A Modular Soft Processor Core in VHDL

Jack Whitham

2002-2003

This is a Third Year project submitted for the degree of MEng in the Department of Computer Science at the University of York. The project will attempt to demonstrate that a modular soft processor core can be designed and implemented on an FPGA, and that the core can be optimised to run a particular embedded application using a minimal amount of FPGA space.

The word count of this project (as counted by the Unix `wc` command after `detex` was run on the LaTeX source) is 33647 words. This includes all text in the main report and Appendices A, B and C. Excluding source code, the project is 70 pages in length.

Contents

Part I.
Introduction

The aim of this project is to demonstrate that a modular soft processor core can be produced. This core will provide some or all of the features of a real processor: but it will be possible to leave out features that are not needed for a particular application, due to the modular nature of the design. In this way, the size of the processor can be minimised. Entire computer systems could potentially be built on a single chip, if each part was small enough. This "system-on-a-chip" approach could reduce the cost, physical size and electrical power requirements of an embedded system.

No fully modular processor has been built to date. The typical processor is either monolithic, with no modularity whatsoever, or very simple modularity (for example, a version may be produced with a floating-point unit on board). Until large-scale Field Programmable Gate Array (FPGA) technology became available, it was not feasible to make a processor that was optimised for one particular application, because any changes to the application would require a new processor, and this meant that a new piece of silicon would be required.

But now large FPGAs are available, it is possible to build a processor on one: The logical functions on the FPGA can be easily reconfigured by software, so it is quite easy to prototype all sorts of different processor designs at no cost. An FPGA makes an ideal test environment for a modular processor optimised for a particular application. To date, however, no soft processor core has been modular: all have provided a fixed set of features.

This project aims to demonstrate that a modular processor can be built on an FPGA, and that it can be optimised to run a particular application by leaving out the parts that are not required. There is no requirement to build all the modules that would make up a complete processor, nor run all applications, but the project may form the groundwork for later projects which do this.

1. Background and Literature

1.1. Soft Processor Cores

A soft processor core is effectively the working part of a CPU (central processing unit), described entirely by some Hardware Definition Language (HDL), and placed on a Field Programmable Gate Array (FPGA). It does the same job as a traditional "hard" processor, but it is implemented on an FPGA, instead of being implemented directly onto non-reconfigurable silicon. T80 [Wallner 2002] and MyRisc [Wallander 1998] are two other soft processors which were looked at during the project.

The workings of a processor are well described in "Computer Architecture: A Quantitative Approach" [Hennessy 1996], which discusses the design principles involved in building a processor.

1.2. A Field Programmable Gate Array

An FPGA is an integrated circuit (IC) that can be programmed to carry out any logical function. FPGAs have a huge number of gates (sometimes millions) on board, and these gates can be interconnected in any configuration necessary to simulate a logic circuit. Interconnections are made entirely by software: a "synthesised" hardware definition for a logic circuit can be uploaded to an FPGA, and the FPGA will then take on the features of that logic circuit. The logic circuit is described by a hardware definition language (HDL).

An FPGA consists of a matrix of "Logic Cells". Each cell on the FPGA that is available for the project (a Xilinx Spartan-IIE XC2S300E) has a 4-input lookup table that can act as a logical function generator, a RAM, or shift register. Each cell also contains a D-type flip-flop. The internal layout of the FPGA is illustrated in Figure 1. On the left, the entire FPGA is shown: each "CLB" - Configurable Logic Block - contains four logic cells. On the right, the internals of two logic cells are shown.

Using these, the Spartan-IIE can can represent up to 300,000 logic gates. The documentation for the device is in [Xilinx 2002]. The development board with the FPGA on it is shown in Figure 2.

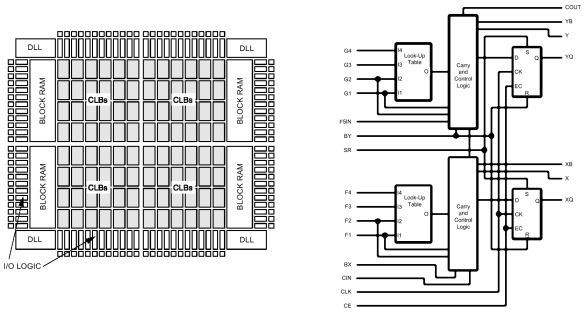


Figure 1: The internals of an FPGA, from [Xilinx 2002]

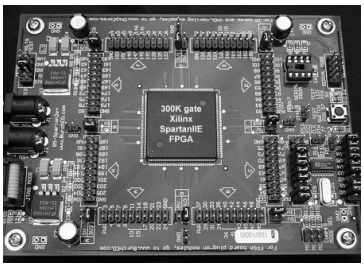


Figure 2: The "BurchEd 5" Spartan II-E board available for the project.

1.3. VHSIC Hardware Definition Language (VHDL)

VHDL is one of two hardware definition languages supported by the Xilinx Synthesis Tool (XST), which is a program that constructs ("synthesises") FPGA hardware definitions from a HDL. XST also supports Verilog, and synthesis from circuit diagrams. XST is analogous to a compiler: just as a compiler translates source code in a language such as C into low-level machine code, XST translates source code into the logical functions that will implement it. These are then translated into FPGA programming instructions: a bit pattern that is downloaded to the FPGA.

VHDL is not like a software language. Expressions in VHDL are essentially descriptions of logic devices, and variables (known as "signals") represent links between devices or registers. VHDL statements usually execute in parallel: whereas statements written in a software language are almost always executed sequentially.

VHDL and Verilog have the same capabilities. But VHDL's similarity to the programming language Ada made it the clear choice for the implementation of this project. As the author was already familiar with Ada, it was hoped that VHDL would be easier to learn than Verilog, and it would thus be possible to get started with the project sooner. The author could also have attempted to design the processor as a circuit diagram. But this would have been so low level that much more work would be required to build a working processor.

A guide to learning VHDL by Ashenden [Ashenden 1998] was studied. It was also possible to learn some VHDL techniques by examining code from other projects. As is described on Page 13.2, the design of the processor's ALU comes from the T80 soft processor core.

1.4. The Motorola 68020

As will be discussed later, it was decided to base the project around the Motorola 68020 processor. The manual for the processor [Motorola 1985] was obtained. It describes everything that a low level programmer would need to know in order to use the processor: the entire instruction set is described in detail along with plenty of information about the other features of the processor. To give a better understanding of the decisions made in designing the 68020, a paper by one of the designers of the 68000 was also read [Tredennick 1988].

Part II.
High-level Project Decisions

2. Should the design be based on an existing one?

As stated earlier, the project aims to produce a modular soft processor core. The first decision facing the designer is whether the processor should be an entirely new design, or based on an existing design. Here, the answer is clear. The soft processor core should be based on an existing processor, so that existing development tools can be used to develop for it. Lots of technical information will be available for an existing design, and the processor will be more compatible with existing software and familiar to programmers who have worked with the original processor - making it more acceptable if it were to be reused in other projects.

This choice saves a lot of time, because as well as producing all the required development tools (a compiler and debugger as a minimum), the development of an entirely new processor would require extensive research into the best design. Designing an entirely new processor requires a lot of work to determine the optimum instruction set and internal layout.

3. Which processor should the soft core be based upon?

The designer must now choose a processor to base the project around. The aim of the project is to demonstrate modularity, so the processor needs to have the characteristic that many of its features are not always needed.

The processor must also be quite simple, so it is feasible (within the project time-scale) to build a working version of the processor that is able to run some applications. This might not be possible if the processor was overly complicated.

4. Which processor should be chosen?

Many types of processor are suitable for this project. The Intel 80386, the ARM processor, the Motorola 68020, SPARC and MIPS are all powerful 32 bit processors that have some modularity, and are all very well documented and understood. The choice between them is really only based on which has the greatest modularity, and which will be the least difficult to implement.

The RISC (Restricted Instruction Set Computing) processors are not such a good choice for the project. The RISC processors listed above are the ARM, SPARC, and MIPS. These processors have a small instruction set, without complicated instructions such as division. Their instruction sets are designed so that a high level language compiler can use practically all the instructions. And this means that few instructions can be omitted. It will be difficult to demonstrate modularity with a RISC processor, even though implementation of the entire processor is simpler because there are fewer instructions. For this reason, it was decided not to attempt implementation of the ARM, SPARC or MIPS processors.

Equally, the 80386 is a poor choice for the project because it is too complex. The CISC (Complex Instruction Set Computing) architecture of the 80386 is very complicated since it had to be binary compatible with two 16 bit predecessors: the 8086 and 80286. The 80386 has hundreds of instructions, and an incredibly complicated system for instruction decoding. It would be very difficult to understand the whole architecture well enough to be able to implement any sort of subset.

This leaves the CISC Motorola 68020. It is based only on the 32 bit 68000 and makes no significant architectural extensions. This makes it a much cleaner, and therefore easy to understand, architecture. It also has only half the instructions of the 80386 and less addressing modes.

The 68020's instruction decoder is far simpler than the 80386 instruction decoder: in fact it has more in common with a typical RISC decoder. The instruction coding scheme is straightforward: all opcodes are 16 bits in width, and the fields in the instruction bits are usually in the same place. For example, register numbers

appear in only one of two places in the instruction - from bits 2 to 0, and from bits 11 to 9. This makes the design of an instruction decoder far easier.

5. Restating the aims of the project in terms of the chosen processor

The aim of this project is to demonstrate that a highly modular practical soft processor core can be produced. The processor will be a subset of the 68020, and able to run some, if not all, 68020 programs. The processor will be individually tailored to run a particular program: it will be generated in VHDL by putting modules together to support exactly the features needed by that program.

By the end of the project, it is hoped that a minimal processor may be built for an arbitrary program written in C. The processor will be as small as reasonably possible for a particular application.

Part III.
Modular Processor Design Decisions

6. Processor Design

In this section of the project, the research that was carried out into processor design is examined. It was decided that the only way to approach the project that would produce a useful result would be to implement an actual processor, capable of running Motorola 68020 code directly. This may seem an obvious choice, but there are a few alternatives which will be discussed briefly before an examination of the features that a processor would need to have.

6.1. Alternatives to a complete processor implementation

Software Interpreter

An alternative to implementing an actual processor was to implement an interpreter for the 68020 code. Interpreters are found in emulators - programs that allow one type of computer to run programs written for another type. A 68020 interpreter would translate each 68020 machine instruction to some other type of instruction, then execute them. This would have the same effect as a real 68020 processor. A processor of some sort would still be required to run the interpreter. But any type of processor could be used: any existing soft processor core would be suitable, as would a very simple new type of processor intended only to run the interpreter software.

The advantage to the interpreter approach is that it allows all of the 68020 instruction decoding and execution to take place in software. Software is very easy to write - easier than a hardware description. But, more than that, software is easy to change. It's easy to drop support for a particular instruction: the code to run it can be left out. There are many ways to do this. Perhaps the most well known would be the C preprocessor `#ifdef` directive, which allows blocks of code to be marked and included in compilation only if a particular label is `#define`d. So an interpreter could easily be optimised for a particular application.

The disadvantages of the interpreter approach are twofold. Firstly, interpreting a language is always slower than running it "natively" - that is, directly on the processor it was intended for, unless for some reason the interpreting processor is significantly faster than the native processor.

A second disadvantage concerns the physical size of the interpreter on the FPGA. There are a limited number of logic gates on an FPGA, and certain approaches to building the processor will use up more gates than others. An interpreted approach will certainly use up more gates than a native processor, because it must include both

a processor to run the interpreter software, and the software itself, probably stored in some ROM (read-only memory).

Although an interpreter approach would allow the project to be completed almost entirely in software, which would make implementation far easier, it seems unlikely that it would make a very good demonstration of modularity. It doesn't really matter if the processor is quite slow, but there will be a serious problem if it takes up most of the FPGA. Part of the aim of the project is to create a minimal processor - which implies the creation of a processor that takes up a minimal number of logic gates.

VHDL Interpreter

One way to make the software interpreter smaller would be to implement it directly in VHDL. VHDL does allow some degree of sequential programming, so it may be possible to implement an interpreter entirely in VHDL. Statements in a VHDL process are "run" in the order they appear: by including `wait` statements, execution can be stopped until an event of some type occurs.

This initially seemed like a good plan, since it is a mixture of a hardware and software approach. It was hoped that the synthesiser would examine the VHDL and work out the minimum hardware needed to run it. All the state machines, registers, adders and subtracters required would be inferred from the VHDL. So some research was carried out: would this be possible?

Unfortunately, it was found that XST does not handle sequential VHDL very well. It is only able to handle VHDL processes that run continuously, or run on a clock edge - it cannot handle the `wait for` or `wait until` statements in the general case. So this alternative is ruled out by the lack of support from XST.

6.2. A real processor

Because the alternatives were not really feasible, the only way to approach the project was to develop a complete soft processor core, with the ability to be modularised. Research was carried out into the features of a complete processor. All contain the following elements:

- Instruction Decoder
- Control Logic
- Arithmetic and Logic Unit (ALU)
- Register File
- Links between Components

The original 68020 is no exception: and all five features will be essential in any implementation of the 68020. In this section, the features are discussed with respect to the project.

6.3. Instruction Decoder and Control Logic

The task of the instruction decoder is to take a machine instruction and determine how the processor should execute that instruction. Machine instructions on the 68020 are all 16 bit words. These 16 bits are called an *opcode*, or operation code. They tell the processor what operation should be performed, and, if that operation is to be performed on some data, where that data is to be obtained from.

The job of the control logic is to manage the many control lines that connect to the processor's components. These lines route data around the processor, arrange for data to be fetched from memory, and control components such as the ALU and register file. Control logic typically takes one of two forms which will be discussed in this section.

A microcoded control unit

In CISC processors, the control logic is often "microcode". The processor is actually controlled by a small program, known as a microprogram, that exists in an internal ROM. Microcode allows very complex instructions to be implemented in a very small area of silicon.

The microcoded control unit shown in Figure 3 is a finite state machine that generates an output for each state: a "Moore machine". The state variable is called the μPC - microprogram counter: and it is stored in a register and updated to the next state on every clock cycle. The main component is the large ROM table. Each row of this table represents one state. Most of the data in the row goes to the processor control lines. However,

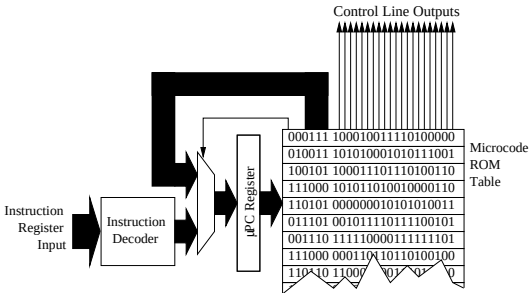


Figure 3: An example of a Microcoded Control Unit

some go back to select the next state. The multiplexer[1] allows the microcode machine to choose whether the next state number comes from the microcode itself, or from the instruction decoder.

Thus, the microcode machine can run through a whole sequence of control line settings, causing data to flow through the processor in an appropriate manner. This allows it to fetch and execute instructions.

There are two problems with the use of microcode to provide the control logic. One is caused by the complexity of the bit patterns. Some tool is needed to generate the table if the microcode is going to take on any realistic level of complexity. Several have been invented over the years, and these are known as HLMLs: high level microcode languages. And, unfortunately, the use of tools to generate microcode leads to sub-optimal sequences, because the possibilities for optimisation are hidden by the HLML. As [Tredennick 1988] states:

> People think of microcoding as programming with wide opcodes. [This] common approach *is* the reason microcoded implementations are slower.

A further difficulty with a microcoded approach is that the microcode will be kept separate from the VHDL by necessity (VHDL cannot easily contain such things), but the two must be kept synchronised. If components are added or changed, it might be necessary to update large sections of microcode. So some interface between the two that ensured the two stayed in synchronisation would be essential.

A hardwired control unit

An alternative type of control logic used in some processors is known as "hardwired". Here, the appropriate sequence of control line outputs is generated by a minimal set of logical functions, which take the output of the instruction decoder and the output of a sequence generator as their inputs. These logical functions are provided by discrete logic gates.

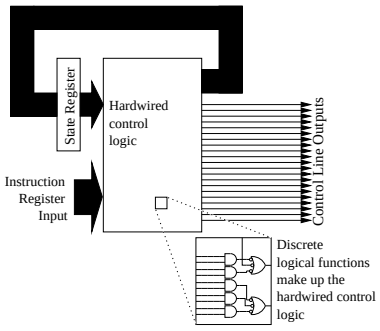


Figure 4: An example of a Hardwired Control Unit

The hardwired control unit shown in Figure 4 has the same function as the microcoded control unit in Figure 3. Here, however, the current state (stored in the state register) and the instruction register input are together

[1] The multiplexer is the the device between the instruction decoder and the μPC register. For more information about the function of a multiplexer, see Page 67.

used to calculate (through a series of combinatorial logic functions) the next state and the control line states. There is no ROM access, so the machine is as fast as the logic that it is built from. The unit is another Moore machine, but this time the next state is calculated, rather than found in a table.

There are two difficulties in taking this approach. The first is similar to a problem with microcode: how can the correct sequence of control outputs be defined? The second difficulty here is arranging for a minimum set of logic gates to be used in the hardwired control unit. Fortunately, this task is easily carried out by a computer: in fact, the ability is built in to the Xilinx Synthesis Tool (XST). The algorithm that is typically used is the Quine-McCluskey method for minimisation of boolean functions.

In a hardwired control unit, there are no lookups in slow ROM - the control information is available immediately. And the fact that the control line states are not written as what appears to be a sequential program encourages the designer to maximise the parallelism that is possible. However, the amount of silicon, or in this case the amount of FPGA cells, required to implement the hardwired control unit could be a limiting factor.

The best control unit for a particular processor

In a RISC processor, like the ARM or MIPS, many instructions are so simple that there is no need for a microprogram. Instead, the instruction decoder directly executes many instructions by setting control lines itself. Very little additional control logic is needed, so a hardwired control unit is quite feasible. This is found in the ARM series of processors.

Microcode is traditionally used to control CISC processors - particularly the 68020 and its contemporary, the 80386. The 68020 has around 85kbits of microcode in on-board ROM, according to [Tredennick 1988]. This microcode actually exists in two levels, which reduces the space required by 20%.

The fact that the designers of the 68020 took this approach would seem to suggest that this project should go in a similar direction, as it indicates that an implementation of the 68020 is likely to be too complex for a purely hardwired approach. However, as will be discussed later, a hardwired approach is actually far better due to the high level features of VHDL.

6.4. Arithmetic and Logic Unit (ALU)

As its name suggests, the ALU provides the processor with primitive arithmetic and logical operations. It is able to add two values (an arithmetic operation), or find the logical AND of two values (a logical operation). A typical ALU, and indeed the 68020 ALU, provides add, subtract, logical and, logical or, and exclusive-or (known as EOR) functions.

Some ALUs may also support multiplication and division directly, but it is quite common to implement these operations using repeated adds and shifts, since this requires less hardware. The 68020 uses the latter approach, as evidenced by the fact that such operations take a minimum of 25 clock cycles, compared to a maximum of 3 clock cycles for an addition [Motorola 1985].

The project's ALU should provide the same features as the real 68020 ALU. All of the features are likely to be used by any application: and none can easily be emulated in software. It may be, however, that some ALU features can be modularised and removed for programs that do not need them.

The ALU makes up the processor's "data path", or "execution unit", along with the register file.

6.5. Register File

The register file stores temporary data that the running program is using. Programs typically store as many of their variables in registers as possible. This is done to take advantage of the high speed of register access compared to memory access. The 68020 has sixteen general purpose registers, split into two groups of eight. One group is for data registers: these are typically used for storing operands for computations. The other group is for address registers. These are typically used for holding pointers to locations in memory. The registers are all 32 bits wide.

All of the registers in the register file can be accessed by the programmer. It is worth noting, however, that a typical processor will have some other registers that are not accessible externally. These temporary registers store values that are internal to the processor. For example, many processors have an instruction

register (conventionally abbreviated to IR) which stores the currently executing instruction so that all parts of the processor may refer to it. Programs cannot access IR directly.

The modular processor may be able to omit registers that are never used by the application being executed.

6.6. Links between Components

As may be expected, whenever data needs to move between two registers, there must be a link between them. Links usually run from register to register, or register to ALU, via multiplexers. The multiplexers allow a particular data source to be chosen as the input to a component. The designer wishes to minimise the number of links, because each link that exists requires extra logic to implement and thus takes up more space on the FPGA.

7. The framework for a minimal processor

In this section, the basic framework of a processor is examined. The processor produced by the project will, by necessity, be more complex than this, but it will follow the basic design: an instruction set processor with a load store architecture.

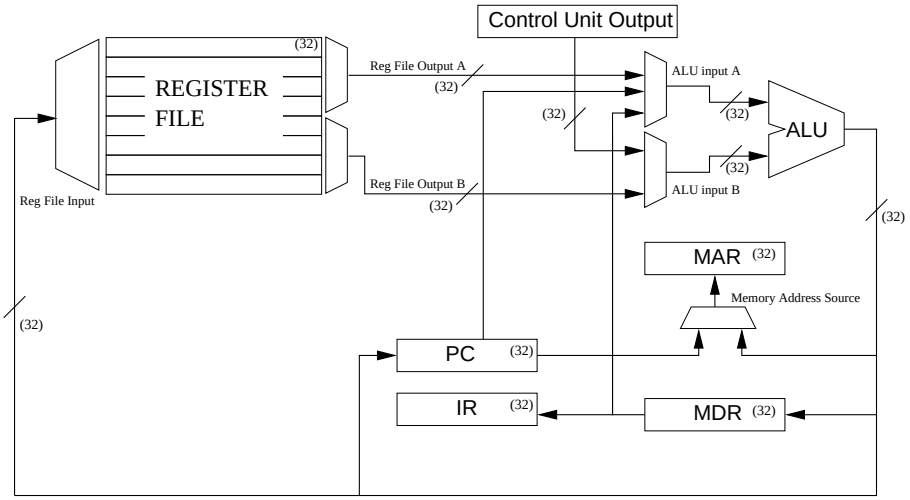


Figure 5: A minimal processor

Figure 5 illustrates the components of a minimal processor[2]. The control unit is not illustrated, because drawing it would also require the drawing of control lines to all registers, multiplexers and the ALU, and this would complicate the diagram unnecessarily.

The ALU is clearly shown, along with its links to the register file, the memory data register (MDR), the memory address register (MAR), and the program counter (PC). The instruction register (IR) is also shown. The arrows indicate the direction that data may flow in. New data is only loaded into a register when the control unit allows it to be.

It is a convention in processor design to represent the interface to the machine's memory using two special registers: MDR and MAR. When the processor wishes to read from memory, it loads the required address into MAR and (soon after) reads the data at that address from MDR. When the processor wishes to write to memory, it loads the data to be written into MDR and the address to be written to into MAR.

The reader may wonder why the PC register is separated from the others. It is separated so that it may be loaded directly into MAR. It is thus possible to fetch an instruction into IR and increment PC at the same time.

[2]Page 67 has more information on the symbols used in the diagram

7.1. How this allows an application to be executed

The model above implements what is known as a load-store architecture, as described in [Hennessy 1996], Chapter 2. The 68020 also has a load-store design, in common with most modern processors. Alternatives include the very simple accumulator architecture (which has only one general purpose register) and the stack architecture. These designs are not suitable for a 68020 clone: the basic architecture must be the same.

