error is very difficult to trace.

## *7.2 Evaluation of Aims*

**Platform Independence**
This project is very portable, it can be transferred between any UNIX machine without any modifications and can be moved to others with only a few hours work. This is quite impressive given the low-levelness of this project and the amount of code involved.

**Accuracy**
This project, by the very ability to play the games, has to be quite accurate. The only problem is the lack of specific information on the Genesis hardware that makes it quite difficult to achieve exactly 100% accurate emulation. It is hoped that after a longer duration of development time that this can be solved.

**Speed**
Although difficult to impress on paper, the project is remarkably fast. It has certainly exceeded my expectation. It is fast enough to play games with, that much is certain - it actually achieves about full speed on a SPARC ultra 1 and on my home Pentium 100. The problem is that there are always those out there who are less fortunate and only have access to lesser machines. It is with this in mind that there is always a goal to make things faster.

## *7.3 Future Extensions*

The only thing missing in this project that would not be improved upon as a matter of course over time is the emulation of Sound.

Sound on the Genesis is particularly hard in that it requires the implementation of three separate devices. A Z80 processor, the Programmable Sound Generator and the Yamaha Frequency Modulation processor all need to be written and time-sliced with the 68000 processor. This is not an easy task and without further information it would be difficult to even speculate how this could be done.

Having described the difficulties I still feel that it would be a good thing to attempt and may do so in the near future after the minor problems with the emulation is sorted out.

**CPU Checking**
One thing that I would like to do is check that my 68000 CPU is completely accurate. There are interpretive emulator engines out there that, although they might be quite slow compared to this

project's emulation, are stable and reliable. It shouldn't be too difficult to integrate the two together so that the output of one 68000 emulator can be checked against the output from another.

**Dynamic Recompiling**
I did not feel after having written the ARM recompiling engine that it would be worth continuing this avenue of development. Unless the compiler is on the same scale as this project it will probably be more effort than it is worth and not produce the performance benefit that would require such programming effort.

## *7.4 Closing note*

I am very glad that this project was written, the following things have been learnt:

- Programming on X
- Programming using Xlib
- Programming using Tcl/Tk
- The Imake utility
- The Autoconf utility
- Programming a large C application

As such this project as been considered very worthwhile and I hope to continue to develop it in the future so that it can be released to the general public - everyone deserves to be able to play these games, whatever platform they use.

# Acknowledgements

This project could not have been accomplished without the support of my peers and the suggestions and helpful attitude of my supervisor, Sara Kalvala.

I wish to thank Sega for producing a thoroughly addictive game, Sonic the Hedgehog, which game me hours of entertainment - both in my childhood and now, via this project's emulation.

# References

Almost all sources are from the Internet, without it this project could never have got off the ground.


1 *Newsgroups*,
rec.games.video.classic, alt.emulators.game-consoles, alt.sega.genesis.

2 C. Doty, *GameSite Genesis Section*
http://www.doitnow.com/~cdoty/gamesite/

3 Unknown, *segadoc.zip*
http://www.classicgaming.com/EPR/genesis.htm

4 S. Williams, *Programming the 68000*
Sybex, Inc 1990

5 M. Gietzen, *GenEm*
http://www.classicgaming.com/emunews/genem019.zip

6 S. Snake, *KGen*
http://www.toodarkpark.demon.co.uk/

7 Unknown, *Genecyst*
http://www2.southwind.net/~bldlust/genecyst.html

8 Ardi (Abacus Research and Development, Inc.), *Executor*
http://www.ardi.com/

9 Bryce Cogswell, *TIBBIT*
http://www.cs.uoregon.edu/~cogswell/tibbit/

10 Unknown, *68k-simulator*
ftp://sunsite.unc.edu/pub/Linux/system/Emulators/

11 Zijian Huang, *EMU68*
ftp://aidan.ncl.ac.uk/pub/local/n4521661/emu/

# Bibliography

S. Williams, *Programming the 68000*, Sybex Inc.

A. Nye, *Xlib Programming Manual*, O'Reilly & Associates, Inc.

D. Young, *Programming and applications with Xt*, Prentice Hall 1992

A. Nye, *X protocol Reference Manual*, O'Reilly & Associates, Inc. 1995

F. Culwin, *An X / Motif programmer's primer*, Prentice Hall 1994

J. Ousterhour, *Tcl and the Tk Toolkit*, Addison Wesley.

B. Welch, *Practical Programming in Tcl and Tk Second Edition*, Prentice Hall PTR 1997

P. DuBois, *Software Portability with Imake*, O'Reilly & Associates, Inc. 1993

S. Harbison, *C: a reference manual*, Prentice Hall 1991

Aho, Sethi & Ullman, *Compilers: Principles, techniques and tools*, Addison-Wesley 1986

H. Hubert, *Electronic Music Synthesis*, J M Dent & Sons 1975