The minimal processor described in Figure 5 is, with the correct control logic, sufficient to execute any computer program. We can say this because:

It is able to fetch and execute instructions. Instructions are fetched from memory, by loading the PC value into MAR (there is a direct route) and then loading the MDR value into IR (again, there is a direct route).

It is able to branch conditionally and unconditionally. Instructions in a program are not necessarily executed in the order they appear in memory. Some cause execution to branch (jump) to another memory location. This is done by loading a new value into PC. On the 68020, branches of this type are done using instructions such as:

`BRA` n - The value n is added to PC: an unconditional branch

`BEQ` n - The value n is added to PC if the result of the last ALU operation was zero: a conditional branch

It is able to do arithmetic operations and make decisions based on the results. Since the ALU inputs can be connected to any two registers, and the output can be stored in any register, it is possible to add, subtract or apply a logical operation to any register pair.

The register values can be loaded and stored in memory (using the MDR and MAR registers), so these operations can be carried out on memory locations too.

The processor can make decisions based on the results of a computation by using its ability to branch conditionally on those results.

It is the ability to make decisions based on results that sets the computer apart from an adding machine. With just the above features, any program could be implemented assuming that sufficient memory existed to run it: this processor, like the processor in any computer, implements a Turing machine with finite memory. It is, therefore, at least as capable as all others of running any program (although it is not necessarily as fast!).

7.2. More complex features of the 68020

In this section, the more complex features of the 68020 are discussed. These features are not essential for a working copy of the processor, and many of them can be omitted or cut down in some way. These features include:

- Bus driving
- Pipelining
- Interrupts and Traps
- Memory cache
- Decimal Support
- Arithmetic and Logical Shifter
- Advanced Addressing Modes
- Coprocessor/Multiprocessor support

Bus driving

The 68020, like most processors, is intended to sit on a bus with other devices, such as RAM and ROM ICs.[3] A bus is a collection of (in this case) 32 data lines, with the property that more than one of the devices connected to the lines may write to them, but not at the same time. This allows two devices to exchange data using the

[3] RAM stands for random-access memory: this memory can be written to and read. ROM, on the other hand, is fixed when the system is built, and can only be read.

bus, so (for example) data can be read from and written to the RAM. The 68020 has some special hardware that controls the bus and arranges for data to be fetched and stored using it.

In this project, the designer is spared the difficulty of implementing bus driving logic, because the processor only needs to exist on an FPGA. All devices can easily be connected directly to the processor, so there is no need for a bus.

Since there is no advantage to simulating a bus on the FPGA, the bus driving features of the 68020 can be ignored. This will save a lot of development time, as those features are very complicated. 51 pages of the manual [Motorola 1985] are taken up by describing how to use the bus: and none of this complexity needs to be part of the project.

As a side note, an FPGA bus standard called Wishbone [Herveille 2002] does exist. The main purpose of this bus standard is to give FPGA components a standard interconnection interface. The advantage to making the project support Wishbone is that it would allow the soft processor core to be easily connected to the many other types of device that can exist on an FPGA, such as serial ports, video drivers, and timers. However, this is not the same as the 68020 bus standard, and there is certainly no requirement to implement it in order to get the processor to work. It should be thought of as a possible extension to the work.

Pipelining

The 68020 has a three stage instruction pipe. This allows it to execute some operations concurrently. Although pipelines allow a processor to run significantly faster, and are found in all modern processors as a result, they are difficult to implement. The pipeline implementor must be careful to avoid many hazards, and a lot more development and testing is required. As pipelining is just a feature to speed up the processor, it can be left out.

Interrupts and Traps

Interrupts and traps are another non-essential feature of the 68020: it is quite possible to demonstrate a working modular processor that doesn't support interrupts or traps.

Interrupts are generated when an external device wishes to get the attention of the processor. Traps, on the other hand, are generated by software: either when an error occurs (an attempt to divide by zero is a typical example), when a program wishes to make a system call, or when virtual memory paging is required. They are known as "software interrupts" on other architectures.

Since the project processor is intended for use in a small embedded system, there is no need for virtual memory or system calls. Other types of trap do not have to be supported either, since there are other ways of handling error conditions.

There is no need to support interrupts either. It is perfectly possible to build a computer system without them, since devices can be "polled" instead. Polling is a process of asking each device in turn if it has any new data. This is slower and tends to waste the processor's time. However, it is much easier than implementing a way to handle interrupts.

Memory Cache

The decision to leave out a memory cache is simple: there is no advantage to having one. As all the ROM and RAM can be on board the FPGA, access to all of it will take only one clock cycle. There is no advantage to using a cache unless access to memory is significantly slower.

Decimal Support

The 68020 has limited support for working with binary coded decimal numbers, through instructions such as ABCD (decimal add). These features are not used by many compilers - in fact, the well-known compiler GCC will never use them. They are not essential to the operation of the processor: all programs can work without them by converting to and from decimal format, and using normal binary arithmetic. As special hardware support is needed for these instructions, it is best if they are left out entirely.

Arithmetic and Logical Shifter

Arithmetic and logical shifts are not particularly complicated logical operations. A shift involves moving every

bit in a register to the left or to the right by a certain number of bits. Special hardware is needed to support this. On the 68020, it is possible to shift the contents of a register by up to 32 bits in a single clock cycle: the number of bits shifted doesn't affect the execution time. The shift hardware is therefore particularly complex.

Since a shifter is by no means an essential part of the processor (although it is a useful part), it will be omitted. Unlike the ALU, it is perfectly possible to demonstrate a working processor without a shifter. If a shifter is required for a particular application, it can be implemented as an extra feature at a later date.

Advanced Addressing Modes

The 68020 has 18 addressing modes, eight of them requiring additional hardware support (in the form of up to two temporary registers and a scaler, which multiplies an index register by a power of two). These modes are very powerful, and allow fast access to some very complicated data structures. Although it is true that a particular program is unlikely to use all of them, GCC is capable of generating code to use each of them in rare situations. Unfortunately, due to the extra hardware requirements, this complicates the processor design.

As a complete set of addressing modes is not required to demonstrate that the processor works correctly, all of the modes requiring additional hardware will be left out of this project. This will simplify implementation and testing of the processor, with the disadvantage that certain C programs will not run on the processor. Certain C data structures (specifically "struct") must be avoided so that the C compiler doesn't attempt to generate code using the unavailable modes. However, it will be quite possible to add support for these modes at a later date.

Coprocessor/Multiprocessor support

The 68020 has support for an optional coprocessor. Implementing this type of support is outside the scope of this project. The same is true of the 68020's support for multiprocessing.

8. Compiling and testing 68020 programs

It was realised at the beginning of the project that some way to build and test 68020 programs would be required. Since it was hoped that the processor would be able to run arbitrary C programs, a C compiler for the 68020 would be required. Additionally, some way to test those programs independently would be needed. If the programs could not be tested on some "reference" system, it would not be clear whether any problems that might occur were due to bugs in the program, or bugs in the processor itself.

An ideal build and test environment would be provided by a 68020-based computer system. Unfortunately, no such system was available for the project. The alternative available was a combination of a cross compiler and an emulator, allowing 68020 programs to be compiled and run on a PC.

The cross compiler chosen was GCC: the GNU Compiler Collection. This compiler could run on the Department's Linux PCs, and produce executable code for the 68020. There are, of course, some alternatives to GCC. The 68000 series of processors formed the basis for the Amiga, and original versions of the Apple Macintosh, and Unix workstations from Sun and Silicon Graphics. With such wide industry acceptance, it is no surprise that plenty of compilers for 68020 processors were built. Many, however, are commercial software and are not available for free. Out of the free compilers, GCC is by far the most developed. Since it forms the basis for successful free software such as the Linux operating system, a great deal of work has been put into its continued development, and the compiler it includes is cutting-edge. It is also well known and understood.

The first stage in building GCC on the Department's Linux systems was to build a cross assembler. So the GNU Assembler (gas) was built from the GNU binutils package, with support for assembling 68020 code. Once this had been built, GCC was built using it, producing the cross compiler.

8.1. GCC Compilation Issues

One problem that occurred during GCC compilation was a missing `crt0.o` file. Discussion of this problem with Department staff indicated that the function of this file is to provide a run-time setup for the program. When an operating system begins executing a program, it provides the program with information about the current execution environment and the program's parameters. The `crt0.o` file is highly operating system dependent: so

GCC doesn't provide a generic version. It is, however, required by the cross linker: any program that is produced must begin with the `crt0` preamble.

The solution that was found to this problem was to modify a sample `crt0.s` file from the source of the GNU C library, glibc. A cut-down version was produced that would just set up the stack and start execution. It can be seen in Section D.1. It takes out support for the environment and gives the stack pointer a fixed value.

Another issue was that a linker script was required. A linker script defines the memory locations that the program and its data will occupy. Since the intention is to compile programs for an embedded system, these locations are fixed. The memory map in the linker file must, however, match the one defined by the architecture.

A memory map describes what various memory locations are used for. It is part of the architecture: processors don't attempt to define the memory map. In a computer system, memory is traditionally divided between three things: RAM, ROM, and I/O devices. When a program accesses a memory address, the memory mapping hardware decides (based upon the address) where the data should come from or go to. An access to memory locations 0 through 1023 might load data from ROM, whereas an access to locations 1024 through 2047 might load data from RAM. Memory maps are not complicated things. Usually, they are implemented by allowing one or two bits of the address to select the memory device in use.

A simple memory map was decided upon, and can be seen in Table 1. It defines 4096 bytes (abbreviated to 4Kbytes) of RAM, and 4096 bytes of ROM. This is all quite arbitrary. It so happens, however, that there is more than enough room for this amount of RAM on the FPGA, in special-purpose memory cells. And 4096 bytes should be more than enough ROM to contain a small embedded application: certainly enough to demonstrate the processor's abilities.

The map was given to the linker script, `tiny.x`, which can be seen in Section D.2. The linker script was based on `m68kaout.x`, supplied with GNU binutils. The addresses of each segment were changed: the program memory starts at 0 (in ROM) and the initial stack address was set to `0x2000`[4] (the top of the RAM). The output device shown, at address `0x8000`, allows programs to send a single byte of output (for example, the result of some computation) to a display. The display is discussed further in Section 10.7.

Table 1: Memory Map for 68020 clone

Addresses between..	map to..	Linker segment	Used for:
`0x0000 - 0x07ff`	ROM	`.text`	The program
`0x0800 - 0x0eff`	ROM	`.data`	Constant data
`0x0f00 - 0x0fff`	ROM	`.other`	Other data
`0x1000 - 0x1fff`	RAM	`.bss`	Stack and global variables
`0x8000 - 0x8000`	I/O		Output device (display)

8.2. The Emulator

A 68020 emulator was needed to allow programs to be tested. An emulator is a program that allows code from one system to be run on another, by providing a virtual machine. An emulator was required that was freely available, supported 68020 instructions and could be made to use the memory map described in Table 1. So the emulator that is chosen must have a changeable architecture.

No freely available 68020 emulator was found, but two free 68000 emulators were found. 68000 processors are much the same as 68020 processors: but the 68020 has a few more instructions, and some instructions have been extended. Physically, the two processors are quite different: for instance, the 68000 has a 16 bit data bus, and the 68020 has a 32 bit data bus. For emulation purposes, only the instruction set differences are of any importance, and this difference is not necessarily a problem.

[4]The prefix "`0x`" indicates that a number is a hexadecimal value. This is the C convention, and it is used throughout this document. Hexadecimal values are in base 16, with letters a through f representing decimal values 10 through 15.

The first emulator, Generator[Ponder 2001], was found to have a fixed architecture - it would be very difficult to use it for testing 68020 programs. The second emulator, vm68k[Sasayama 2001], was a virtual machine for the 68000 in a library. It would be quite easy to build an architecture around it, since it comes with nothing more than support for the processor. A wrapper must be written to provide the architecture and the program it should execute. vm68k was written as part of an emulator for a 68000-based workstation, but it is fortunately well abstracted from the architecture of the workstation. It proved to be easy to use in this application.

Reading program binaries

One question arises from the emulator research: how are program files to be read in? Program files, or binaries, are not necessarily "flat" (unstructured) files. When they are generated by a linker, they are often generated with more than one segment. There are many different formats for these files: commonly used ones include COFF, ELF, and a.out. Typically, the different segments will go to different locations in memory. Sometimes, there will also be a symbol table, listing all the assembly labels that appeared in the program for debugging purposes.

Making a program to read one of these formats is not easy. Of course, the specifications are all available freely for all the formats supported by GCC, and reference implementations are included in some GCC programs, such as `objdump` (a disassembler). However, it was decided that the effort that would be put into writing the code for reading ELF or a.out would be better spent on more relevant parts of the project. So the very simple "Intel Hex" output format was chosen. Intel Hex is a format that is familiar to the author from earlier work, and it is so simple that a reader can be written in minutes.

9. What features can be modularised?

Since it is has now been decided which processor features will be implemented and which will be omitted, it is now important to decide which features of the processor will be modularised. Modularising a feature may mean that it can be omitted entirely (for example, support for a particular instruction might be removed) or that only part of it might be available (for example, the exclusive-OR feature might be left out of the ALU if the program didn't need it). It is a matter of optimising the processor to run a particular program. It is not a case of making each feature into a standalone module, such as a VHDL entity. The modules are not necessarily distinct from each other, so the process is really a matter of optimising a particular part, or set of parts, so that certain features are only enabled if necessary.

After some research and thought, the following features of the processor were thought to be capable of being modularised:-

ALU operations: Not all the ALU operations are necessary for every program. Although ADD is always required (for internal operations such as $PC \leftarrow PC + 2$), EOR, OR, AND and SUB may not always be needed.

Registers: Some programs will not use every register available in the processor's register file. Short assembly programs, in particular, will only use a few of the registers.

Addressing Mode Support: Some of the 68020's addressing modes are rarely useful, and as will be discussed later, this allows some addressing modes to be left out entirely.

Instruction Support: As is the case with addressing modes, the 68020 has many more instructions than would be needed in a typical program.

Since support for particular instructions can easily be left out of both the instruction decoder and the control unit state machine, there is a clear opportunity for modularity here.

Addressing Width: Suppose it is known that the highest address that the program will ever access is, say, `0x8000`. In this case, there is no need for more than 16 bits in address lines, buses and registers, because the 17th bit and all higher bits will always be zero. The processor implementation only needs to have support for addresses that will actually be used, so this part can also be optimised.

9.1. Modularisation of Instruction Support

Of all the modularisation tasks, the greatest improvement to the size of the processor will be gained by modularising support for machine instructions, and removing unnecessary ones.

The 68020 is a CISC processor. Compilers are notoriously poor at finding the best instructions to use when compiling a program for a CISC processor. Finding the optimum instruction is a very difficult search problem, so compilers tend to stick to a subset of available instructions. For instance, GCC will never use the 68020 instructions `ABCD`, `PACK` or `MOVEM`[5]. All of these do complex things that might be useful to a program, but the problem of working out how best to apply them is too complex for GCC, and indeed all but the most specialised compilers.

And even if GCC could generate most of the 68020 instructions, all of those instructions could never appear in the type of small program that can be run on an embedded system. The program would simply be too short to hold them all.

So typical programs won't use all 68020 instructions. Since the control store takes up much of the space on the 68020 die, it is certain that a large improvement would result from modularising support for each instruction, and leaving out the ones that are not required. Consequently, much of the project work will be directed into it.

The modularisation is achieved by *generating* the instruction decoder and control unit state machine - some program will be written that takes a 68020 program as input, and produces the minimal instruction decoder and control unit state machine to execute it. It is a matter of examining every opcode used by the 68020 program, working out which instructions are needed to provide the functionality, and then generating the VHDL to provide instruction decoding and control line sequencing for those opcodes. This would satisfy part of the main aim of the project by tailoring these components to the program.

9.2. Modularisation of Registers

Depending on the implementation of the register file, it may be possible to modularise each register, making each one removable. This could lead to a substantial saving in logic. Each register requires 32 flip flops to implement and also at least 32 data links running to and from it. Clearly, if this type of implementation is chosen, the fewer registers there are the better.

Unused registers would be detected by scanning the opcodes (in the program scanner, discussed in Section 11.3) and making a note of all registers that are used. Registers not in this set could be eliminated. This functionality could be built into the generator program for producing the instruction decoder and state machine.

9.3. Modularisation of ALU operations

As with the register file modularisation, implementation of this feature would be done by scanning the opcodes in the 68020 program. Each opcode would be examined to determine which ALU operations it would require. The set of required ALU operations would be built up and used to generate an ALU with support for those operations and no others.

9.4. Modularisation of addressing modes

The 68020 has 18 addressing modes, most of which are rarely used. [Hennessy 1996] has some interesting statistics on addressing mode usage in Chapter 2. Hennessy and Patterson evaluated three well-known programs (TeX, gcc and spice) on a VAX system[6], generating statistics on their use of a number of addressing modes. Figure 6 shows their results. As can be seen, addressing mode usage varies from program to program. Spice rarely uses the "Register Deferred" mode, and TeX never uses the "Scaled" addressing mode. This has a lot to do with

[5] This can be seen by looking at the part of GCC that generates 68020 code. A `grep` on the files involved, in the GCC source tree at `gcc/config/m68k`, indicates that these opcodes can never be produced by GCC from a high-level language.

[6] VAX systems are CISC machines that may be considered to be cousins of the 68000: they are both descendants of the PDP-11 minicomputer. The set of VAX addressing modes is very similar to the set of 68000 addressing modes.

the compiler used as well as the actual architecture of the programs, but the point that should be noted is that typical programs rarely, if ever, need all the addressing modes. This is especially true of short programs.

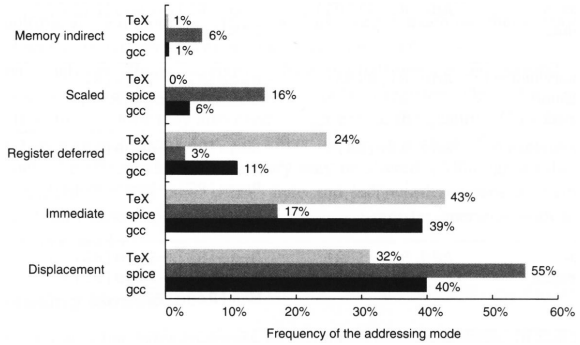


Figure 6: Usage of memory addressing modes in three programs (from [Hennessy 1996], page 76)

Each addressing mode could be modularised, allowing unused modes to be left out. This would be implemented, as before, by scanning the program's opcodes and making a note of all the addressing modes that are used. Leaving out an addressing mode would save the logic in the control unit that implemented the sequence for it.

9.5. Optimisation of the Addressing Width

The main problem that inhibits this optimisation is that the designer must find a way to work out what addresses will be used by a program. The data width of address handling logic must be enough to represent all addresses used by the program.

The range of addresses is known by the writer of the 68020 program, and something that is known when the memory map is decided, but not something that the opcode scanning program could work out.

It is not possible to infer the range of addresses that will be used by scanning through the opcodes. Finding this would generally require the program to be run, since not all the addresses that are used are given in the binary - some are calculated at runtime. The program, however, might run forever. In this case, there would be an infinite number of addresses computed. Although in specific cases it would be quite clear that all the addresses would be within specific bounds (a sequence would emerge), in the general case no such pattern could be seen. This is similar to the halting problem: it is impossible to tell, in general, whether a program will ever terminate.

So, in this case, the maximum address must be specified by the user in some other way. The generator would then produce all address-related components with the appropriate bit width.

9.6. Writing the generator

Writing the generator was quite an ambitious undertaking, and so the task was split into a number of implementation sub-tasks that could be completed individually:

- Writing the control unit generator (Section 10.1)
- Writing a minimal instruction decoder generator (Section 10.2)
- Adding any other components to the processor framework - ALU, etc. (Sections 10.3 to 10.7)
- Writing a program scanner to determine what opcodes are needed (Section 11.3)
- Writing sequencing instructions for opcodes (Section 12)

10. Designing processor components in VHDL

In this section, various methods for designing and implementing the processor components in VHDL are discussed.

10.1. Control Logic

VHDL makes the implementation of a hardwired control unit quite easy: some of its high-level features are ideally suited to this application. State machines are not difficult to write in VHDL, and control line assignments are trivial, because using VHDL removes the difficulty of working out the minimal discrete logic for a hardwired control unit.

In hardware, every control line is just a binary number: '0' or '1', clear or set. VHDL allows a higher level view to be taken. Control lines are given names (like variables in a software language). Although they can carry simple binary numbers, it is often useful to use "enumerated types". In these cases, the data takes one of a few preset values, each described by name. So the function of a component can be set without working at the binary level, making it easy to change. The designer can also easily see where each operation takes place, since they are described by name and not by bit pattern. Adding new functions to each component is easy. XST is able to check that these types are used correctly, so a line can never be set to an incorrect value. XST also decides how each value is mapped to a low level arrangement of bits.

Writing a state machine in VHDL

A state machine consists of a register (to contain the state number) and a decoder for the next state. In the case of a control unit, the state decoder will also produce control line outputs. These features can be seen in the sample VHDL state machine in Figure 7.

```
decoder : process ( state ) is              -- state is the state variable.
begin
    next <= "0000" ;
    ...                                     -- default control line assignments
    ...                                     -- are made here.
    case state is
    when "0000" =>  ...                     -- things that should happen in
                    ...                     -- state 0 are done here.
                    next <= "0001" ;        -- next state is state 1.

    when "0001" =>  ...                     -- things that should happen in
                    ...                     -- state 1 are done here.
                    if ( input = '1' ) then -- if input is 1, branch to
                       next <= "0010" ;     -- state 2, otherwise state 3.
                    else
                       next <= "0011" ;
                    end if ;
    ...
    end case ;
end process ;

state_machine_register : process ( next , clock ) is
begin
    if ( clock = '1' )          -- the state changes on a clock
    and ( clock'event ) then    -- edge. The next state is specified
        state <= next ;         -- by the state machine above.
    end if ;
end process ;
```

Figure 7: The General Form of a VHDL state machine

In Figure 7, an outline state machine is split into two processes. The `decoder` process decodes the current state into a set of control line assignments and a next state assignment. Each state is represented by a separate case in a large `case` statement. The `state_machine_register` process is a register, storing the state variable by assigning `state <= next` on each clock edge.

Figure 7 also shows an example of a branch within the state machine. In state "`0001`", an `if` statement examines some input and branches to different states according to that input. As can be seen, it is easy to put conditions like this into a VHDL state machine.

It's also easy to give control lines default values. If a state in the `case` statement does not assign anything to a particular control line, it takes a default value specified before the `case` statement began.

The use of a state machine like this in the control unit seems ideal. It is easy to assign control lines - it can be done by name. It is also easy to do conditional branches.

Each state is numbered, so states cannot generally be added or removed without renumbering. And it is not easy for the designer to remember which numbers apply to which states. One solution to this problem is to make the state variable an enumerated type, so each state is referenced by name.

A modular control unit

A modular control unit will be required for this project. Every machine instruction is executed by one or more state machine states. If all the instructions that require a particular state are never used, then that state will never be reached, and it can be eliminated.

Essentially, support for each type of 68020 instruction can be thought of as a module. All of the modules that are needed to support a particular set of instructions must be consolidated into one place: the control unit state machine.

Once it has been determined which instructions are needed, the appropriate modules are brought together by the generator. It is possible that some may depend on others, as some instructions may be similar enough that they can share microcode states. In this case, the generator must detect this requirement and bring in the additional modules that are required. The modules will then be written out in VHDL as a state machine, in much the same form as seen earlier.

The source of the modules

If the control unit was non-modular, each instruction would be defined directly in the state machine `case` statement. This could be done here, but it would have to be possible for the generator program to parse the state machine `case` statement, break it down into modules somehow, and then generate it again minus the parts that are not needed. But that would require the generator to be able to parse VHDL - to be able to differentiate between states, and to pick out which states could follow a particular state (essential to satisfy the dependency requirements).

This is actually quite a difficult problem. The generator's parser has to be as powerful and as smart as the VHDL parser used by XST. It's not enough to just scan for each piece of VHDL matching "`when "`*number*`" =>`" and assume that this indicates that a new state is starting. What if a particular state contained a `case` statement, perhaps to select a control line output? Every `when` in this inner `case` statement would be read as a new state. Detecting which states could follow a particular state is potentially even more difficult. It's easy if, in every case, the assignment takes a single form, such as "`next_state <= "`*number*`"`". But what if the designer wished to write the assignment in some other way?

Fortunately there is no need to solve this problem. The generator can insist that the module descriptions it reads are not pure VHDL - that complicated parts, such as the start of a new state or the setting of the next state, are written in some other easily recognisable form. Of course, it is still a good idea to keep the other parts as VHDL: doing so gives a lot of flexibility for control line assignments and conditional branches.

All of this makes it much easier to write a generator. Now the generator's job is to produce the state machine VHDL by very simplistic translation of some module descriptions. The job is no longer to interpret some VHDL, work out what the modules are, and then produce the VHDL for those modules.

It was decided that module descriptions would be placed in a series of "state machine" files. These files would be pseudo-VHDL; VHDL with three additional commands - as seen in Table 2. The state labels in the generated VHDL are really numbers instead of names, but this is transparent to the module writer. It is easier to use numbers for state labels, because then the next state can be (by default) the current state plus one. This means that the writer does not have to `LABEL` every state and explicity `JUMP` from one state to another.

Note that `JUMP` may appear in an `if` or `case` statement because it translates to an assignment to the next state signal. However, `CLOCK` may not. `CLOCK` basically translates to a new state label, with an appropriate number. It cannot be put into an `if` because the generator would have to split the `if` across two states. This would mean the VHDL would have to be parsed properly, and this is something that we wish to avoid. The command is called `CLOCK` because it really means "wait for a clock edge": every state transition happens on a clock edge.

Table 2: Pseudo-VHDL for State Machine Module Files

Command	Meaning
`LABEL` *name*	Label the current state as *name*. States only need to be labelled if they will be `JUMP`ed to.
`JUMP` *name*	The next state will be the state labelled with *name*.
`CLOCK`	Indicates that one state has finished and a new one has begun. VHDL contained between two `CLOCK` commands, or before the first `CLOCK` command in a file, goes into a single state X. VHDL after one `CLOCK` goes into state X plus one, and so on.

Advanced State Machines

Research into the 68020 instruction set indicated that the following operations are carried out by many different instructions:

- Use the "Effective Address[7]" field of the current opcode to load an operand.
- Use the "Effective Address" field of the current opcode to store a result.
- Fetch an immediate value into a register.

These operations are quite complex. Obviously, handling them in more than one set of states is not a good idea. In a software language, they would be handled by a subroutine of some sort. This would avoid cutting and pasting the same code into each routine, which would be wasteful of program memory and would also be very difficult to maintain.

A way to handle this is to do some operations, particularly the effective address ones, before instruction execution begins. As these are common to many instructions, the instruction decoder might determine that effective address decoding would be required. It would then run special effective address states before starting instruction execution. This has some disadvantages. It would add some overhead to all instructions using an effective address, even if they were only using a simple addressing mode such as Register Direct. More difficulty is caused by the `MOVE` instruction, which has two effective address fields that must be decoded separately.

The 68020 designers appear to have used the above approach with a series of clever modifications to handle all the special cases. Special cases are unpleasant things to have to handle, so alternative methods will be considered.

The use of multiple state machines was investigated. A "sub-state machine" could provide the Effective Address operations. It would take over from the main machine when called in some way. Unfortunately, this introduces more problems. It is difficult to design a good way to describe this sort of multi-level state machine. It is wasteful that every state machine has to have its own control hardware (state register, etc) and, in experiments, it was found to be quite difficult to keep the machines synchronised.

But a better alternative exists. There is no reason why the state machine cannot use subroutines. Imagine if the `JUMP` command featured in Table 2 was extended to include a stacking operation: a `CALL` and `RETURN` command could be added. Then any state could call a subroutine, consisting of one or more states. When that subroutine was finished, it would jump back to the return state, taken from the stack. For the (small) overhead of some additional stack logic, a flexible system of subroutine calls will be available.

Very little logic would be needed to provide the stack, because subroutines are unlikely to be nested deeply in the stack machine (it is not as if recursion is ever needed). Only a few stack registers would be needed. The stack pointer register and increment/decrement hardware would also be tiny because of this. If the stack had space

[7] This field is six bits wide and occupies the least significant bits in some opcodes. It specifies an addressing mode, which indicates the location in memory (or a register) where one of the operands for the instruction can be found.

for 8 items, the stack pointer would be only 3 bits wide, and a 3 bit adder and register could be implemented in as few as 3 FPGA cells. And it needn't slow the system down: with careful design, all stack operations could take place in the same time taken by a `JUMP`.

The use of a stack approach means that states that implement common tasks can be reused, and reused easily without the need for handling special cases. There is no limit on the number of subroutines that may be used, so any other common operation may also be moved into a subroutine to save space in the control unit. And the fact that `JUMP` and `LABEL` are already required means that this is just an extension to an existing system.

State Machine Stack Requirements

The stack operations should complete in a minimal amount of time - stack operations shouldn't waste a clock cycle. The stack should support `CALL` and `RETURN` commands: one to call a new subroutine by label, another to return from one. Table 3 specifies these new commands.

Table 3: Additional Commands for State Machine Module Files

Command	Meaning
`CALL` *name*	Call the state labelled as *name*, by putting a return point (the implicit next state: the current state plus one) on the stack, then jumping to the state labelled as *name*, and increment the stack pointer.
`RETURN`	Decrement the stack pointer register, and jump to the state on the top of the stack.

Just as with `JUMP`, it should be possible to put `CALL` or `RETURN` in an `if` or `case` statement. This will make them as flexible as `JUMP` is: use of a stack should not require any features to be taken away.

10.2. Instruction Decoder

In the modular instruction decoder, a few simplifying assumptions can be made. First, it can be assumed that all the opcodes that may be executed are known by the generator. This assumption is central to the entire project: since the processor is intended to be ideally tailored to the program, the program must be known in its entirety before generation can begin.

Given this first assumption, we can assume that no illegal opcode ever reaches the decoder - obviously, an illegal opcode would be picked up when the program was examined.

So the instruction decoder doesn't need to fully decode each opcode, because it is known that the opcode will be a member of a set of possible opcodes, taken from the program that will be executed. This set will be obtained by the program scanner.

Research into the design of the modular instruction decoder began by looking at how a complete instruction decoder would be implemented for the 68020.

68020 instruction decoder

The original 68020 instruction decoder fully decoded each opcode, so that any illegal opcode was always detected. The 68020 designers used Karnaugh maps to find the minimal logical functions that decoded each opcode bit pattern to a state value [Tredennick 1988]. The process was done by hand.

Here, the generation of the instruction decoder must be automatic. However, the use of VHDL means that there is no need to attempt any minimisation of logical functions in the generator: the decoder can be written entirely in high level VHDL.

An examination of the 68020 instruction set reveals that it is not always easy to identify an opcode. In a RISC processor, instruction decoding is usually just a matter of examining about 4 bits in the opcode. Unfortunately

the same is not true in the 68020. The situation shown in Table 4 is very common. Table 4 shows all six possible forms of the `ADD` instruction. All have very similar opcode formats, but the operations required are very different. For example, form 1 sends the result to a data register, and form 2 sends the result to an address in memory[8].

Table 4: `ADD` - A Difficult Decoding Problem

	Opcode	**Function**	**Size**
1	`1101` yyy `0`SS $EEEEEE$	$D_y \leftarrow D_y + [EA]$	Any
2	`1101` yyy `1`SS $EEEEEE$	$[EA] \leftarrow D_y + [EA]$	Any
3	`1101` yyy `1`SS `000`xxx	$D_y \leftarrow D_y + D_x + Extend$	Any
4	`1101` yyy `1`SS `001`xxx	$[A_y] \leftarrow [-A_x]+[-A_y]+Extend$	Any
5	`1101` yyy `011` $EEEEEE$	$[EA] \leftarrow A_y + [EA]$	Word
6	`1101` yyy `111` $EEEEEE$	$[EA] \leftarrow A_y + [EA]$	DWord

In order to exploit redundancies in the opcode format, the 68020 designers have packed six operations into one opcode form. Suppose an instruction decoder has determined, from the high nibble, that the opcode is one of the above. It must now narrow it down to one operation by applying a series of rules.

- If bits 6 and 7 (numbering from 0 as the least significant) are 00, 01 or 10, then the opcode has a valid size field (marked SS in Table 4). The operation could be 1, 2, 3 or 4:
  - If bit 8 is zero, then the operation is 1.
  - If bit 8 is one, the operation could be 2, 3 or 4:
    - If bits 3 to 5 are all zero, the operation is 3.
    - If bits 3 to 5 are 001, the operation is 4.
    - If bits 3 to 5 are neither 001 or 000, the operation is 2.
- If bits 6 and 7 are 11, the opcode doesn't have a valid size field. The operation could be 5 or 6.
  - If bit 8 is zero, then the operation is 5.
  - If bit 8 is one, then the operation is 6.

This type of opcode makes things more difficult for the designer of an instruction decoder. The type of the instruction is not indicated by the same bits in every case. There are complicated rules for decoding.. some instructions have simple rules (e.g. operation type 1), and others have very complicated rules involving the checking of many parts of the bit pattern (e.g. operation 3).

The tests must be applied in parallel to obtain single cycle instruction decoding. An easy method for doing this is used in the vm68k library. vm68k has a series of tables, with (together) 65536 entries - one entry for every possible opcode ($2^{16} = 65536$). The opcode itself is used as the index into the tables. The appropriate row of the table contains a few pieces of information about how to execute the operation.

VHDL, however, provides a less wasteful solution. A series of `if` statements can be nested together to determine which operation is needed. The code fragment in Figure 8 is an example.

The VHDL above will be minimised by XST into logical functions that translate the instruction register bits into the instruction decoder output. This seems an ideal way to solve the problem of decoding: all the opcodes that are special cases can be handled by code like this.

Unfortunately, this is not the best solution for this project for one simple reason: it cannot be used to make a modular instruction decoder; one tailored to a particular program, and supporting only the instructions used

[8] Note: The bitfield marked yyy or xxx is a 3 bit register number. The bitfield marked SS is the 2 bit operation size (byte, word or double word), and the bitfield marked EEEEEE is the effective address field.

```
                -- Examine most significant nibble
case instruction_register ( 15 downto 12 ) is
    ...
when "1101" =>  -- This is one of the ADD operations, but which one?
    if ( instruction_register ( 7 downto 6 ) /= "11" )
    then
        if ( instruction_register ( 8 ) = '1' )
        then
            instruction_decoder_output <= ADD_TYPE_1 ;
        else
            case instruction_register ( 5 downto 3 ) is
            when "000" =>  instruction_decoder_output <= ADD_TYPE_3 ;
            when "001" =>  instruction_decoder_output <= ADD_TYPE_4 ;
            when others => instruction_decoder_output <= ADD_TYPE_2 ;
            end case ;
```

Figure 8: Part of a VHDL instruction decoder

by that program. How can unnecessary `if` statements be removed when a particular program will never need those decisions to be made?

It may, for example, be known that although the `ADD` instruction is present in the program, it is only present in the first form shown in Table 4. So only the most significant nibble needs to be examined to determine that it is an `ADD` of type 1.

Optimised instruction decoder requirements

The instruction decoder must be generated so that it is able to decode only the instructions that will actually be used, and no more.

Any method of solving this problem would require the decoding instructions for each type of opcode to appear in some sort of database. The instruction decoder generator would take the decoding instructions for every required opcode from the database and put them together into a minimal decoder. It would do this by looking at the difference between the opcodes.

This "opcode database" would specify (in some machine-readable way) all the bit patterns that could make up a particular instruction. It would need to specify the difference between the six different operation types for `ADD` (and any other opcodes where this pattern occurs) so that an appropriate sequence of states could be used for each.

10.3. Arithmetic and Logic Unit (ALU)

The project's ALU must provide the same features as the real 68020 ALU: features common to all ALUs. It will be possible to leave some of these out, and therefore it will be modular.

The program will be examined to determine which ALU operations will be required. Add will always be needed, because internal operations such as incrementing the program counter depend it. The logical operations may not always be required, and this may allow the amount of logic required to be reduced.

10.4. Register File

The register file must provide eight data registers and eight address registers. All must be 32 bits wide: a double word. However, it must be possible to update only the least significant word of either set of registers, and also possible to update only the least significant byte of the data registers to support byte and word length operations.

Examination of the 68020 instruction set indicates that it is never necessary to read from more than two of these registers at once. This would only be needed if it was possible to have more than one instruction executing at once, with a pipeline, and this is a feature that is being left out for simplicity. So the register file must have two outputs. Similarly, the register file only needs one input, as only one register is updated at a time.

10.5. Memory implementation

The processor required memory before it could be shown to work: a ROM was required to contain the program being executed, RAM to hold the program's stack and global data, and hardware to manage the memory map. None of these are actually part of the processor, but they are essential if the processor is to actually execute anything. Each of these components is discussed in turn in this section.

A note about memory accesses

The 68020 is a full 32 bit processor, which means that it has a 32 bit data bus. It can therefore fetch 32 bits from RAM simultaneously, taking only one bus cycle to load a register from memory.

This might suggest that it would be most convenient to organise the memory, like the 68020 does, as 32 bit words. Then fetches and stores could be done in one clock cycle (there is no bus, so memory access is done directly).

Unfortunately, this fails to take into account two problems. The first problem is that of unaligned memory accesses. The 68020 is a byte-addressed processor - addresses can legally point to any address in memory. However, the memory is not byte-addressed, and this can be a problem.

Suppose, for example, that a program wishes to read two 32 bit words from RAM, at addresses `0x1000` and `0x1005`. The first access is no problem, but the second is an unaligned access. Because the memory is organised as a number of 32 bit words, the addresses of each word are all on 32 bit boundaries: 0, 4, 8, 12, 16, and so on. `0x1000` is on a 32 bit boundary, so it addresses one complete 32 bit word. `0x1005` is not on a 32 bit boundary: it addresses 24 bits from one 32 bit word at `0x1004`, and 8 bits from another at `0x1008`. Both words must be fetched, and the processor must use the correct bits from each to get the word that was wanted. Clearly, this requires an extra fetch and extra logic to handle this situation. A similar problem occurs when writing to an unaligned address.

The solution often employed to solve this problem is to make the whole processor only work with aligned addresses. This is done in the RISC PowerPC processor. But the 68020 clone cannot simply throw an exception if an unaligned access is attempted. This would put a huge restriction on the type of code that could be executed.

Fortunately, there is a simple solution to this problem: make the memory byte-addressable. If the memory is organised as bytes, there are no restrictions on alignment and there is no need for special techniques to be used to get around the alignment problem. This is slower, because fetching a 32 bit word will now take 4 fetches. However, the ease of implementation is a great advantage: it saves a lot of design effort, a lot of testing, and more space is available on the FPGA since the memory access hardware is simple.

How should the ROM be implemented?

In VHDL, ROM is implemented as a large table. Like real ROM, the table translates an address into the data at that address. An example is shown in Figure 9.

```
case address is
...
when "1100100" => data <= "11111111" ; -- ff
when "1100101" => data <= "11111100" ; -- fc
when "1100110" => data <= "01000010" ; -- 42
when "1100111" => data <= "10101110" ; -- ae
...
```

Figure 9: Part of a VHDL ROM

This table could be built by entering the ROM data by hand - looking at a dump of the program binary. But this is clearly ridiculous for programs of any size: even a tiny 100 byte program would take a long time to enter.

The alternative is to write a program to generate a ROM table. Since it has already been decided to write a program to read Intel Hex binaries (in Section 8.2), the ROM generator can read a binary in this format and produce a VHDL entity from it. The entity will translate an address into the data at that address, using a VHDL table. It is trivial to implement this code, since an Intel Hex reader is already needed for the vm68k emulator.

Please note that, although the ROM will be generated by a program, this program should be separate from the program that generates the rest of the processor. We do not want the user of the processor to be forced to use this particular ROM: we want the user to be able to choose a ROM to suit the application. Although it would be possible to make the generator program also produce the ROM (after all, it must scan the 68020 program, so it could generate a ROM while it did that), this would force the use of this particular ROM on the user of the processor. So the ROM is assumed to be generated independently of the other parts of the processor.

How should the RAM be implemented?

The designer is faced with another choice when deciding how RAM should be implemented. The memory map discussed earlier decided that 4Kbytes of RAM would be more than enough. So the choice is only where this memory should be located.

Here, the real choice is between the use of the FPGA's on board block RAM (as used for the register file) and an off chip SRAM module. This is because the designer cannot really hope to implement the RAM as 4096 8 bit registers on the FPGA - the amount of logic needed would be a complete waste of FPGA space.

The BurchEd FPGA board comes with an SRAM module that can be attached, and provides 512 Kbytes of memory. With a 15ns access time, it will be quite possible to access the SRAM at the speed of the FPGA - but the fact remains that VHDL will still have to be written to drive it.

There is enough block RAM on the FPGA to provide the space required. The XC2S300E FPGA has a total of 64 kbits of block RAM, organised into 16 blocks, according to [Xilinx 2002]. 4Kbytes will take up space in eight of these blocks, as each block will hold 4Kbits of data, or 512 bytes. It is easy to write VHDL that uses the Block RAM: the manual provides an example.

It was decided that the RAM should use the FPGA's on board block RAM, because this is much simpler. There is no need to produce a driver for the off chip SRAM.

Memory Mapper design

The memory mapper should implement the memory map seen in Table 1 on Page 12. The design can be quite simple, since a single nibble of the address (bits 12 to 15) selects the type of memory being accessed. If this nibble is zero, the access is directed to ROM. If it is one, the access is directed to RAM. And if it is eight, the access is directed to the output device.

10.6. Debugging Hardware

It is very important to have some sort of system in place for making sure that the processor executes code as expected, and to allow faults to be found. The system should allow the tester to verify all aspects of the operation of the processor at the lowest level.

It is possible to verify that the processor works by giving it a test program that outputs a known result to the output display (as mentioned in the memory map). If the result appears, the test passed. This is fine if the processor works correctly, but it can only indicate when something is wrong and the tester is then left guessing about the problem. A more comprehensive debugging system is required.

To design a useful debugging system, the features of a typical software debugger were examined. Software debuggers have a number of very useful features, including:

- Display contents of variables and registers
- Single step at the machine instruction level
- Single step at the source code level (line by line)
- Run (at full speed) until a breakpoint is reached

But what is required is actually a hardware debugger. Some logic is needed that can provide some sort of debugging display, capable of displaying many different aspects of the processor's operation, and also provide some way to slow down or stop the processor so that its changing internal state can be watched properly.

The design inspiration for this part of the work is the PDP-11 console: a set of lights and switches on the front of the processor. The lights displayed an address register and a data register. The switches allowed the PDP-11 to be halted, single stepped through instructions or bus cycles, and also allowed memory to be examined. Figure 10 illustrates the console.

Figure 10: DEC PDP-11/40 console

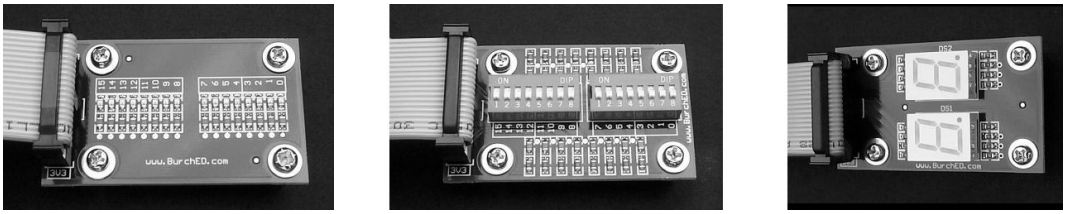


Figure 11: Three BurchEd Add-on Boards. From left to right: 16 LED board, 16 switch board, and a dual seven segment display board which can display two hexadecimal digits

The BurchEd FPGA board provides several add-on boards which could be useful for building some kind of debugging hardware. Figure 11 illustrates these.

As on the PDP-11, the debugging hardware should be driven by the switches. These will allow the tester to interact with the debugger, examine registers and memory, and stop the processor. Once stopped, it should be possible to advance the processor by one clock cycle: from the processor's perspective, no unit of time is shorter than a clock cycle. So, at this level, the most information about the processor's internal operations will be available. And by stepping through several clock cycles, it will be possible to step through the instructions as well. Stepping should take place when a button or switch is pressed.

The debugging hardware should allow examination of as many internal registers as possible, without intrusively affecting the design decisions. For example, it wouldn't be easy to allow the debugger to modify any registers. Doing this would involve substantial additions to the architecture: another multiplexer on every input to every register.

The hardware should also allow examination of some memory addresses. There is no need to allow all possible memory addresses to be examined - this would require 32 switches for all the address bits - so some subset will have to be decided upon.

It is important that the debugging hardware should be an optional module. By its nature, it is not an essential part of the processor, and it should be easy to leave it out.

10.7. Output Device

The memory map (see Table 1 on Page 12) makes reference to an output device. The output device allows a single byte of output to be displayed, thus allowing a program to produce a sequence of results.

Since the debugging hardware discussed in the previous section will use a display, it makes sense to share the display between the output device and the debugging hardware. Perhaps in one of the debugger's display modes, it could show the last value written to the output device at `0x8000`. This would allow the program's output to be monitored, but also allow the processor to be debugged if the correct output failed to appear.

11. The Generator

In Section 9, the processor features that could be modularised were discussed. It was suggested that all of the modular features could be provided by the automatic generation of parts of the VHDL for the processor. This would make it easy to eliminate useless states from the control unit, useless decoding logic in the instruction decoder, and would help to optimise the ALU. It would also allow the address register width to be set, and unused addressing modes to be eliminated.

The automatic generation of the components will be directed at one or more files. Only a minimal amount of VHDL should be generated so that the user never needs to modify the generator or output files.

11.1. How should VHDL files be generated?

There are two ways to handle automatic generation. One is to have the generator program produce VHDL entities to represent each modular component. Not all parts are generated - some are hand written and are placed in entities that make use of the automatically generated parts. Figure 12 illustrates this approach. It shows the VHDL entities that would make up the processor. The surrounding "core" entity contains all the others and ties them together by connecting signals between them. The shaded entities are the automatically generated ones.

This approach helps the implementor to think about the separation between the components of the processor. It is easier to understand how changes in one will affect another, so the chances of mistakes being made are reduced.

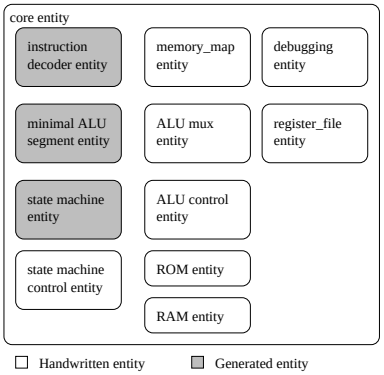


Figure 12: Multiple Entity Approach

An alternative approach is to put all VHDL in the same file. The automatic generator can "import" fixed, hand written VHDL to tie the automatically produced parts together, so there is no need to spend a long time writing extra entities to connect the parts. Figure 13 illustrates this approach.

The deciding factor that allows the second option to be chosen instead of the first is that the multiple entity approach is actually harder work for the implementor, because extra VHDL has to be written to link the entities together. And the easy to understand structure of the multiple entity approach can be preserved by allowing multiple files to be imported into the automatically generated file, without the need to write extra linking VHDL. The fact that all the generated VHDL goes into one file along with all the handwritten VHDL is of no real importance, since from the implementor's perspective, it all comes from well-separated files. Additionally, a bare minimum of VHDL is generated by this approach.

So generation will produce a single VHDL file, after scanning a 68020 program to determine the optimal configuration for the processor components. Parts of this file will be fixed at the implementation stage: these parts will be imported from a series of VHDL files. Other parts will be automatically generated.

It makes a lot of sense to adopt a hierarchical structure here. A top level, hand written file will import other fixed parts (to support the various processor components) but also indicate where the generator should generate VHDL. This gives a good degree of flexibility - just by modifying the top level file or one of the ones it imports,

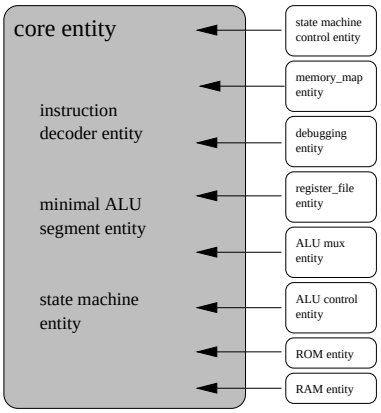


Figure 13: Imported Entity Approach

the internal structure of the processor can be easy modified by the user or tester to suit his or her requirements. Importing can be done in a way that is analogous to the C preprocessor, with statements such as "`INCLUDE` *file*" that are replaced in the output with the contents of the named file.

11.2. Generator Directives

If the generator is to produce the VHDL output file by the method described in the previous section, a number of *directives* will be required. These are instructions to the generator to tell it what files to read and what to generate.

The C preprocessor imports files by the `#include` directive. Similar directives are needed here. The form of these directives is arbitrary, as long as they are easily recognisable by the implementor. Table 5 lists the directives that were selected. (It is no coincidence that these are similar in form to the commands for the state machine compiler: `CLOCK`, `JUMP` etc. These directives translate into valid VHDL just as those commands did.)

Table 5: Directives For Generator Input Files

Directive	**Meaning**
`INCLUDE` *file*	The directive is replaced in the output by the contents of *file*
`INSERT STATE MACHINE`	The control unit state machine is built and put into the output file, replacing the directive.
`INSERT INSTRUCTION DECODER`	The instruction decoder is built and put into the output file.
`INSERT OPTIMISATION` *op*	A piece of VHDL is inserted to optimise a particular operation *op*.

11.3. Design of a 68020 program scanner

The program scanner must find every opcode that is used in a 68020 program binary, so that it is known which instructions, addressing modes, and ALU operations are required. It is an essential part of the generator.

The only way to approach this problem is to scan through the file, examining all the opcodes that *might* be executed. It might be thought that the problem could be solved by running the program, perhaps in the vm68k

emulator, and making a note of each opcode that was executed. But this is not possible: How would the scanner ensure that every possible execution path was executed? A static approach must be taken.

This is still not an easy task. Not all of the program binary is code (data segments also appear in the binary). And opcodes vary in length: some have immediate data following them. Luckily, a tool has already been written to interpret binaries in this way - the disassembler.

The GCC disassembler, `objdump`, can take a binary and dump out all the opcodes used in it. It is smart enough to only disassemble the program's text segment, which will contain only 68020 instructions. The program scanner will use the disassembler output to build the set of required opcodes. There is clearly no reason to attempt to replicate the disassembler's function, when the output can be used directly.

12. Designing state machine sequences for instruction execution

So far, it has been said that the control unit will guide the processor through the phases of fetching, decoding and executing an instruction by producing a correct control line sequence. But how should these control line sequences be designed?

Of course, the instruction execution must be compatible with the 68020 - a particular instruction must do what the 68020 manual [Motorola 1985] states that it should. There is already a complete specification for each instruction. And this specification must be translated into a series of state machine states, each with control line assignments.

It will certainly help to define each series of states by "Register Transfers". This is a sort of pseudocode: it isn't compiled or translated directly. Register transfers are often used to describe internal processor operations: they are commonly shown in processor documentation and appear throughout [Motorola 1985] and [Hennessy 1996]. They are simply assignments to a register: $PC \leftarrow PC + 2$ (increment PC) is one example, and others appear in Table 4. There can be any number of register transfers in a state, as long as the processor can allow them to take place simultaneously.

These register transfers are actually implemented by setting one or more control lines. But by discussing the transfer in this higher-level way, the operation that is actually taking place is much clearer.

The first stage in using register transfers to design the implementation of an instruction is to define the transfers sequentially, as if each one took place in a separate state machine state. The next stage is to merge these states together where possible, perhaps reordering the operations if this doesn't break the overall effect, so that the instruction is implemented in a minimal number of states. Finally, the register transfers can be translated into control line assignments and written out in VHDL. These tasks are best done by hand.

Part IV. Implementation Phase

13. Implementing the fixed parts of the processor

The fixed parts of the processor are those that are never changed by generation - they are entirely hand written. It is a good idea to implement the fixed parts of the processor first. Then the names and types of the connections between generated components and fixed ones are all decided upon when writing the generated parts, making the task somewhat easier. The design of each fixed part will now be discussed.

The core VHDL file, which includes all of these parts and tells the generator where to include the generated parts, is called `input.vhd`. Source code can be found in Section E.7, Page 82.

13.1. The Control Logic

It has been implied that the entire state machine would be generated. This cannot actually be the case. A stack controller is needed: it is part of the state machine, but it is a fixed part. The state machine tells it what to do, and it uses the stack to provide the required operation. It is a small, but very significant, part of the state machine.

A State Machine Controller

After some experimentation, a state machine controller was designed and implemented. It provides both `CALL` and `RETURN` stack operations, as well as a mechanism for `JUMP`. A state may use any of these operations, which set the next state. If none are used, the next state is the current state plus one.

The controller works using three control lines. They are `call_requested`, `return_requested` and `call_state`. Every pseudo-VHDL command (except `CLOCK`) is translated to a setting of these three lines, which are zero by default. Table 6 shows the settings for each operation. As can be seen, `call_state` is used to notify the controller which state should be `CALL`ed or `JUMP`ed to.

Table 6: State Machine Controller Truth Table

`call_requested`	`return_requested`	Action	Controller Operations
0	0	`NONE`	$state \leftarrow state + 1$
0	1	`RETURN`	$stack\_pointer \leftarrow stack\_pointer - 1$ $state \leftarrow stack[stack\_pointer]$
1	0	`CALL`	$stack[stack\_pointer] \leftarrow state + 1$ $stack\_pointer \leftarrow stack\_pointer + 1$ $state \leftarrow call\_state$
1	1	`JUMP`	$state \leftarrow call\_state$

It's essential that all of the operations take place within a clock cycle, because if they took longer than that, some of the state machine's time would be wasted by the controller. This is tricky to arrange, because it's difficult to change the stack pointer and, at the same time, load or unload data from the stack. The solution turned out to be to do all the stacking operations on the opposite clock edge to state transitions. There are two clock edges: positive-going and negative-going. The controller used the negative-going clock edge to update the state variable and stack pointer, but stack store and load operations were done on the positive-going clock edge.

When a `RETURN` operation was requested, the state number to be returned to had already been fetched from the stack on the previous positive-going clock edge. So the controller only needed to copy this data into `state` and update the stack pointer on the negative-going clock edge.

When a `CALL` operation was requested, the stack pointer was updated along with `state` by the controller on the negative-going clock edge. The storage of the old `state` value in the stack was deferred until the next positive-going clock edge, by storing the old value in a special "`value_to_be_stacked`" register and setting a "`write_enable`" flag for that clock cycle only. In this way, `CALL` and `RETURN` took just one clock cycle: the same amount of time as the far simpler `JUMP` operation.

The state machine controller that was implemented has an 8-item stack, meaning it needs only a 3 bit stack pointer. Stack overflow will not be detected as there is no hardware to do this. Fortunately, it is possible for the debugging hardware to monitor the stack pointer.

The state machine controller has support for a reset button. If a reset line is set, the stack pointer and current state are reset to zero, thus starting the processor again. This reset line is connected to the debugging hardware: a particular switch setting caused a reset.

⇒ *The source code of* `state_machine_controller.vhd` *can be found in Section E.12, Page 89*

13.2. The ALU

It is quite easy to build a 68020 compatible ALU in VHDL. It can be done using discrete logic - perhaps based on an ALU design taken from one of many textbooks describing such things. But a better way is to use a design that can be recognised by XST as being part of an ALU. XST can then use an optimised implementation, minimising FPGA usage and maximising speed.

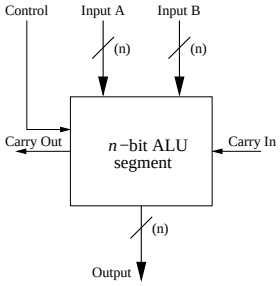


Figure 14: An ALU segment for n bits of data

The ALU design used here was inspired by a design seen in T80 [Wallner 2002], a soft processor written in VHDL that is a clone of the Z80 microprocessor. The Z80 ALU is only 8 bits wide, but it is otherwise very similar to the 68020 ALU. The T80 design features a procedure, AddSub, that will either add or subtract a set of n bits. The procedure provides an ALU "segment". Figure 14 illustrates one ALU segment in diagrammatic form. Several segments are chained together, as seen in Figure 15, allowing the ALU to obtain the Carry and Overflow outputs generated by the operation. This design is recognised by XST and made into an optimised implementation.

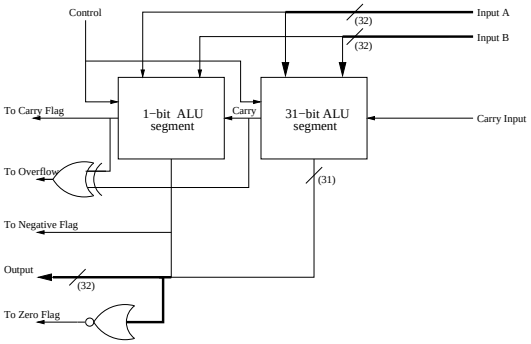


Figure 15: A 32 bit ALU

The typical ALU has four flag outputs in addition to the result output. The four flags are traditionally named Carry, Overflow, Negative, and Zero (shortened to CVNZ). They refer to the result of the operation and are self-explanatory.

The Negative and Zero flags are easily generated. The Carry flag is simply connected to the carry output of the ALU segment that deals with the most significant bits. The Overflow flag is rather more complicated. It is the exclusive-OR of the carry outputs from the two most significant bits, so it is '1' if these carry flags differ[9].

The ALU will support all typical ALU operations: Add, AND, EOR, OR, and subtract. Because the designer wishes to minimise the number of data links running to the ALU from the registers, it is also a good idea to provide a reversed subtraction operation. All the other operations are commutative: $A + B = B + A$. Subtraction, however, is not commutative, so the only ways to provide both A - B and B - A are to arrange for both B and

[9] Because, if both are the same, then they are either both zero or both one. If both are zero, then the operation cannot have overflowed - because the carry out is zero. If both are one, then the carry out was not caused by the 31st bit, it was caused by some earlier bit. So the carry was not caused by an overflow - it is a borrow caused by a subtraction.

A to run to each input of the ALU, or provide a reversed subtraction operation. The reversed subtraction is believed to be preferable since it appears that XST is able to provide this feature in the optimised ALU logic it is able to generate.

The 68020 is able to do arithmetic operations on byte, word and double word types. This complicates the ALU design slightly: one might think that all three types could be treated as the same due to the properties of 2's complement arithmetic. Unfortunately, the fact that the ALU must provide four flag outputs means that this is impossible. It is essential that the ALU gets the flag information from only the first 8 bits for a byte operation, and only the first 16 bits for a word operation.

The alu_segment entity

The ALU that was implemented is based on a VHDL entity, `alu_segment`. It provides all the operations that the 68020 might require, including reversed subtraction. It operates on a variable number of bits that is set by a generic parameter.

The operation is quite simple. If a logical operation is required, it is applied directly. Otherwise, an addition is applied to the inputs. If a subtraction is required, the 1's complement of one input (A or B, depending on whether this is a normal or reversed subtraction) is obtained by using a logical NOT of that input. This becomes a 2's complement because the carry input is expected to be inverted by the VHDL that contains the segment.

The ALU segment's use of existing VHDL primitives for all its operations means that XST is able to recognise it as an Adder/Subtracter with Carry Output. XST is able to use an optimised implementation of it on the FPGA, taking a minimal amount of space.

⇒ *The source code of* `alu_segment.vhd` *can be found in Section E.3, Page 78*

The container

The containing VHDL used six `alu_segment` entities. Figure 16 shows the final design of the ALU. Note that when a word-length operation takes place, the byte-length segments are still used (along with the byte-length flag generator), and when a double-word length operation takes place, the word-length segments are still used. This minimises the size of the logic required: it would be pointless to have three separate ALUs for handling each operation size, although that would be another way to approach the problem. The figure shows that three sets of flags are produced - one for each data type - but only one output. For byte or word operations, only the low 8 or 16 bits of the output are valid.

The Carry Input Generator shown sets the carry input to zero for add operations and to one for subtract operations (to obtain a 2's complement of the input). It outputs the reverse in only one situation: when the Extend flag must be taken into account. The 68020 has two operations (`ADDX` and `SUBX`) that use Extend as a carry input.

⇒ *The source code of* `alu.vhd` *can be found in Section E.1, Page 73*

Modularising the ALU segment

Unfortunately, one side effect of putting the ALU segment into a separate entity is that the internals cannot be generated automatically. This might appear to make it impossible to make the ALU segment modular, and remove the operations that are not needed in a particular program. But this is not the case. An alternative way to remove certain ALU features is to tell XST that the control lines cannot take particular values. This allows the ALU to be modularised easily, without worrying about how to correctly generate each segment.

It was noticed that, if there is no assignment of a value "X" to a particular control line, then any logic that is enabled only by "X" will not be synthesised. So if the control lines to the ALU can never request an EOR operation, then the EOR hardware will not be synthesised by XST.

But it is not really feasible to eliminate all assignments of the ALU control lines to EOR. This would require the input VHDL to be parsed by the generator. However, there is an alternative. An easy way to ensure that a control line never takes a particular value is to create a VHDL function that makes assignments to that control line and restricts the value appropriately. This automatically generated "optimisation function" is described in Section 15.3.

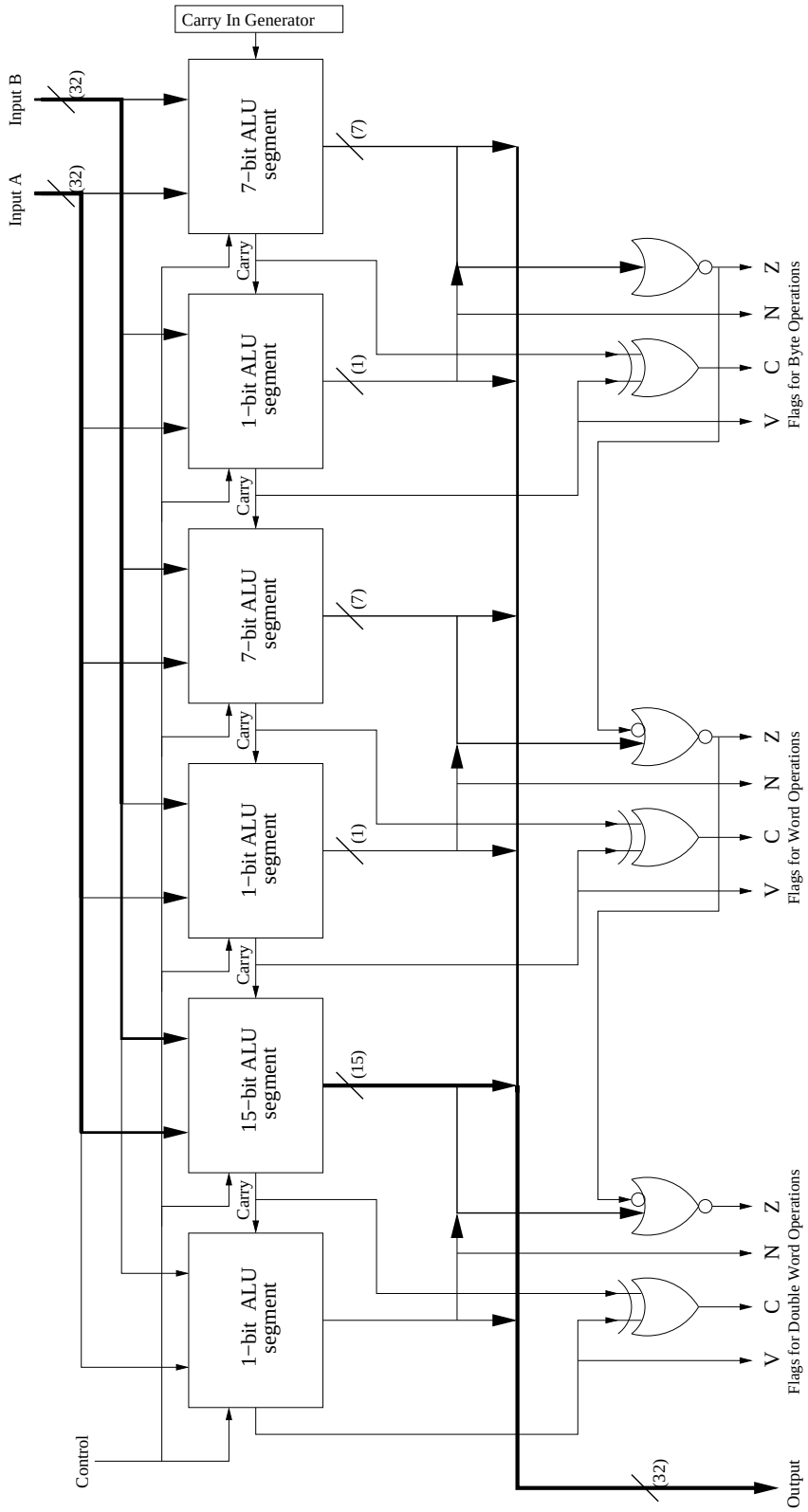


Figure 16: ALU Final Design

13.3. The Register File

There are two choices for the implementation. The file could be implemented as sixteen separate registers, each implemented as a separate VHDL process. Then the outputs of all the registers could run to two 16 input multiplexers to allow two registers to be selected.

Alternatively, the register file could be implemented using Block RAM. A small RAM, 32 bits wide by 16 rows deep, can provide all the functionality required. And all the hardware required to read and write data to and from it is already present on the FPGA.

Advantages to the first approach are twofold. One advantage is that access is asynchronous: the control unit can change which registers it is looking at without having to wait for a clock edge. This means that it is possible to do some operation on one pair of registers and then do another on another pair without having to waste an extra clock cycle to allow the second pair to be fetched. Another advantage is that registers are modular and can easily be removed.

The synchronous block RAM does not have either property. But the block RAM approach has the great advantage that it uses almost none of the configurable logic on the FPGA. No multiplexing hardware is needed - it's already part of the RAM. The data is stored in special-purpose memory, so no FPGA logic is used for storage or access. In this case, there is no advantage in removing some registers: no FPGA space will be saved by doing so.

Thus, this is a choice between a small implementation that is slower, and a large implementation that is faster and has some potential for modularisation. It should be noted, though, that there are two optimisations that can be made to speed up the block RAM implementation, resulting in very few wasted clock cycles.

The first optimisation is to store the data registers and the address registers in separate areas of block RAM, so that a total of four registers can be accessed at a time: two address registers and two data registers. This enables the second optimisation. Practically all 68000 instructions that access registers have the register numbers that are wanted in two bit positions in the opcode: at bits 0 to 2 and at bits 9 to 11. By using these bit patterns as address and data register numbers, and looking up their contents during the instruction fetch and decode phases, practically all the registers that an instruction might want to use are available when it is executed.

Because of the above two optimisations, it is quite possible to use block RAM to provide the register file. The potential for modularisation in the register file is not really that important, because very little space will be saved on the FPGA through such modularisation. The block RAM doesn't use much FPGA space anyway.

A sample piece of VHDL was found in the XST documentation that represents a dual-port RAM. A dual-port RAM allows read access to two locations in memory simultaneously, and write access to one of those locations, which is what is required as a minimum for the register file. A VHDL entity was written to provide reusable dual-port RAM functionality, based on the XST sample. It provides RAM of a variable size: the address width and data width can both be set by generic parameters.

⇒ *The source code of* `xilinx_dp_ram.vhd` *can be found in Section E.14, Page 90*

The entity was used to provide two separate register files: one for eight data registers and another for eight address registers.

⇒ *The source code of* `register_file.vhd` *can be found in Section E.10, Page 86*

13.4. The Memory Subsystem and Output Device

As discussed earlier, the memory subsystem is not really part of the processor, but it is essential for testing it. It (minimally) consists of a RAM (for global variables and the program stack), and a ROM (to hold the program binary). In this case, an output device capable of displaying a number is also included to help with testing. A memory mapper is also required, since there will be more than one memory device. The memory mapper needs to divide up the memory space as shown in Table 1 on Page 12.

Memory Mapper Implementation

The difficulty in building the memory mapper comes only from the fact that it must route data in two directions.

It must be able to send data to the appropriate type of memory, but also retrieve memory from it. Figure 17 is a diagram of its function.

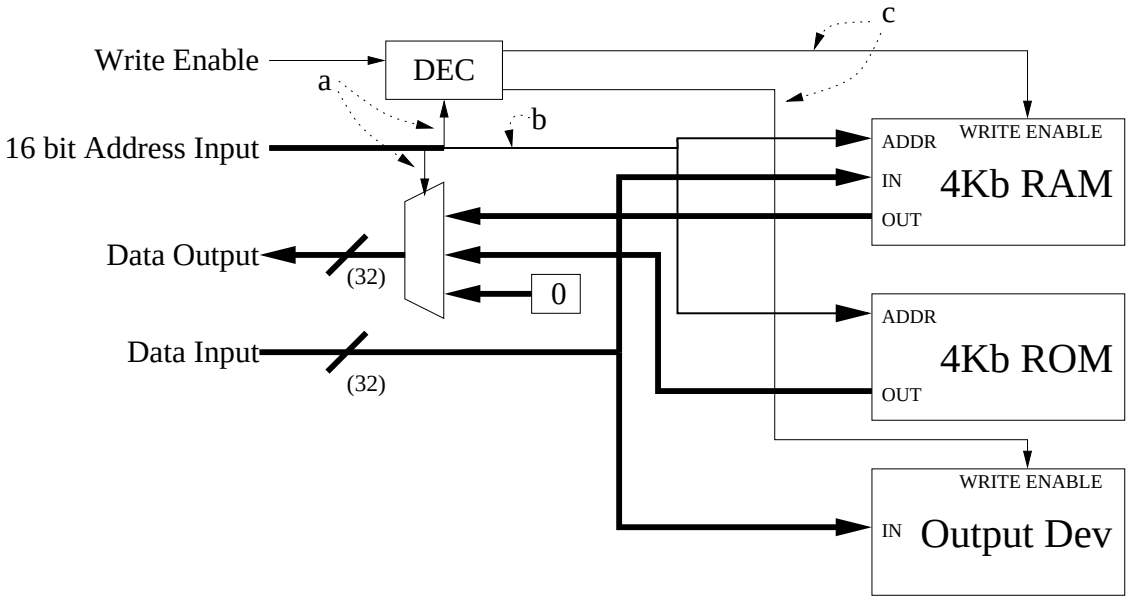


Figure 17: The memory mapper

Only the least significant 16 bits of the address are used. They are split into two parts. The high nibble is sent, via the lines labelled "a", to the output multiplexer and a decoder (labelled "DEC"). The output multiplexer selects a memory data source based on the value of the high nibble. If the nibble is zero, ROM is selected, and if it is one, RAM is selected. Otherwise, the output is zero, because an access to non-existent memory (or a read from the output device) has been attempted. The decoder is used for writing to memory. If the Write Enable input is on, then one of the control lines (labelled "c") will be switched on. This tells the device on the control line that the data on the input should be written to memory (which is why no control line runs to the ROM). The line that will be switched on is chosen by the high nibble of the address: the line to RAM is selected by a one in that nibble, and so on. By this method, the appropriate memory device is accessed for read and write. The low part of the address goes to both the ROM and RAM, where it indexes the correct location within the 4k block.

The design seen here is analogous to one commonly used in a real computer. Normally, however, all devices would sit on a bus and the decoder would enable them for reading or writing as needed. This requires fewer data lines (the same ones can be used for reading and writing) and no multiplexer is required. However, it requires each device to have bus-driving logic, and this is something that is avoided here.

RAM implementation

The RAM was very easy to implement thanks to the fact that a dual-port RAM entity had already been built for the register file. The data width of the RAM was one byte, and the total size was 4096 bytes, as discussed earlier. One of the two read ports, and the write port, were connected to the memory mapper to allow the program to use the RAM. The second read port was connected to the debugging hardware to allow it to inspect some RAM locations.

ROM implementation

In Section 10.5, it was determined that the ROM would be best implemented by having a program generate a ROM entity from a program binary in Intel Hex format.

A simple C++ program, `romfactory`, was written. It reads an Intel Hex binary using a C++ class, `ProgramRAM`. `ProgramRAM` subclasses the vm68k `memory` class: meaning that it provides an interface that vm68k

can use as memory - it has various functions for loading and storing information. A regular RAM would be initialised to zero, or left uninitialised, when constructed, but `ProgramRAM` reads an Intel Hex binary and initialises the memory space from that. So vm68k is able to use it to execute the program.

Although `ProgramRAM` was really written as part of the driver for vm68k that allows 68020 programs to be tested, it can be easily reused by `romfactory` to read the program binary.

Output Device implementation

The output device was implemented by creating a byte-width register, `last_output`. This register is updated whenever a program writes to memory location `0x8000`. The number that was last written to this register can be seen in one of the modes of the debugging hardware, discussed later.

VHDL source

⇒ *The source code of* `memory.vhd` *can be found in Section E.8, Page 83*

Note that both the decoder and multiplexer functions are carried out by the same process, `memory_map_process`. For convenience, the process also handles loading the output data into the output device register.

13.5. Debugging Hardware Implementation

It was decided in Section 10.6 that the debugging hardware would allow the user to monitor internal processor registers, inspect some RAM contents, and run the processor in a single-step mode. Clearly, in order to do this, it must be able to interact with the user. The debugging hardware should also not be an essential part of the processor, but this will be dealt with later.

The user interface components available are illustrated in Figure 11 on Page 24: a dual seven segment LED display, a row of 16 switches and a row of 16 LEDs. Additionally, the FPGA board has a single push button on it. The 16 LEDs are not particularly useful: it is hard to read the value of a register from those, as it must be expressed in binary. The seven segment displays are far more useful: here, the data is read as a number, which is far easier for a person. (New input or output devices could be built and interfaced to the FPGA if necessary, but there is no call for that here).

So the button and switches can be used as an input, and the seven segment displays can be used as an output. Two dual seven segment display boards were available: when connected side-by-side to the FPGA, they allow a 16 bit number to be shown.

It was decided that some of the switches would control the processor's speed (the debugger has to be able to single step the processor) and some others would control what data appeared on the display. The display would not be particularly useful if it always displayed the same information - the contents of one register, for example - because this is very inflexible. The solution to this is to connect the display to the output of a multiplexer, and connect various data sources to the inputs. The multiplexer would be controlled by some of the switches.

Debugging Output

Ideally, the user would want to be able to inspect any piece of information relevant to the processor. Unfortunately, there can only be a limited amount of information available at any one time, for two reasons. Firstly, only two of the data and address registers can be read at once, because of the use of block RAM to represent these registers. Secondly, every data source that is available for inspection adds to the amount of logic that must be synthesised because the data must be routed to the display.

So some compromise has to be reached: the most useful information has to be available to the user. It was decided to include the following types of information:

- As much information as possible about the contents of the register file.
- All internal processor registers - PC, IR, etc. except for the memory access registers (MAR etc).
- The ALU flags: Carry, Overflow, etc.

- The first 16 bytes of the RAM: `0x1000` to `0x100f`.
- The last byte written to the output device (`0x8000`)
- The current state number and stack pointer in the control unit state machine.

A maximal set of registers are made available to the tester. Unfortunately, since the display can only show 16 bit values, only the low 16 bits are available in most cases. This shortcoming is not such a serious problem as even looking at only the low 16 bits will still provide a lot of information to the tester. In the case of the PC register, the range is only from `0x0000` to `0x0fff` anyway - only 12 bits are needed.

It will certainly be useful to inspect the workings of the state machine - indeed, most processor problems are likely to be caused by bugs in the control line settings, due to the complexity of this part.

It was found that only 16 inputs to the multiplexer were needed to provide all of the above debugging outputs. The output that was wanted could therefore be selected by a four-bit binary number, or four switches. A fifth switch could select between inspecting a regular debugging output (a register, etc) and inspecting RAM. This is why only 16 RAM locations can be viewed: four switches were readily available to select the address. Of course, there were many more switches available on the switch board, but only five of them were needed for the debugging output - it would have been less convenient to use additional switches.

Testers of the processor should arrange to store variables in this space if they are to be viewed. Fortunately, this is where global variables will be placed automatically by the C compiler.

Table 7 contains a list of the possible debugging outputs along with their switch settings. The purpose of some of the registers in this table has not yet been discussed - they will be talked about in later sections. The source of the debugging multiplexer can be found in `debugging.vhd`.
⇒ *The source code of* `debugging.vhd` *can be found in Section E.5, Page 80*

As stated earlier, two dual seven segment LED displays were chosen as the output device. The LED display does need a small amount of controlling logic, to translate a byte into a number for each display. This function is provided by a simple ROM. Each nibble is translated into seven outputs: one for each segment of the display. A VHDL entity, `seven_segment_driver`, provides this functionality for each display.
⇒ *The source code of* `seven_segment_driver.vhd` *can be found in Section E.11, Page 88*

Single-stepping support

A button is available on the FPGA board, which seems an ideal way of stepping the processor: press the button, and the processor advances to the next clock edge. Since everything happens within the processor on a clock edge, there is no higher resolution than this for watching the processor's operations.

However, it is not particularly useful if the processor can only operate when stepped. Since a typical instruction may take four or five clock cycles, and some may take many more, it isn't practical to run an entire program in this fashion. So the processor must also support a "full-speed" mode, in which it runs at the speed of the FPGA's clock (a phase-locked loop controlled clock that can run at any speed from 1 to 100 MHz).

To support these modes of operation, two more switches were allocated to change the mode, as seen in Table 8. Note that it is possible to get from the reset state into either mode by moving only one switch: this is deliberate.

A substantial amount of VHDL was needed to support these features. The first part that was required was a debouncer for the button. When the button is pressed or released, the contacts within it will bounce slightly. This will often result in a number of "false presses" being recorded by the logic connected to the button. With the type of button available here, there are only two ways to avoid the problem, both involving special logic circuits. One way is to poll the button's state at a particular interval (for example, every 100 milliseconds). When a state change is detected, the button must have been pressed or released. Unfortunately, it's quite possible that the button may be polled while it is bouncing, in which case a false state change will still be detected. A better way is to start a counter when a state change is detected. Until the counter reaches a certain point, no further state changes will be considered. Thus, bounces are ignored. If the counter's count-up-to value is chosen correctly, the bounce problem is eliminated entirely.

Table 7: Debugging Outputs

Switch Setting	Display
00000	OA: `operand_address` (low 16 bits)
00001	OV: `operand_value` (low 16 bits)
00010	PC: `pc_register` (low 16 bits)
00011	IR: `instruction_register`
00100	A_x: address register file output X (low 16 bits)
00101	D_x: data register file output X (low 16 bits)
00110	$A_{y15..0}$: address register file output Y (low 16 bits)
00111	$A_{y31..16}$: address register file output Y (high 16 bits)
01000	$D_{y15..0}$: data register file output Y (low 16 bits)
01001	$D_{y31..16}$: data register file output Y (high 16 bits)
01010	$IDR_{15..0}$: immediate data reg (low 16 bits)
01011	$IDR_{31..16}$: immediate data reg (high 16 bits)
01100	`state`: control unit current state variable
01101	`call_stack_at_ptr_minus_one`: the item on the top of the state machine stack
01110	Low byte contains `call_stack_pointer`: the state machine stack pointer. High byte contains the ALU condition codes.
01111	Low byte contains `last_output`: the last byte to be written to the output device. High byte contains a bitfield composed of various internal flags.
1XXXX	RAM: the byte at address `0x100X`.

The debouncer is in the `button_debouncer` process. It consists of an up counter that counts from 0 to `0x8000` when the button is pressed or released. If the button stays stable for the time taken for this up count, then the state change is considered to be valid, and the `button_clock_event` register is sent high until cleared by another signal.

The VHDL source that manages the debouncer and `button_clock_event` is in `clock.vhd`.
⇒ *The source code of* `clock.vhd` *can be found in Section E.4, Page 79*

Removing debugging support

Debugging support can be removed from the processor by removing the processes that manage it. The clock controller may be replaced by one line of VHDL: "`clock <= fast_clock ;`", which drives the processor at the FPGA clock speed at all times. The debugging output multiplexer may be removed entirely.

14. Implementing control line sequences for 68020 instruction execution

In Section 12, a method of implementing the 68020 instruction set was described. The instruction set is first defined in terms of high level register transfers, with one transfer per clock cycle. If this cannot be done for every instruction, then it is done for a representative set of instructions. These register transfers are high level in the sense that, for example, if we wish to increment register D_0 by 5, we can specify just $D_0 \leftarrow D_0 + 5$. There is no need to worry that this will require the current value of D_0 to be loaded and then the output of the ALU to be

Table 8: Processor Stepping Mode

Switch Setting	Processor Speed
00	Not applicable - processor is reset
01	Pressing the button advances the processor to the next clock edge (so pressing it twice advances the processor by one clock cycle).
1X	FPGA board speed

stored in D_0. This only becomes important later.

From the high level register transfers, the implementor derives a minimal set of low level register transfers that can be used to implement the high level ones. Every register transfer in this set adds to the complexity of the logic that makes up the processor, because every one means an extra data link and an extra input to a multiplexer. The aim is to make the processor as small as possible, not as fast as possible. So all data links must be essential. None can be present purely to allow some register transfers to be parallelised when those transfers could be done one after another, although if it is possible to parallelise two transfers for some other reason, this should certainly be done.

Finally, from these register transfers, the implementor designs the VHDL required to implement the data links they require. This VHDL doesn't change - it is a fixed part of the processor - but it couldn't be designed in Section 13, because it can only be efficiently designed once the required register transfers are known.

14.1. Beginning to implement the 68020 instructions

Implementation began with a careful examination of the instruction set. The instructions were all chosen for a reason, and there is a logical structure to the set: the 68020 designers didn't want to make their task harder than necessary. Understanding this structure makes implementation far easier.

It was soon noticed that many of the instructions are very similar. For instance, the ADD and SUB instructions are almost identical. The data comes from the same places in both, and the result goes to the same place. The only difference is that ADD uses the ALU in "add" mode and SUB uses the ALU in "subtract" mode. What this means is that ADD and SUB can share the same state machine sequence. The only thing that the state machine must do to ensure correctness is to examine some bits in the instruction register to decide which ALU operation to apply.

Whenever two or more instructions are similar enough that sharing a state sequence is possible, those instructions are said to belong to the same *family*. Only one set of states needs to be written for each family, so finding as many families as possible is useful.

Table 9 lists the families of opcodes that appear on the 68020. These were found by looking at the instruction set and trying to find instructions that are similar to each other. The first column indicates which bits of the instruction register are used to distinguish between members of the family. The second column gives the opcodes in the family. As can be seen, many instructions fit into one of the 68020 opcode families.

14.2. Defining the high level register transfers that are required

It would be a very costly exercise to work out which register transfers are needed to support all the 68020 instructions. Fortunately, there is no need to work out register transfers for more than one member of each family: the operations are the same for every member of a family. Additionally, only a small number of instructions need to be investigated before it becomes very likely that no more required register transfers will be found. After all, the 68020 designers also wanted to avoid having too many different types of register transfer, for exactly the same reasons as this project does.

Table 9: 68020 opcode families

IR bits	List of family members
15-14	**ABCD, ADDX, NBCD, SBCD, SUBX**: Decimal and Binary arithmetic operations using the Extend flag.
14-12	**ADD, AND, EOR, OR, SUB**: Arithmetic operations using data registers.
14	**ADDA, SUBA**: Arithmetic operations using address registers.
11-9	**ADDI, ANDI, EORI, ORI, SUBI**: Arithmetic operations using immediate operands.
8	**ADDQ, SUBQ**: Arithmetic operations using very short immediate operands.
10-9/4-3	**ASL, ASR, LSL, LSR, ROL, ROR, ROXL, ROXR**: Shift/rotate operations.
11-8	**BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BRA, BVC, BVS**: Conditional branch operations.
7-6	**BCLR, BCHG, BSET, BTST**: Bit test/set operations.
11-9	**CLR, NEG, NEGX, NOT**: Single-operand arithmetic and logical operations.
11-8	**DBCC, DBCS, DBEQ, DBF, DBGE, DBGT, DBHI, DBLE, DBLS, DBLT, DBMI, DBNE, DBPL, DBT, DBVC, DBVS**: Decrement and branch on condition instructions.
11-8	**SCC, BDCS, SEQ, SF, SGE, SGT, SHI, SLE, SLS, SLT, SMI, SNE, SPL, ST, SVC, SVS**: Set according to condition: a byte is cleared or set according to the specified condition.
11-8	**TRAPCC, TRAPCS, TRAPEQ, TRAPF, TRAPGE, TRAPGT, TRAPHI, TRAPLE, TRAPLS, TRAPLT, TRAPMI, TRAPNE, TRAPPL, TRAPT, TRAPVC, TRAPVS**: Trap on Condition. A TRAP is generated if the condition is true.

To find the required register transfers, a set of opcodes were chosen. One opcode was chosen from each family in Table 9, except those families which will not be implemented[10]. From the arithmetic operations, the subtract operation was chosen. It is the only non-commutative operation: the only one where the order of the operands affects the result. So if it works correctly, it is certain that the other commutative members of its family will also work correctly.

Table 9 doesn't cover all the instruction types that the processor may need to execute. So some control operations were chosen: JMP, JSR, and RTS, along with some data transfer operations: MOVEQ, and MOVE. The high-level register transfers to implement these operations and the ones chosen from Table 9 were written down. They appear in Appendix C. For the moment, it is assumed that some way of loading and storing data at an effective address has been worked out. It is also assumed that subroutines to "Decode *DataSize*" and "Fetch Immediate Data" exist.

Having looked at these operations, which represent a cross-section of 68020 operations, the set of required register transfers is known. It is very unlikely that any other operations might exist that would demand new register transfers. It should be possible to define any others using a sequence of existing transfers.

The register transfers required were collated into Table 10. The same symbols used in Appendix C are used here, with the one exception that the • mark is now used in place of an arithmetic or logical operation, to show

[10]In Section 7.2, features that would be left out of the processor were discussed. These features included decimal arithmetic instructions (ABCD, SBCD, and NBCD), interrupts and traps (TRAPCC, etc.), and shifts (ASL, etc.). The bitfield operations (BCLR, etc.) cannot be implemented without these features either, so they are left out too.

that every register transfer involving the ALU can be used with any operation. Readers should note that x and y are the values of two 3 bit fields in the instruction register.

Table 10: Collated Register Transfers

$A_x \leftarrow A_x \bullet DataSize$	$A_y \leftarrow A_y \bullet 4$
$A_y \leftarrow A_y \bullet A_x \bullet ExtendFlag$	$A_y \leftarrow A_y \bullet DataSize$
$A_y \leftarrow A_y \bullet [EA]$	$D_x \leftarrow D_x \bullet 1$
$D_y \leftarrow D_y \bullet D_x \bullet ExtendFlag$	$D_y \leftarrow D_y \bullet [EA]$
$D_y \leftarrow instruction\_register(7..0)$	$PC \leftarrow EA$
$PC \leftarrow PC \bullet IDR$	$PC \leftarrow PC \bullet instruction\_register(7..0)$
$PC \leftarrow [A_y]$	$[A_y] \leftarrow PC$
$[DestEA] \leftarrow [SourceEA]$	$[EA] \leftarrow 0 \bullet [EA]$
$[EA] \leftarrow 0$	$[EA] \leftarrow [EA] \bullet D_y$
$[EA] \leftarrow [EA] \bullet IDR$	$[EA] \leftarrow [EA] \bullet y$
$[EA] \leftarrow$ `0xff`	

14.3. Thinking at a lower level

To date, the register transfers have been very high level descriptions of what is required. The descriptions have been a mixture of pseudo-code and register transfers that are not always directly possible, but are possible only through a series of operations. For example, consider $[A_y]$. This is shorthand for "the value stored in memory at location A_y". In order to get this value, the processor must fetch data from memory. This could take up to four fetches if the operand is a double word, as it is for $PC \leftarrow [A_y]$. So this operation must involve a temporary register to store $[A_y]$ as it is loaded.

Of course, before implementation is possible, all of these complicated high level register transfers and pseudo-code must become actual register transfers. In the next sections, ways to arrange this will be discussed.

Implementing effective address (EA) operations

Around half of the 68020 instructions have an effective address field. This field allows one of the instruction operands to be specified in a very flexible way: it can be a data register, or a memory location specified in an address register, and many more possibilities as defined by the 68020's addressing modes. The effective address field occupies the six least significant bits of the instruction word (with one exception[11]). The six bits consist of a three bit Mode specifier, which indicates the addressing mode to be used, and a three bit Register specifier, which gives a register to be used for the operation.

Typically, an operation will want to use the effective address for loading and storing data. Take, for example, the ADD instruction, which is specified in the instruction set description in [Motorola 1985] as $[EA] \leftarrow D_y + [EA]$. What this means is that the instruction should take the value specified by the effective address (EA), add it to a particular data register (D_y) which will be specified in the instruction word, and store the result at the effective address again. If the effective address is a data register, then the operation is very simply $D_x \leftarrow D_y + D_x$. But if the effective address is a memory location, then the situation is more complicated. A whole series of operations will be required:

- Decode the effective address to determine the absolute memory location it indicates. Store this address in a register.
- Load the data at that address into a register. This is the current value of the operand.

[11] The MOVE instruction has two effective address fields, one for the source, another for the destination. Fortunately, the methods that handle one of these fields can be reused to handle both.

- Do the operation, updating the value.
- Store the updated value at the absolute memory location.

Two new registers are needed: one to store the absolute memory location that the effective address decodes to, and another to store the value at that address. In this implementation, the former was called `operand_address` (OA), and the latter was called `operand_value` (OV). These names were arbitrary. Although it is clear from the description of the effective address system detailed in the manual that both must exist, the manual does not say what the makers of the 68020 called them.

$[EA]$, the label for an effective address value used in Table 10, can now be replaced with OV - the short name for the `operand_value` register. This means that OV is available on ALU Input A in place of $[EA]$. Similarly, EA can be replaced with OA - the short name for the `operand_address` register.

Since the effective address operations are needed for more than one instruction, it makes sense to put them all in subroutines. Subroutines are needed to:

1. Decode an effective address into an absolute memory location, to be stored in `operand_address`. Call this one `decode_ea`.
2. Load the data at the address specified in `operand_address` into `operand_value`. Call this one `load_operand_value`. The register transfer operation done by this subroutine is $OV \leftarrow [OA]$.
3. Store the data in `operand_value` at the address specified in `operand_address`. Call this one `store_operand_value`. The register transfer operation done by this subroutine is $[OA] \leftarrow OV$.

With these three subroutines defined, an instruction can use an effective address by calling `decode_ea`, and then `load_operand_value`. It then carries out the operation, using OV in place of $[EA]$. Finally, `store_operand_value` can be called to write the data back, if it has been updated. The high level register transfers that used $[EA]$ directly can be translated to low level register transfers that can actually be implemented: ones that obtain OV, use it, and write it back. Table 11 shows this translation for the ADD, SUB, OR, AND and EOR instructions (the same sequence supports all members of this family, because the operation required is decoded from the instruction word).

Table 11: Implementing High Level Register Transfers using Low Level Register Transfers (1)

High Level	**Low Level Sequence**

Decode *DataSize*
if $instruction\_register(8) = 0$, then
$\quad D_y \leftarrow D_y \bullet [EA]$
else
$\quad [EA] \leftarrow [EA] \bullet D_y$
	Decode *DataSize*
	CALL `decode_ea`
	CALL `load_operand_value`
	Decode *ALU_operation*
	if $instruction\_register(8) = 0$, then
	$\quad D_y \leftarrow D_y \bullet OV$
	else
	$\quad OV \leftarrow OV \bullet D_y$
	$\quad$ CALL `store_operand_value`

New register transfers needed for the effective address decoding subroutine

The effective address decoding subroutine, `decode_ea`, must determine a value for OA from the information in the Mode field of the instruction word. Each of the 16 addressing modes that address memory calculate the OA value differently.

The entire set was examined and evaluated at length, and it was found that certain modes would need additional registers in order to be implemented. However, as discussed in Section 7.2, the modes requiring extra hardware will be left out of the implementation.

So the only addressing modes that will be considered are those that can be implemented using the internal registers that are already available. Table 12 shows all the 68020 addressing modes that will be supported, with the low level register transfers that decode OA.

Table 12: Register transfers required to support 68020 addressing modes

Mode name	**EA field**	**How OA is obtained**
Data Register Direct	`000 XXX`	OA is undefined, because no memory access is involved here.
Address Register Direct	`001 XXX`	OA is undefined, see above.
Address Register Indirect	`010 XXX`	$OA \leftarrow A_x$
Address Register Indirect with postincrement	`011 XXX`	$OA \leftarrow A_x$ $A_x \leftarrow A_x + DataSize$
Address Register Indirect with predecrement	`100 XXX`	$A_x \leftarrow A_x - DataSize$ $OA \leftarrow A_x$
Address Register Indirect with Displacement	`101 XXX`	*OA is loaded from memory address PC* $OA \leftarrow OA + A_x$
Address Register Indirect with Index, Displacement	`110 XXX`	Omitted - requires extra hardware.
Memory Indirect Pre/Post Indexed	`110 XXX`	Omitted - requires extra hardware.
Absolute Short	`111 000`	*OA is loaded from memory address PC*
Absolute Long	`111 001`	*OA is loaded from memory address PC*
PC Indirect with Displacement	`111 010`	*OA is loaded from memory address PC, then:* $OA \leftarrow OA + PC$
PC Indirect with Index and Displacement	`111 011`	Omitted - requires extra hardware.
PC Memory Indirect Pre/Post Indexed	`111 011`	Omitted - requires extra hardware.
Immediate	`111 100`	$OA \leftarrow PC$ $PC \leftarrow PC + DataSize$

As a side note, it was mentioned earlier that addressing mode support would be modularised. This is done in the state machine sequence for `decode_ea`, but not by removing or changing the code. It is done using two optimisation functions. These functions are discussed in Section 15.3.

Using the OA and OV registers to implement the other operations

Whenever there is an operation which requires memory to be fetched or stored in memory, OA and OV can be used along with `load_operand_value` and `store_operand_value`. Table 13 shows a translation of such an operation from a high level register transfer (on the left) to a low level register transfer (on the right). All the register transfers seen in this table have appeared before.

The $PC \leftarrow OA$ operation needed for the JSR and JMP instructions can be implemented by routing OA through the ALU: $PC \leftarrow OA + 0$.

Table 13: Implementing High Level Register Transfers using Low Level Register Transfers (2)

High Level	**Low Level Sequence**
$PC \leftarrow [A_y]$ (for RTS)	$OA \leftarrow A_y$ CALL `load_operand_value` $PC \leftarrow OV$
$PC \leftarrow EA$ (for JMP and JSR)	`decode_ea` $PC \leftarrow OA$
$[A_y] \leftarrow PC$ (for JSR)	$OA \leftarrow A_y$ $OV \leftarrow PC$ CALL `store_operand_value` $PC \leftarrow OA$
$[DestEA] \leftarrow [SourceEA]$ (for MOVE)	CALL `decode_ea` for SourceEA CALL `load_operand_value` CALL `decode_ea` for DestEA CALL `store_operand_value`
$[EA] \leftarrow 0$ (for SCC)	CALL `decode_ea` CALL `store_operand_value`
$[EA] \leftarrow$`0xff` (for SCC)	CALL `decode_ea` $OV \leftarrow$`0xff` CALL `store_operand_value`

Register transfers needed to load and store operand values, and fetch immediate values

Table 14 shows the sequence of register transfers for `load_operand_value`, `fetch_immediate_data` and `store_operand_value`. To save space, the sequence is only shown for the store or fetch of a single word. $OV_{15..8}$ indicates that the fetch is taking place into bits 15 to 8 of OV - the most significant byte.

This is the first time it has been necessary to show which transfers happen in which clock cycles. When data is fetched from memory, a request for it must be put in one clock cycle before the data must be available. So MAR is programmed in the first clock cycle, and then the data is available in MDR in the second. The word CLOCK indicates a state boundary where execution waits for a clock cycle, a convention introduced for the state machine generator.

Choosing ALU data sources to support Register Transfers

Many of the transfers require the ALU. Data must be available from registers: A_x, D_x, A_y, D_y, OV, OA, PC, IDR, and from y, $instruction\_register(7..0)$, and $DataSize$, and also some miscellaneous immediate values: 0, `0xff`, and 4. Multiplexers could be added to the ALU to allow any of these sources to be sent to either ALU input, but this would be a waste of logic. It would be better to try to route each data source to only one of the ALU inputs. This would minimise the size of each input multiplexer.

Because the ALU supports a "reverse subtract" operation, any register transfer of the form $Out \leftarrow A \bullet B$ can always be replaced by $Out \leftarrow B \bullet A$, just by using the reversed subtraction operation instead of the regular subtraction operation. This is very useful, because it means that it is only necessary to consider which two sources must be available to the ALU for each operation: it isn't necessary to consider which inputs those sources can reach.

The problem of finding the minimal set of sources for each ALU input multiplexer is a graph colouring problem[12], with two colours: one for each ALU input. In the graph, the nodes are the data sources, and an

[12] The graph colouring problem is solved when each node in the graph has been assigned a different colour to all of its neighbours.

Table 14: Implementing load_operand_value, store_operand_value, and fetch_immediate_data

Subroutine	**Low Level Sequence**
Load OV	$MAR \leftarrow OA$ *Prepare to fetch 1st byte of OV* $OA \leftarrow OA + 1$ CLOCK $OV_{15..8} \leftarrow MDR$ *Store 1st byte of OV* $MAR \leftarrow OA$ *Prepare to fetch 2nd* $OA \leftarrow OA + 1$ CLOCK $OV_{7..0} \leftarrow MDR$ *Store 2nd byte*
Store OV	$MAR \leftarrow OA$ *Set address for store of 1st byte* $MDR \leftarrow OV_{15..8}$ *Set data for store of 1st byte* $OA \leftarrow OA + 1$ CLOCK $MAR \leftarrow OA$ *Set address for store of 2nd byte* $MDR \leftarrow OV_{7..0}$ *Set data for store of 2nd byte* $OA \leftarrow OA - 1$

Load IDR	$MAR \leftarrow PC$ *Prepare to fetch 1st byte of IDR*
	$PC \leftarrow PC + 1$
	CLOCK
	$IDR_{15..8} \leftarrow MDR$ *Store 1st byte of IDR*
	$MAR \leftarrow PC$ *Prepare to fetch 2nd byte*
	if restore_pc_after_immediate_fetch then
	$PC \leftarrow PC - 1$ *Restore the PC to the original value*
	else
	$PC \leftarrow PC + 1$
	CLOCK
	$IDR_{7..0} \leftarrow MDR$ *Store 2nd byte*

edge between two nodes indicates that those two sources are needed simultaneously. Once the graph has been 2-coloured, all sources of a particular colour will go to the same ALU input.

Figure 18 illustrates the (uncoloured) graph. Note that *instruction_register*(7..0) is referred to here as $IR(7..0)$. Note also that a lot of the sources have been consolidated into one node, called PGI, which stands for "processor generated immediate". All the short immediate values (0, `0xff`, 4, y, etc) can be represented by PGI, because only one of them is needed at once. It also means that only one multiplexer input will be needed to allow the ALU access to all of these short values. If 0, `0xff`, 4 and y were all directly connected to the ALU input, XST would generate a 32 bit multiplexer input for each. Since PGI allows them to be indirectly connected, the logic that this would waste is saved.

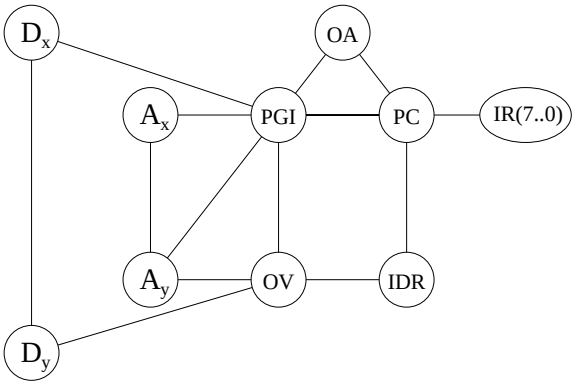


Figure 18: Data sources for ALU operations

Unfortunately, this graph cannot be 2-coloured. There is always at least one node that cannot be assigned either colour. This node must be shared between both ALU inputs. One minimal colouring is shown in Figure 19. The shading of the nodes indicates which ALU input they will be assigned to.

As can be seen, PGI is the node that has been chosen as the one that is present on both inputs. This is a particularly good colouring, because PGI is useful for all sorts of operations including those where a register value is routed through the ALU instead of being transferred directly. Suppose, for example, that we wish to transfer $PC \leftarrow IDR$. One way to do this is to do an ALU operation: $PC \leftarrow IDR + 0$. PGI is programmed to output zero, and when zero is added to IDR, the result is still IDR. This saves a data link between IDR and PC by reusing the link through the ALU. A zero must be available on both ALU inputs if this is to be generally possible. So sharing PGI makes perfect sense.

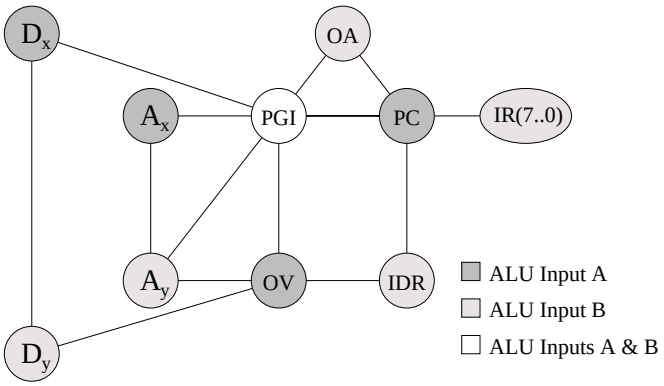


Figure 19: Data sources for ALU operations, assigned to an ALU input

What other transfer operations are required?

Now all the required data sources for the ALU have been determined. The next stage is to look at other types of transfer operation. Each of the registers in the processor has one or more inputs too.

Table 15 shows where each register can be loaded directly from, in a single operation without routing through any other component or register. For instance, the second line shows that OA may be loaded from the ALU output or memory (MDR). It has links to both of these.

Table 15: Links Between Registers

	ALU	OA	MDR	PC
$PC \leftarrow$	√			
$OA \leftarrow$	√		√	
$OV \leftarrow$	√		√	
$IR \leftarrow$			√	
$IDR \leftarrow$			√	
$MAR \leftarrow$		√		√
$A_x \leftarrow$	√			
$A_y \leftarrow$	√			
$D_x \leftarrow$	√			
$D_y \leftarrow$	√			

Clearly, the fewer of these links, the better. But they are all unavoidable, as will be explained. Each set of links will now be examined, in column order.

Table 15 shows that the destinations of the ALU operations can be A_x, D_x, A_y, D_y, OV, OA and PC. All of these are essential, because any of these can be the target of an arithmetic operation. Arithmetic operations may happen between any pair of data or address registers, so A_x, A_y, D_x and D_y are essential. OV is also essential: arithmetic operations take place on OV wherever an arithmetic operation takes place on an effective address. OA and PC are both needed so that they can be used to iterate through memory addresses: operations like $PC \leftarrow PC + 1$ are common during data fetches.

The destination of the OA and PC registers can only be MAR. These are needed so that OA and PC can both provide the address for memory accesses.

The destinations of the MDR register are the registers that memory can be fetched directly into. `load_operand_value` must be able to load values from memory into the OV register, and the instruction fetcher must be able to load instructions into IR. And immediate values must be loaded into IDR. Even the OA register needs to be loaded from memory, because memory addresses appear in extension words.

It is not really possible to leave out any of these transfers by loading memory into another register, and then transferring it afterwards, because all of these registers may be needed at the same time, and in any case this wouldn't save any data links.

DataSize and other supporting registers

In many of the tables in this section, and Appendix C, *DataSize* was used whenever the operation size (byte, word or double word) was important. Sometimes *DataSize* was assigned (as in BCC) and sometimes it was decoded from the instruction register in some way. *DataSize* was also tested in "if" statements and used to generate immediate values. But it was never explained what *DataSize* actually is.

DataSize is a register that contains the data size of the current instruction. Instructions on the 68020 are byte, word, or double-word sized. Quite often, this is specified by a code in bits 6 and 7 of the opcode. But this is not always the case. For instance, the CMPA operation may be word or long sized, and the size is specified by bit 8 of the opcode. So some way is needed to set *DataSize* from the instruction sequence.

To implement this, a control line called `operation_size_control` was added. It allows the operation size to be set directly to WORD, DWORD or BYTE. However, since a lot of operations use a size inferred from the instruction word, it also allows the operation size to be "SET_FROM_IR". This decodes the size from bits 6 and 7 of the instruction word.

⇒ *The source code of* `operation_size_control_process.vhd` *can be found in Section E.9, Page 86*

DataSize is a supporting register. Because it only needs to be set once per instruction, instruction sequences can handle situations where they must do something slightly different depending on the operation size. This is particularly useful in `load_operand_value` and `store_operand_value`. These need to load or store data of a different size depending on the operation size. They have no other way of knowing what the operation size should be, because they don't know which instruction is being executed.

There are two other supporting registers. One is `restore_pc_after_immediate_fetch`, the other is `ea_move_destination_control`. Both of these are single-bit registers.

`restore_pc_after_immediate_fetch` indicates to the `fetch_immediate_data` subroutine that it should put the value of *PC* back to what it was when the subroutine was called. There is one case where this is useful: in the branch instruction. Branches are specified relative to the end of the opcode, not the end of the immediate values following it. So it is vital that *PC* is restored before the branch is taken. Adding this supporting register allows `fetch_immediate_data` to do something slightly different in the one special case where this is required. If it was not present, two versions of `fetch_immediate_data` would be required.

When `ea_move_destination_control` is set, `decode_ea` uses the second effective address field instead of the first. A second effective address field appears only in the MOVE instruction, so again this allows a single special case to be handled without needing two copies of a subroutine.

How the ALU operation required by a member of a family is decoded

Table 11 describes the state machine sequence for one of the families shown in Table 9. It includes the line "Decode *ALU_operation*". In the case of the ADD, SUB, OR, AND and EOR family, bits 12 to 14 of the instruction word indicate which operation is required. Table 16 shows how these bits translate to each operation.

Table 16: Translation of instruction word to ALU operation, for ADD, SUB, OR, AND and EOR instructions

IR bits 14..12	Op	IR bits 14..12	Op
000	OR	100	AND
001	SUB	101	ADD
010		110	
011	EOR	111	

It was noticed that the same translations are also used by ADDA, SUBA, ADDX, and SUBX even though they are in a different family. This reuse suggests that it is not really a good idea to put a `case` statement in the state machine to select the ALU operation, because it must appear in more than one state.

Additionally, although it would be possible to implement the ADD instruction only as it is seen in Table 13, an optimised implementation might be preferred. For example, ADD operations on two data registers could be done in one clock cycle if this was handled as a special case within the processor. In this case, within one instruction, there would be two separate places where the ALU operation would need to be decoded.

Both of these factors indicate that decoding of ALU operations would be best placed outside of the state machine. And this is what was in fact done. The `alu_control_mux` process is notified of the current instruction family (e.g. ALU_I_FAMILY for ADDI, SUBI, etc) using the `alu_mode` control line. It then examines appropriate bits of the instruction register to decide which actual ALU operation should be used. It also decides whether the condition code register (which stores the carry, negative, zero and overflow flags) should be updated for the current operation.

`alu_mode` can also be assigned ALU_ADD or ALU_SUBTRACT for ALU operations that are done for internal reasons, such as $PC \leftarrow PC + 1$. The condition code register isn't updated for these operations.

By using this method of operation, any state machine sequence for a particular family can access the required ALU operation without needing any decoding of its own. A simple assignment to the `alu_mode` control line is

all that is required.
⇒ *The source code of* `alu_muxes.vhd` *can be found in Section E.2, Page 75*

How ConditionTrue works

In Appendix C, some operations featured a test for *ConditionTrue*. This test has not yet been explained. Many operations that test a condition code belong to families. All the branch and DBcc instructions are examples. This is because every instruction on the 68020 that tests condition codes does so in exactly the same way. A four bit pattern in bits 8 to 11 in the instruction word tells the processor which conditions to test.

It makes sense to put the condition test in a special process, so that the case statement that provides it does not appear in more than one place in the state machine. This special process examines bits 8 to 11 in the instruction word, does the test required by them, and sets a `condition_true` signal that can be tested within the state machine.

The condition tests are mostly arithmetic - greater than, greater than or equal, equal, and so on. Some test the overflow and carry flags. Usefully, [Motorola 1985] lists all the tests in a figure that is reproduced here (Figure 20). As can be seen, the table includes the bit pattern of each condition and the method for computing it, specified in terms of the C (carry), V (overflow), N (negative), and Z (zero) flags.

It is a simple matter to implement these in VHDL. It is made even easier by the realisation that one of the bits in the condition (the least significant) always inverts the result of the test.
⇒ *The source code of* `do_branch_process.vhd` *can be found in Section E.6, Page 82*

CC	carry clear	0100	$\overline{C}$
CS	carry set	0101	C
EQ	equal	0111	Z
F	never true	0001	0
GE	greater or equal	1100	$N \cdot V + \overline{N} \cdot \overline{V}$
GT	greater than	1110	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
HI	high	0010	$\overline{C} \cdot \overline{Z}$
LE	less or equal	1111	$Z + N \cdot \overline{V} + \overline{N} \cdot V$

LS	low or same	0011	$C + Z$
LT	less than	1101	$N \cdot \overline{V} + \overline{N} \cdot V$
MI	minus	1011	N
NE	not equal	0110	$\overline{Z}$
PL	plus	1010	$\overline{N}$
T	always true	0000	1
VC	overflow clear	1000	$\overline{V}$
VS	overflow set	1001	V

Figure 20: 68020 Condition Codes, from [Motorola 1985]

Instruction Fetch and Decode - Register Transfers for the Processor's Main Loop

Register transfers have now been designed for the execution of every instruction and every subroutine. The only missing operations are the ones that the processor must do before every instruction: fetch and decode. These are very simple, and easily implemented without adding any new register transfers. Table 17 illustrates the processor's main loop.

14.4. Implementing the Register Transfers in VHDL

At this point, register transfers for all the processor's operations have been designed. They need only be translated to VHDL, and the work will be complete. But before this can take place, the VHDL that actually provides the register transfers must be implemented.

There are three distinct parts to be implemented. A generator for the PGI data source is needed, along with the ALU input multiplexers, and finally, the actual register transfer process that manages all the registers. Each part is discussed below.

The processor generated immediate (PGI) data source

The PGI data source was required to generate a fairly substantial set of very short numbers. In the register transfers examined above, it was found that it would need to generate 0, 1, 2, 3 and 4. It would also need

Table 17: Register Transfers in the Main Loop

Initialisation: $PC \leftarrow 0$ *PC must be set to the program's start*
CLOCK

Fetch: $MAR \leftarrow PC$ *Prepare to fetch 1st byte of instruction*
$PC \leftarrow PC + 1$
CLOCK
$IR_{15..8} \leftarrow MDR$ *Store 1st byte of instruction*
$MAR \leftarrow PC$ *Prepare to fetch 2nd byte*
$PC \leftarrow PC + 1$
CLOCK
$IR_{7..0} \leftarrow MDR$ *Store 2nd byte*
$PC \leftarrow PC + 1$
CLOCK
Decode and Execute
CALL *instruction_decoder_output*
CLOCK
Then return to fetch..
JUMP $fetch$
CLOCK

to generate y - a number taken from bits 11 to 9 in the instruction word and used by ADDQ. Finally, it would need to generate a value based on the data width of the current operation for register transfers such as $A_x \leftarrow A_x + DataSize$. All of these numbers fit within 4 bits - the greatest is 8.

The use of a 4 bit multiplexer to select between these numbers saves making both the 32 bit ALU multiplexers much larger to be able to select from these numbers directly. So a 4 bit multiplexer process called `alu_input_muxes_2` was written in VHDL to generate the PGI value, based on a control line called `pgi_source`.
⇒ *The source code of* `alu_muxes.vhd` *can be found in Section E.2, Page 75*

The ALU input multiplexers

Both multiplexers were placed into a process called `alu_input_muxes`. Two control lines, `alu_source_a` and `alu_source_b`, were used to control the outputs of each multiplexer.
⇒ *The source code of* `alu_muxes.vhd` *can be found in Section E.2, Page 75*

The register transfer processes

One process, `register_transfers`, handles most data transfers between registers, including those from the ALU output to a register and those from memory to a register. The only three types of transfer that are not handled here are the transfer of OA or PC to the memory address register (handled by `memory_address_mux`), the transfer of the contents of OV to memory (handled by `memory_input_mux`), and the transfer of ALU data to A_x or D_x, which is discussed below.

Every register has one control line, which selects the source of the data for that register, or is set to an "UNCHANGED" setting if the register should be left alone. For instance, the PC register has a control line, `pc_source`, which can be set to either `ALU_TO_PC` (to load PC from the ALU output), or left as the default `PC_UNCHANGED`.

An alternative to this design was to have one or two control lines that control all register transfers. However, the implementor would then have the difficult problem of how to handle more than one register transfer in a single clock cycle. This is easy when there is one control line per register, but if control lines were shared between registers, the implementor would have to ensure that every transfer required appeared in the register transfer process.
⇒ *The source code of* `memory.vhd` *can be found in Section E.8, Page 83*

Register transfers to the address and data registers

As mentioned above, transfers to A_x and D_x were not handled by the register transfer process. Both these registers are actually implemented in block RAM, and data only reaches them from the ALU. So the control lines to load data into them are actually just the "write enable" lines of the block RAM. No extra process is needed to control this. The two lines used were `reg_update_address_x` and `reg_update_data_x`. By default, these lines are zero. When set to one, the value of A_x or D_x is updated to match the ALU output.

14.5. Implementing the state machine sequences for each instruction

Now a series of control lines have been defined to provide every register transfer required, translation of the low level register transfers to VHDL can begin. Table 18 shows some of the translations that are used.

Table 18: Translating Low Level Register Transfers to VHDL

Low Level	**VHDL**

$PC \leftarrow PC + 1$	`alu_input_a <= ALU_A_PC ;`
	`alu_input_b <= ALU_B_PGI ;`
	`alu_mode <= ALU_INT_ADD ;`
	`pc_source <= ALU_TO_PC ;`
	`pgi_source <= PGI_ONE ;`

$MAR \leftarrow OA$	`mar_source <= OA_TO_MAR ;`

$A_x \leftarrow A_x - 4$	`alu_input_a <= ALU_A_ADDRESS_X ;`
	`alu_input_b <= ALU_B_PGI ;`
	`alu_mode <= ALU_INT_SUBTRACT ;`
	`reg_update_address_x <= '1' ;`
	`pgi_source <= PGI_FOUR ;`

$IR_{15..8} \leftarrow MDR$	`ir_source <= MDR_TO_IR_1 ;`
$IR_{7..0} \leftarrow MDR$	`ir_source <= MDR_TO_IR_0 ;`

Every register transfer translates directly to one or more lines of VHDL, fitting within a single state machine state. ALU operations translate to four or five control line assignments. These set input A, input B, the operation required, and the destination. An additional line that may be set is the PGI source. All other operations take a single assignment. Using these translations, it is quite easy to turn the register transfers into VHDL. During translation, the implementor aimed to make as many things as possible happen in a single clock cycle: this improves the efficiency of the processor. Of course, only one ALU operation can take place at a time, but any other register transfers shown in Table 15 can take place simultaneously. So the typical state will do several register transfers and an ALU operation.

All the instructions discussed in this section were implemented by this translation process.

Support for each instruction or family was put into a separate file. Additionally, each subroutine was put into a separate file, and the processor's main loop was put into a file named `start.sm`. These files can be seen

in Appendix I. Putting each sequence into a separate file will allow the state machine generator to include only the sequences that are needed by the current program. Files of particular interest include:

- The main loop (`start.sm`) on page 170.
- `decode_ea` on page 155.
- `load_operand_value` on page 158.
- `store_operand_value` on page 160.
- `fetch_immediate_data` on page 162.

15. Implementing the generator

At this point, all the building blocks for the processor have been implemented. It is now time to implement the generator that will take the building blocks and produce a minimal processor from them, optimised to run just one application.

The generator's job is to produce a single VHDL file containing all the changeable parts of the processor. As discussed in Section 11.1, this should be done by reading through files provided by the user. All VHDL in these files always goes into the output file. But some special directives (those that appear in Table 5 on Page 26) are replaced by generated components: the state machine, the instruction decoder, and so on.

The generator, which is called the "State Machine Compiler" (SMC), was written in C++ in order to take advantage of STL[13]. The main procedure of the generator reads an initialisation file that tells it where to find the various files it needs: where the root VHDL input file is, where state machine sequences can be found, and where the opcode database is. Using this information, it creates an instance of the `Control` class, which provides all the functionality of the program.

`Control` reads in the root input file, scanning for input matching one of the directives in Table 5. Any input that doesn't match goes straight through to the output. Input that does match causes some type of generation process to take place, producing the state machine, instruction decoder or an optimisation function. These generation processes are explained on the following pages.

⇒ *The source code of* `control.cc` *can be found in Section G.3, Page 96*
⇒ *The source code of* `control.h` *can be found in Section G.4, Page 100*
⇒ *The source code of* `main.cc` *can be found in Section G.5, Page 101*

15.1. The state machine generator

The state machine is generated by the following process:

Loading Phase: First, load in all the state machine sequences from a directory specified in the generator's configuration. Each sequence will execute one instruction, provide a subroutine, or (in one case) the processor's main loop.

Requirements Phase: Then, use information from the program scanner to work out which sequences are needed, based on what instructions appear in the program.

Integration Phase: Integrate all the sequences into one state machine: the "Master State Machine".

Generation Phase: Turn the master state machine into VHDL: putting it into the output in place of the "`INSERT STATE MACHINE`" directive.

[13] The Standard Template Library (STL) contains generic code for managing sets, linked lists and trees

⇒ *The source code of* `state.cc` *can be found in Section G.16, Page 128*
⇒ *The source code of* `state.h` *can be found in Section G.17, Page 131*
⇒ *The source code of* `state_machine.cc` *can be found in Section G.18, Page 132*
⇒ *The source code of* `state_machine.h` *can be found in Section G.19, Page 138*
⇒ *The source code of* `state_machine_loader.cc` *can be found in Section G.20, Page 139*
⇒ *The source code of* `state_machine_loader.h` *can be found in Section G.21, Page 142*

The Loading Phase

The loading phase is very simple. An instance of the `State_Machine_Loader` class is created by the `Control` class. Then, a procedure named `Add_State_Machine_Directory` is called with the name of a directory containing state machine sequences. For every file that matches the glob pattern "`*.sm`", `Add_State_Machine_Directory` calls the procedure `Add_State_Machine`. This procedure arranges for an instance of the `State_Machine` class to be created, based on the file. This class reads the file into memory.

State machines are represented in memory as a linked list, in which each state is a separate list item. A state may include any number of commands, and these are also stored as a linked list. Commands within a state are either actual VHDL, used for control line assignments and conditional tests, or special state machine commands, such as `JUMP` and `RETURN` (as seen in Tables 2 and 3). The boundary between two states is denoted by the presence of the `CLOCK` directive, so named because it means "wait for a clock edge".

The `State` class is responsible for containing a single state, and the `Command` class holds a single command, and (optionally) its parameter. The parameter is typically the name of the label to be `JUMP`ed to or `CALL`ed. Each `State` may have zero or more labels assigned to it by the `LABEL` directive: any of these labels can be used in a `JUMP` or `CALL`.

The `State_Machine` class has some powerful features. Two of them are provided by the `Depends_On` and `Provides` functions, which both return sets of labels. `Provides` returns a set of all the labels that are defined within the state machine: in other words, a list of all the states within it that are accessible by `JUMP` or `CALL`.

`Depends_On` returns a list of all the labels that this state machine requires in order to execute, but does not actually define. This is a list of all of the labels that are not actually within the state machine, but are still accessed by `JUMP` or `CALL` - in other words, a list of subroutines that the state machine needs.

The Requirements Phase

The `Provides` function is used by `State_Machine_Loader` to make a mapping that associates each state label with the state machine sequence that provides it. In this way, when the `Require_Microsub` procedure is called with the name of a subroutine that is required, the `State_Machine_Loader` can immediately see which state machine sequence will be required to provide that subroutine, just by looking at the mapping.

For every instruction in the program, the program scanner is able to determine which state machine sequence will be required to execute it. As a result of this, `Require_Microsub` can be called with the label of each required state machine sequence. During these calls, `State_Machine_Loader` builds a set of required state machines called `required_machines`.

The Integration Phase

The integration phase takes place in the `Build_Master_Machine` function. All the required state machines in the list are moved into one master state machine sequence. This process begins by finding the "root state machine". This state machine contains the processor's main loop: it is in a file called `start.sm` and begins with the label "start". It is known that the first state in this state machine is the one that the processor must start in, because that state resets PC and begins the first instruction fetch.

`Incorporate_Sub_Machine` copies all the states from each required machine and adds them to the end of the destination machine. After this, the master state machine is *finalised*. Absolute state numbers are assigned to every state in the machine. The numbers are assigned sequentially from zero, so the state after a state numbered n is $n + 1$. Since every state is now identified by a number, the parameters of every `JUMP` and `CALL` can be translated from a name to an absolute state number. This is done in the Generation phase.

The Generation Phase

In the Generation phase, the master state machine is written out as a VHDL `case` statement. Every state is now labelled by number, although VHDL comments are generated to show the original name of each state (if any). The directives are translated into the VHDL control line assignments that they represent - see Section 13.1 for details of these.

This process is carried out by the `Compile_Machine` function, whenever "`INSERT STATE MACHINE`" is seen in the input VHDL. The work is done recursively: the `case` is generated in the `State_Machine` class, but each state within it is generated by the `State` class.

15.2. The instruction decoder generator

Section 8 explained the requirements for an optimised instruction decoder: that is, an instruction decoder capable of decoding only the opcodes that appear in a particular program.

The Opcode Database

The first requirement was that the valid bit patterns for all 68020 instructions should be specified in an "opcode database". The opcode database that was designed is very simple: a flat file in which each instruction is associated with an opcode bit pattern. Just one line is needed per 68020 instruction.

One way to build an opcode database would be to create a table with one entry for every possible bit pattern (there would be 2^{16} entries). The instruction that each bit pattern decoded to would be entered in the table. Unfortunately, although some instructions (e.g. RTS) have only one possible bit pattern, many have hundreds or even thousands. The worst is the MOVE instruction, which has 9000 possible bit patterns. It is very difficult for the implementor to enter the correct instruction in so many places, because it is so easy to enter it in a subtly incorrect place.

It is best if every instruction takes up only one line, because this makes the work of the implementor very easy. But how can a single line entry specify bit patterns so precisely that it is possible (for example) to distinguish between all the forms of the ADD instruction seen in Table 4 on Page 20?

After much thought, a method to specify the possible bit patterns for each instruction was designed. It is based on the realisation that the 68020 instruction bit patterns always consist of a number of fields, concatenated together. A "field" is a group of 1 or more bits, all grouped together in the opcode, which may have some rules constraining the value of those bits. The field that has been the subject of a lot of discussion is the 6 bit Effective Address (EA) field But there are other fields, such as the 2 bit Operation Size field, and the 3 bit Register Number field.

Figure 21 illustrates how one sort of ADD instruction (type 1) is composed of five fields. The first is fixed (1101), but the rest define which operation is required, and what it should operate on. Significantly, not all of these fields can take any value. The operation size field, for instance, cannot be "11".

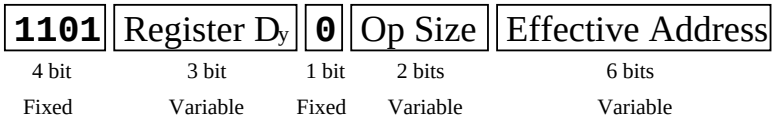


Figure 21: Fields in the ADD instruction (type 1)

Aside from those fields that have fixed values, like 1101 here, there are only a few different sorts of field. Every sort of field has fixed rules about what values it can have. For instance, no Operation Size field is ever "11", and no Effective Address field is ever "111110", because that pattern is reserved for future expansion.

So every instruction was defined in terms of fields. The implementor went through [Motorola 1985], looking at the definition of each instruction, and translating it to a set of fields. Whenever new field types were required, they were invented on the spot and the rules defining them were written down separately. They were also assigned a unique ASCII letter, to be used to describe them.

Every instruction bit pattern was described by a series of 16 ASCII letters and numbers, one for each bit. The value of a fixed field was specified directly by a binary sequence using "1" and "0". Variable fields were described by the ASCII letter assigned to them. So the ADD instruction above was described as shown in Figure 22.

```
1101 RRR 0SS EEEEEE
```

Figure 22: ADD instruction, as described in ASCII

The first 4 bits are described as the fixed field 1101. The next 3 bits are a register number field (RRR). Then, there is a 1 bit fixed field (0). Then, there is a 2 bit operation size field (SS) followed by the 6 bit effective address field (EEEEEE).

The opcode database consists of 131 lines that begin like this - one for every instruction. The line contains the name of the state machine sequence that executes the instruction, which is the label assigned to the first state in that sequence. It also features a comment, giving a short description of the instruction to aid the implementor's memory. The database was named `opcode_map`.

⇒ *The source code of* `opcode_map` *can be found in Section H.1, Page 145*

Scanning the opcode database to produce the instruction decoder

The opcode database is scanned by the `Opcode_Map_Reader` class.

⇒ *The source code of* `opcode_map_reader.cc` *can be found in Section G.10, Page 117*
⇒ *The source code of* `opcode_map_reader.h` *can be found in Section G.11, Page 121*

For each instruction, the `Read_Opcode_Map` function produces a description that indicates which bit patterns make up that instruction. The description takes the form of a directed acyclic graph. Every edge in the graph is a test of a particular field's value, by application of the rules for that field. Figure 23 illustrates this for ADD.

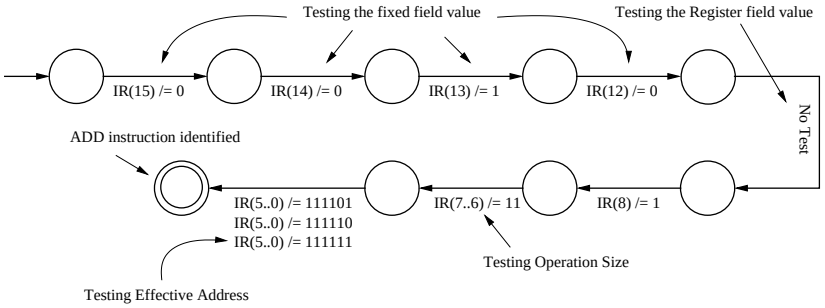


Figure 23: ADD instruction described as a graph

Note that all the rules in Figure 23 are of the form $field \neq value$. This is deliberate. It was noticed that all fields have more "allowed" values than "not allowed" values, so describing fields in terms of what values are not allowed makes the rules simpler.

Treating each instruction description as a finite state automaton

Readers who are familiar with automata may recognise the graph in Figure 23 as a finite state automaton. In fact, it is a deterministic finite state automaton (DFA[14]) that accepts exactly the instruction bit pattern, and

[14] A DFA is "executed" by following state transitions. Execution begins in the initial state, which is drawn with an arrow leading from nowhere pointing to it. At each state, execution follows the state transition whose condition is true. Usually, data is read from an input and compared to the value written by the transition. If they match, the transition

nothing else. However, because this DFA is also a tree (it has no cycles), it can be executed instantaneously: all the tests on every edge can be performed at once.

It was this observation that led the author to the realisation that certain aspects of automata theory could be used to produce a "master" DFA that decodes every 68020 instruction, with one accept state per instruction. This DFA could be built automatically from all the instruction descriptions, and the information gathered by the program scanner could be used to simplify it. This would have the great advantage that unused branches (ones leading to instructions that could never be executed) could be pruned. And tests that would always be true or always false could be eliminated. So the decoder could be generated and then simplified automatically.

To produce this master DFA, it would be necessary to merge all the instruction description DFAs into one DFA. On the surface, this is not a difficult problem. After all, the DFA is nothing but a series of nodes separated by edges representing decisions. To merge another DFA in, the program needs only to scan both DFAs from left to right and look for the first decision that differs, which is the merge point. Unfortunately, this doesn't work in general because not all instructions have the same fields. Look at Figure 24. The two instructions (ADD of type 1, and ADD of type 5, to use the numbering from Table 4) here have fields of different widths. When merging them, the merge program must find a way of distinguishing between the decision at (1), which is essentially "bits 7 to 6 are not 11" and the decision at (2), which is essentially "bit 7 is not 0". (Up until this test, the two are identical.) Merging will only work in the specific case where all the decisions are based on the same number of bits.

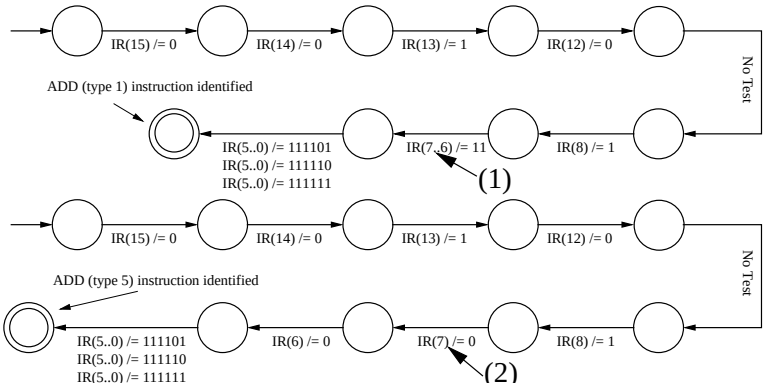


Figure 24: An example of two instruction decoding DFAs that are difficult to merge (ADD of types 1 and 5)

Fortunately, automata theory already provides a solution to this problem. A non-deterministic finite state automaton (NDFA) is much the same as a DFA, except that it is possible to have more than one possible choice at each node. It is possible for a particular input to lead to any number of NDFA nodes, whereas in a DFA a particular input always leads to just one node or rejection. Crucially, the same input can lead to both an accept and a reject state in an NDFA.

Another way to describe each instruction is to use an NDFA that accepts every possible bit pattern, but also rejects exactly those that the field rules do not allow. Figure 25 illustrates such an NDFA for the ADD instruction. All bit patterns are accepted - the path along the top of the figure will accept any 16 bit pattern. But some - those that are not valid ADD bit patterns - are also rejected (the rejecting nodes are marked with an R). For instance, if the most significant bit of IR is zero, the transition marked (3) will be taken. It leads directly to a reject state.

This is actually a far better way to describe each instruction. Every decision is based on testing one bit only. And, when all the decisions are based on testing one bit, merging is possible. Unfortunately, the instruction is now described by an NDFA, and since NDFAs are non-deterministic, they cannot be used to make the deterministic decoding decisions that are required.

is followed to the next state. Execution ends when an accepting state (drawn with a double circle) is reached, or when no transition can be followed. In the latter case, the DFA has rejected the input.

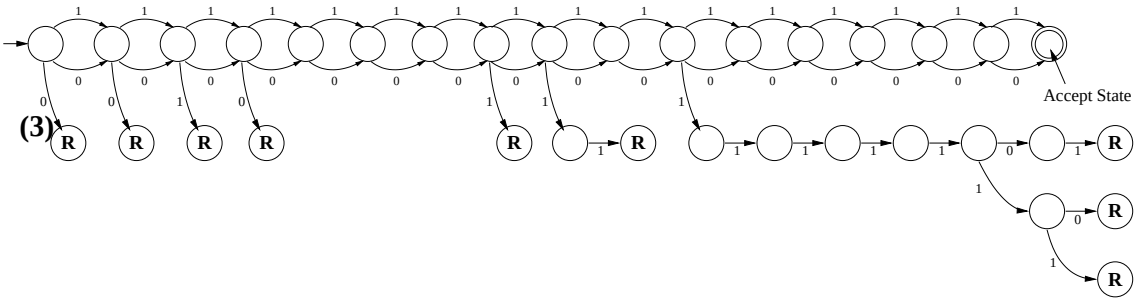


Figure 25: NDFA that accepts the ADD instruction

Automata theory provides the solution to this problem. There is an algorithm that can translate an NDFA into a DFA. It is possible to make any NDFA deterministic: the set of "languages" accepted by all NDFAs is exactly the same as the set of languages accepted by all DFAs. An equivalent DFA for Figure 25 produced using this algorithm is shown in Figure 26.

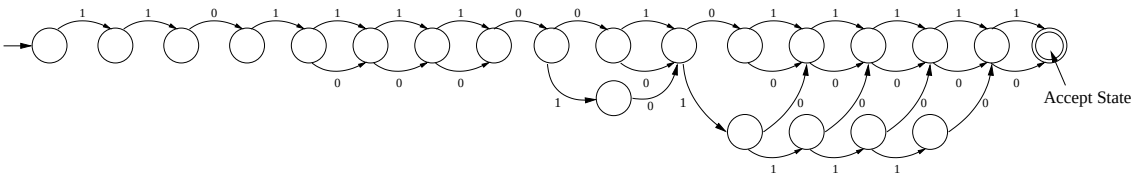


Figure 26: DFA that accepts the ADD instruction

The algorithm works with a recursive, i.e. self-calling, procedure called `Transform_NDFA_To_DFA`. It is called with two parameters. One is the current NDFA node, another is the current DFA node. The procedure is initially called with the initial state in the NDFA, and the initial state in a new, empty, DFA. Each DFA node is going to represent one or more NDFA nodes.

The procedure examines the transitions from the current NDFA node, which are divided into two types: those that are taken when the input is 1, and those that are taken when the input is 0. For each transition type x:

- Build a set S of all the NDFA nodes that could be reached by transition x from the current NDFA node.
- If S has at least one member, then a DFA node will be needed to represent transition x. If such a node does not exist, it is created. This node is called N, and it is made a successor of the current DFA node on transition x.
- If S contains a rejecting state, then N is marked as a rejecting node, regardless of whether it is marked as an accepting one.
- If S contains an accepting state, and N is not marked as a rejecting node, then N is marked as an accepting state.

Having done this, the successors of the current NDFA node are visited by a recursive call to `Transform_NDFA_To_DFA`. Thus, the whole NDFA is converted to a DFA.

The NDFA for each instruction is built as described above, and then translated into an equivalent DFA. The DFA accepts exactly the instruction, in much the same way as the one in Figure 23. But this time, every decision is done on a single bit. So it will be possible to merge this DFA with others to produce a master DFA.

DFA and NDFA classes

A class called `NDFA_DAG` was written to represent both NDFAs and DFAs (as a DFA is a special type of NDFA). It is essentially a container for the two: almost an abstract data type, but not a pure abstract data type due to its ability to generate VHDL to represent a DFA. The class `NDFA_Node` was used to represent a single node of the NDFA/DFA contained within `NDFA_DAG`.

⇒ *The source code of* `ndfa_dag.cc` *can be found in Section G.6, Page 102*
⇒ *The source code of* `ndfa_dag.h` *can be found in Section G.7, Page 105*
⇒ *The source code of* `ndfa_node.cc` *can be found in Section G.8, Page 105*
⇒ *The source code of* `ndfa_node.h` *can be found in Section G.9, Page 116*

The `Read_Opcode_Map` function creates a new NDFA for every instruction using this class. By default, this NDFA accepts every bit pattern. But calling the `Reject_Pattern` procedure causes a particular value of a particular field to be rejected. This means that the `Read_Opcode_Map` function can apply the rules it knows about each field to reject particular values of each field. By doing this, it ensures that the NDFA will reject all invalid bit patterns for the instruction. The result is similar to that seen in Figure 25.

This NDFA is then made deterministic (made into a DFA) by a call to `Make_Deterministic` which applies the algorithm discussed earlier by a call to `Transform_NDFA_To_DFA`. This produces a DFA that recognises exactly this instruction, like the one in Figure 26.

Producing the master DFA

Although the master DFA could be produced by merging the DFAs in the way described on Page 54, there is a simpler way. The best way to merge all the instruction description DFAs into a master DFA is to treat each of them as an NDFA. NDFAs can be freely merged, because a single input value can lead to more than one node. And as the algorithm to convert an NDFA to a DFA has already been written, it is easy to produce the master DFA from this.

Pruning the master DFA

It is important that the master DFA is as small and simple as possible, because every decision in it will be translated into a decision made by the instruction decoder. The process is called pruning, and it is easy because the `Make_Deterministic` procedure always makes a DFA that is also a tree. Therefore, there is only one path from the root to each node, and there are no cycles. Algorithms can thus work on the DFA recursively.

Pruning is possible when the program scanner has provided a list of all the opcodes that are required. For each opcode, the DFA is traversed. During traversal, every node is marked as "visited". When the accept state is reached, it is marked as "enabled".

During pruning, every unvisited node is removed by the `Delete_Dead_Branches` procedure, which recurses through the DFA. A dead branch is one that leads from a unvisited node, or leads only to rejection, or leads to an accept state that is disabled. `Delete_Dead_Branches` goes to the leaf nodes of the DFA tree, and while recursing back up the tree, it applies the rules for detecting dead branches, deleting any that it finds.

Next, the tree is "compressed". Compression deletes unnecessary decisions, and again works recursively. If a node only has one successor, then the node is redundant - having reached that node, it is certain that decoding will reach its successor (recall that the DFA doesn't need to detect invalid opcodes, because all opcodes are assumed to be valid). Equally, if a node has two successors, but both go to the same accept state, then either successor can replace the node, because only one accept state can be reached from this node.

Dead branch elimination and compression reduce the DFA to a very minimal tree. This tree contains the minimal number of decisions required to differentiate between the valid opcodes. Although there is no proof that it is a minimum (the smallest possible) tree, there is no doubt that it is very close to that. In experiments, it was found that the ratio of the number of opcodes to the number of decisions is typically one to one.

Generating the VHDL for the minimal decoder

VHDL is inserted into the output file in place of the "`INSERT INSTRUCTION DECODER`" directive. It is generated from the DFA recursively. Each node in the DFA is translated to a new `if` statement, testing an appropriate bit

of the instruction word. Each accept state is translated into a statement of the form "`decoded_state <= "`n`" ;`", where n is the number of the first state in the sequence that executes the instruction.

Figure 27 shows an output from the generator, produced for a small program with only three instructions: MOVE, ADDQ, and BRA. As can be seen, the three instructions can be differentiated in just three decisions.

Figure 27: Instruction Decoder for a very small program

```
instruction_decoder : process ( instruction_register ) is
    variable ir : word_register ;
begin
    ir := instruction_register ;
    if ir ( 14 ) = '1' then
      if ir ( 13 ) = '1' then
        decoded_state <= "010110" ; -- (branch)
      else --ir(13)= '0'
        decoded_state <= "000111" ; -- (alu_q_family)
      end if ;
    else --ir(14)= '0'
      decoded_state <= "001110" ; -- (move_family)
    end if ;
end process instruction_decoder ;
```

15.3. The ALU and Effective Address optimisers

Earlier, a method of modularising the ALU by restricting the control line values was discussed. This was to be done using an "optimisation function". The operation of such functions will now be examined. These functions can also be used to modularise the addressing modes that are available.

Optimisation functions are generated by the `Optimisation_Manager` class, which holds a list of objects to represent each type of optimisation. Figure 28 shows an example.

Figure 28: An example of an optimisation function, set up to allow only addition and subtraction within the ALU

```
subtype param_alu_internal_op is alu_internal_op_type ;
function apply_alu_internal_op ( i : in param_alu_internal_op )
                                        return param_alu_internal_op is
begin
  if ( i = ALU_INT_ADD ) or ( i = ALU_INT_REV_SUB ) or ( i = ALU_INT_SUB )
  then
    return i ;
  else
    return ALU_INT_ADD ;
  end if ;
end function apply_alu_internal_op ;
```

Wherever the ALU control lines are assigned, a call to the function is made:

```
alu_internal_op <= apply_alu_internal_op ( ALU_INT_ADD ) ;
```

In this case, the `ALU_INT_ADD` assignment is carried straight through to the ALU control line, `alu_internal_op`. When XST optimises the code, it sees that that the function will always return `ALU_INT_ADD`, so it replaces the assignment with `alu_internal_op <= ALU_INT_ADD`. On the other hand, assignments to (say) `ALU_INT_EOR` are not carried through. Since they will never be used, it doesn't matter what they are replaced with, so they are replaced with any valid operation that will be supported. In this case, they are replaced with `ALU_INT_ADD`. Then, XST can see that EOR will never be used, an eliminate it from the ALU.

Of course, this function needs to be automatically generated. To do this, a list of all the ALU operations and addressing modes that are required by the program must be built.

Determining which ALU operations are required

The 68020 will always need an ALU that supports addition and subtraction. These operations are essential, because register transfers such as $PC \leftarrow PC+1$ are always carried out by the processor, no matter what program is running. It is the other ALU operations (AND, OR, CMP and EOR) that may not always be essential.

These operations will only be used when one or more instruction is present in the program that needs them. Certain 68020 instructions are associated with particular ALU operations: for instance, the EORI instruction requires the EOR operation, and the ADDX instruction requires the ADD operation. The program scanner is able to produce a set of all the instructions used in the program, so what is needed is some way to associate each list item with the ALU operations it requires (if any).

The opcode database already has one entry per instruction, associating each opcode bit pattern with the state machine sequence that executes it. So this is a natural place to include the set of the ALU operations that each instruction requires. An extra field was added to the file to describe this set. Each ALU operation was assigned an ASCII character, according to the convention seen in C (see Table 19). Strictly, there is no real need to include the Add and Subtract operations since they are always needed. They are included anyway for completeness: if an instruction explicitly requires Add or Subtract, it can still be specified.

Table 19: ASCII codes for ALU operations

Code	Operation	Code	Operation
`+`	Add	`-`	Subtract
`&`	And	`|`	Or
`^`	EOR	`c`	Compare

The C++ generator software includes a set of `Optimisation` classes, of which `ALU_Optimisation` handles ALU optimisations. One instance of `ALU_Optimisation` is created, and tracks the ALU operations that are required. It holds a set of required operations, and may add to this set when the `Notify` procedure is called. `Notify` is called by `Optimisation_Manager` for every instruction found by the program scanner: once for every ASCII character found in the set of ALU operations for that instruction. It translates each character into an allowed control line value: `^` becomes `ALU_INT_EOR`, and so on.

⇒ *The source code of* `alu_optimisation.cc` *can be found in Section G.1, Page 96*
⇒ *The source code of* `alu_optimisation.h` *can be found in Section G.2, Page 96*
⇒ *The source code of* `optimisation.cc` *can be found in Section G.12, Page 122*
⇒ *The source code of* `optimisation.h` *can be found in Section G.13, Page 124*

Generating the optimisation function in VHDL

The `Generate_VHDL` procedure (inherited by `ALU_Optimisation` from `Basic_Optimisation`) produces the optimisation function from the list of allowed control line values. The function always takes the form seen earlier in Section 15.3, with variations only in the list of allowed control lines varies.

Addressing mode optimisations

Support for certain addressing modes can be removed in the same way as ALU operations were: by an optimisation function. In fact, most of the code is re-used. The optimisation function is used in a slightly different way, since it is called within the state machine.

The ALU optimisations were driven by per-instruction information about which opcodes were required. Addressing mode optimisations cannot work in the same way, because information about the addressing modes used can only be gathered by looking at the opcodes present in the program, not by looking at the instructions.

Every opcode using an addressing mode is examined. The three bit Mode and three bit Register specifiers that make up the effective address field are extracted. Section 11 has information about the function of these

subfields. Sets are built up, containing the values that may be seen in these subfields. These are then used to generate two optimisation functions, one for each field.

The `EA_Optimisation` and `EA_Reg_Optimisation` and class handle addressing mode optimisations. In each, the `Notify` procedure is responsible for adding to the set of required addressing modes. It is called for every instruction found by the program scanner, and if a particular instruction includes an addressing mode, the mode used is examined. The bit pattern is extracted and added to a list of addressing modes that may appear. This works in essentially the same way as `Notify` in `ALU_Optimisation`, except that information about the required modes is gathered from the opcode and not from the information in the opcode database.

`EA_Optimisation` maintains the set of possible values of the Mode field, and `EA_Reg_Optimisation` maintains the set of possible values of the Register field.

Using the addressing mode optimisation functions

The function generated by `EA_Optimisation` is called `apply_ea_mode`. It works in exactly the same way as the ALU optimisation function, limiting its return value to the input values that are possible for the current program. It is used in the effective address decoder `case` statement, where it restricts the selection:

```
case apply_ea_mode ( ea_mode ) ( 2 downto 0 ) is
when "010" => -- Address Register Indirect mode:
    ...
```

`apply_ea_reg` is generated by `EA_Reg_Optimisation`. It works in the same way:

```
when "111" =>
    case apply_ea_reg ( ea_reg ) ( 2 downto 0 ) is
    when "000" =>  -- Absolute address (Word mode)
        ...
```

XST will recognise that some of the `case` statement choices will never be taken. The logic that represents these can be eliminated.

15.4. The program scanner

The only part of the generator that has not yet been described is the program scanner. The program scanner is a very simple component because it obtains its list of instructions from GNU `objdump`, as discussed in Section 11.3.

The scanner is found in a procedure called `Require_Opcodes_In_File`, which reads in a file output by `objdump`. It reads every line that matches a particular regular expression: a regular expression that finds the first line describing each instruction, as instructions may be split over several lines if they have many extension words. From this line, the regular expression extracts the opcode of the instruction. For example, here is the output of `objdump` for a very small program.

```
a.out:     file format ihex

Disassembly of section .sec1:

00000000 <.sec1>:
   0: 203c 0a0f 0a09  movel #168757769,%d0
   6: 23c0 0000 8000  movel %d0,0x8000
   c: 5280            addql #1,%d0
   e: 6000 fff6       braw 0x6
...
```

From this file, the scanner would read opcodes `0x203c`, `0x23c0`, `0x5280` and `0x6000`. The extension words are irrelevant and are discarded. The names of the instructions that `objdump` has identified are also unimportant, because the instruction decoder can be used to identify each opcode. So no attempt is made to parse the instruction names: only the opcode is read.

Both the `Optimisation_Manager` class and the `Opcode_Database` class are informed of each opcode that is read. Each of these classes decode the opcode. In the case of the `Opcode_Database` class, the opcode is registered as being present and nodes in the instruction decoder DFA are marked as "visited". In the case of the `Optimisation_Manager` class, information is found about the optimisations that are relevant to the opcode.

Part V.
Evaluation and Conclusion

16. Evaluation

Implementation is now complete, so the project work will be evaluated. Evaluation will look at five aspects of the work:

Does the State Machine Compiler work? The tests used to verify the correctness of the state machine compiler will be discussed.

Does the processor work? This will involve some tests to verify that the processor runs programs correctly.

How much FPGA space does it take up? One requirement for the processor was that it should take a minimal number of logic gates. This will be evaluated in various conditions.

How does it compare to other soft processor cores? The processor will be evaluated against some contemporary cores.

How extensible is the processor? Is it possible to add new instructions?

16.1. Does the State Machine Compiler work?

The main proof of the operation of the State Machine Compiler (SMC) is that it generates correct VHDL that can be synthesised into a working processor. However, various other tests were also used to check that it operates correctly. SMC has the following features to aid testing:

- Assertions are widely used throughout the SMC. The C++ `assert()` macro is used to test over 50 conditions during execution of the program. In this way, SMC partly tests itself.

  An excerpt from the `NDFA_Node` class appears in Figure 29, illustrating one such assertion. Here, the assertion checks that a node marked as an accepting state has an attached `Accept_State` object. This will always be the case, unless some other code has incorrectly marked a state as an accepting state.

- SMC has a debugging setting in which plenty of information about its operations is printed out. This allows the tester to check through SMC's operations. In particular, it is possible to check that the instruction decoding DFA is produced and compressed correctly, because it is printed out as a tree.

These features allowed the tester to try various inputs to SMC and check for correct behaviour. The state machine generator and instruction decoder generator were essentially tested separately. The state machine generator was tested with various incorrect inputs: duplicated labels, JUMPs to non-existent inputs, empty states, VHDL after the final CLOCK statement, lack of a root state machine, and many more.

The instruction decoder generator was tested during its development. A special test module was written that allowed the user to input an opcode. The opcode would then be decoded using the DFA. This module was extended so that it only supported a user-defined set of opcodes. It was then extended again to add support for compressing the tree. Having tested the decoder with many different opcodes, the tester gained a high degree of confidence in its correct operation.

Figure 29: An assertion in NDFA_Node

```
void NDFA_Node :: Print_NDFA_DAG ( bool with_pointers , FILE * fd ,
                                   const char * old_tab_str )
{
    if ( is_accept_state )
    {
        assert ( accept_state != 0L ) ;

        accept_state -> Print_Info ( fd , old_tab_str ) ;
```

16.2. Does the processor work?

In total, 23 different types of instruction were implemented on the processor. Because some of these types are families, 32 actual instructions are available. Table 20 lists them. These instructions are enough to run a lot of small programs, written in C or assembly.

Table 20: The 32 Implemented Instructions

CMPA	SUBA	ADDA	CMPI	ORI	ANDI	SUBI	ADDI
EORI	CMP	OR	SUB	EOR	AND	ADD	ADDQ
SUBQ	Bcc	CLR	DBcc	JMP	JSR	LEA	LINK
MOVE	MOVEQ	NOP	PEA	RTS	Scc	TST	UNLK

To demonstrate that the processor works correctly, a variety of different programs were tested on it. The first programs to be tested were very short assembly programs that output a number sequence. Page 59 features an example of such a program. However, proper tests require more realistically large programs.

Testing with a C program

One aim of the project was to allow C programs to be run. So one test focuses on the processor's ability to execute a C program. `fib.c` is a Fibonacci sequence generator written in C.
⇒ *The source code of* `fib.c` *can be found in Section F.1, Page 91*

The sequence generator outputs the Fibonacci sequence, sending the numbers to the display: 0, 1, 1, 2, 3, 5, etc. The processor and program are known to be operating correctly when this sequence is seen on the output. The sequence quickly reaches the limit of the display (255), but even when it has gone past this point, the operation of the processor can still be checked by comparing the sequence to that seen on the output of the vm68k emulator.

The Fibonacci program worked perfectly. This demonstrates that the processor is capable of running a C program correctly, which in itself demonstrates that a large number of instructions are implemented properly.

Complete Functional Verification Test

But the Fibonacci program doesn't really demonstrate that all instructions are correctly implemented: they just work well enough in that case. The `fvt.s` assembly program attempts to test every instruction sufficiently to give a high degree of confidence in the correct operation of the processor. Particular attention was paid to instructions that were not thought to be well tested in the Fibonacci C program.
⇒ *The source code of* `fvt.s` *can be found in Section F.2, Page 91*

The program assumes that the MOVE and CMP instructions work correctly - if they do not, one of the first tests will fail anyway. `fvt.s` runs an instruction and then tests that it has affected the processor and memory

in the way that was expected. For instance, the LINK instruction modifies a register value, a location in RAM, and the stack pointer. All three outputs are tested for correctness by `fvt.s`.

`fvt.s` also tests UNLK, JSR, RTS, JMP, DBcc, CLR, Scc, SUBA, LEA and PEA. These are not well tested by `fib.c`. The program outputs a number to the display according to the number of the test it is running. If a test fails, then the program will stop with the failed test number on the output display. However, if all tests pass, a success code appears on the display. Thus, `fvt.s` demonstrates that the processor is able to run all of the instructions. The program ran all the instructions correctly, repeating the test ten times before displaying the success code.

16.3. How much FPGA space does the processor take up?

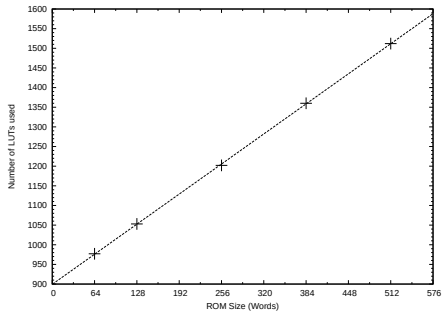
The FPGA space taken up by a particular design is measured by XST in "slices" and "4 input lookup tables". As described in Section 1.2, the lookup tables are used to generate logical functions. There are two of them per slice, and two slices per configurable logic block. FPGA space can be talked about in terms of either of these: but it is best to use the number of lookup tables since these are always more of these than there are slices. XST prints information about the number of slices and lookup tables (LUTs) during synthesis to indicate the amount of FPGA space used by a design.

In the test setup, the processor's size depends not only on the actual size of the processing logic, but also on the size of the debugging hardware and the ROM, because those parts are synthesised at the same time as the processor. Obviously, it would be incorrect to count those parts, so their effect on the amount the synthesised logic will be examined first.

The amount of space taken up by ROM

Figure 30 is a graph relating the number of words of ROM used to the number of LUTs on the FPGA. The processor was built with support for only one instruction and with debugging support removed. Programs of size 64, 128, 256, 384 and 512 words were used. As might be expected, there is a linear relationship between the amount of ROM used and the number of LUTs, and each LUT holds, on average, 0.84 words of ROM.

Figure 30: Graph of the number of LUTs used to represent ROM



The amount of space taken up by debugging hardware

The processor was generated with support for a few instructions and a small ROM. Without changing this configuration, it was built with and without the debugging hardware. The debugging hardware increases the number of LUTs required from 967 to 1178: it requires an additional 211 LUTs.

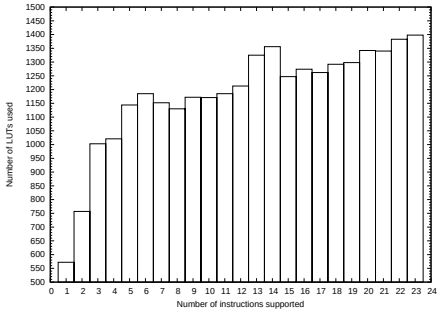
The effect of the number of supported instructions on the processor size

The project aimed to create a processor that was ideally suited to running a particular program, and was an

optimal size for that program. It was intended that this should primarily be achieved by omitting support for instructions that are not required.

The processor's size is certainly strongly affected by the number of instructions that are required for a program. Figure 31 shows how the processor size is affected by the number of different instructions in the program. A very short program called `23instructions.s` was written containing one of each of the 23 instructions that were implemented. The program does nothing: it just forces the processor to synthesise support for all instructions. After synthesis, the last instruction was removed and the processor was re-synthesised for the new program. This continued until all instructions had been removed.

Figure 31: Graph of processor size against program complexity



⇒ *The source code of* `23instructions.s` *can be found in Section F.3, Page 95*

As can be seen, adding support for an instruction makes the processor bigger, or approximately the same size. The relationship between the instructions supported and the processor size is far from linear. As the number of instructions present increases, the amount of extra hardware required for each gets smaller. There is a big difference between the space required by 1, 2 and 3 instructions. But there is little difference between the space required by 22 and 23 instructions. This is unfortunate. The instructions are sharing hardware with each other, so the total amount of hardware required changes less as the number of instructions increases.

It is expected that a program of any practical size will use at least half the implemented instructions. Experiments showed that the Fibonacci C program used 18 instruction types, and the processor and ROM for it took up 1440 LUTs. Had it used all 23 implemented instructions, it would have used up 1441 LUTs. Just as in Figure 31, the extra instructions have very little effect on the overall size of the processor.

The size of the processor can be reduced by leaving out support for instructions, but the effect is almost negligible if the program contains more than about 3 different types of instruction. Since any practical program would have many more types of instruction than that, not much space is actually saved by modular instruction support.

The effect of the number of ALU operations used

Leaving out operations in the ALU does have a substantial effect on the processor size. Processors were generated for two programs. The programs were the same size, so the same amount of ROM was generated, but the first program consisted only of ADD instructions, and the second was a mix of EOR, OR, AND, ADD and SUB instructions. All of these instructions use the same state machine sequence (they belong to one family), so the programs had identical state machines. Only the ALU differed between the processors.

The ALU optimiser removed support for EOR, OR and AND from the ALU for the first processor. This made the entire processor and ROM use 1202 LUTs. The second processor had a full featured ALU. The second processor and ROM used 1363 LUTs. 161 extra LUTs were required for three ALU operations.

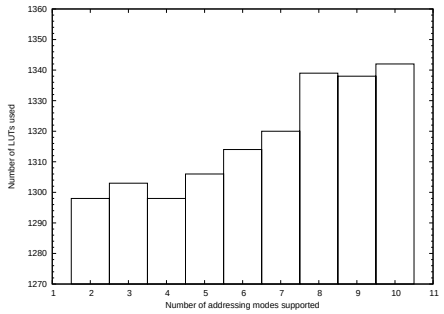
It is possible that some large programs might leave out one or two ALU operations. The Fibonacci C program used only ADD, SUB and AND, so some space would be saved by leaving out EOR and OR support. But most

large programs would probably need all ALU operations.

The effect of the number of addressing modes used

Leaving out addressing modes does have some effect on the processor's size. A series of test versions of the processor were built, for the same program, but with support for a varying number of addressing modes. Addressing modes were removed in the reverse of the order seen in Table 12. The graph in Figure 32 was drawn to show the effect of removing these modes on the processor's size.

Figure 32: Graph of processor size against addressing mode support



As can be seen, adding support for addressing modes has only a small effect on the size of the processor. Adding all modes only adds 40 extra LUTs. This difference is not particularly significant: it accounts for only around 3% of the size of the processor. This is because removing the modes only affects a few states in the state machine. It does not allow large pieces of hardware to be removed, because all the hardware used for addressing mode support is used elsewhere too.

16.4. How does it compare to other soft processor cores?

In order to compare the processor fairly to other soft processor cores, the memory map, RAM, ROM and debugging support were removed. The processor was made into a VHDL entity with only an interface to the memory. It is important that the cores are compared fairly, so all must have a similar interface. Here is the 68020 clone's interface:

```
port ( clock : in std_logic ;
       memory_address : out dword_register ;
       memory_output : in byte_register ;
       memory_input : out byte_register ;
       memory_write_enable : out std_logic ;
       reset : in std_logic ) ;
```

To further ensure a fair comparison, the 68020 was built with support for all the 23 instruction types that were implemented. The other processors are not modular: they are always complete, so comparisons can only be made between complete processors.

The 68020 clone was first compared to a processor called MyRisc [Wallander 1998]. MyRisc is a clone of the 32 bit MIPS processor. It is a complete processor, and being a 32 bit processor it has a similar ALU to the one used in this project. However, since it is RISC, it has a much smaller control line sequencer than the 68020 clone.

The 68020 clone was then compared to a processor called T80 [Wallner 2002], a Z80 clone. This is actually the only other CISC soft core that could be found. The control line sequencer is quite complex, but the ALU is far simpler: it is only 8 bits wide.

Table 21 compares features of the three processors.

As can be seen in Table 21, the 68020 compares quite favourably to MyRisc - the two have a similar size. It compares very well to T80: it is about half T80's size.

Table 21: Soft Processor Core Comparison

Processor	Type	Width	LUTs
68020 clone	CISC	32 bit	1303
MyRisc	RISC	32 bit	1395
T80	CISC	8 bit	2241

16.5. How extensible is the processor?

It is quite easy to add support for new instructions to the processor. Doing so is just a matter of adding a new state machine sequence, and updating the opcode map with details of the new instruction. There are plenty of examples available to guide an attempt at doing this. The user of the processor could implement custom instructions to meet his or her needs. These instructions could easily re-use the ALU and existing register transfers, or could use extra hardware integrated into the processor.

For example, the 68020 logical shift and rotate instructions could be implemented by adding a "shifter" process, and a new state machine sequence for logical shifts and rotates. One way to do this would be to make OV into a shift register: a register with shift capability. It would have new control lines, which would be programmed by the new state machine sequence.

16.6. Summary of the Evaluation

Tests showed that the processor is able to correctly execute all the 23 instruction types that were implemented. Tests also demonstrate that the processor's size varies according to the number of instructions required, the number of ALU operation types needed, and the number of addressing modes needed. In other words, the processor works and is generated as expected.

Unfortunately, the modularisation doesn't work as well as was hoped. There is little difference between a processor that supports many of the instructions and a processor that supports all of them. Unless the program is short, the processor will always have support for most instructions, and consequently will always be about the same size.

Nevertheless, even when support for all instructions and all ALU operations has been enabled, the processor is still quite small when compared to other soft processor cores. This is probably because only a subset of the 68020 instructions were implemented. A lot of less useful 68020 features were omitted.

17. Conclusion

This project has demonstrated that a clone of the 68020 can be implemented on an FPGA, supporting a subset of the 68020 features. It also showed that the features could be tailored to the program in four areas. The instruction decoder, control sequencer, ALU and addressing mode support could all be optimised to match the program. However, it was found that the optimisation doesn't work as well as expected.

The processor compares well to other soft processor cores in terms of its usage of the FPGA, but it should be remembered that it doesn't support all the 68020 features even when all features are included. The processor runs at up to 10MHz according to XST, and executes each instructions in five or more clock cycles.

A lot of things could be done better if the project was attempted again. Choosing to implement the registers using Block RAM was a mistake, because read accesses to the registers took one clock cycle. Values could not be looked up immediately, and this meant that a lot of extra work was needed to update registers. It was thought that this approach would save FPGA logic, and this is true, but it doesn't save very much at all. The registers would have been better implemented using asynchronous RAM, as is done in MyRisc.

It was also a mistake to put the RAM, ROM and memory mapper in the same VHDL entity as the rest of the processor. This was done to make the processor easier to generate, but it makes it more difficult to reuse the

processor in other applications with different memory configurations.

The `operation_size` control line should really have been set by the instruction decoder, and not by the state machine sequence. A lot of instructions had to have initial setup states where this line was programmed, wasting space in the state machine and a clock cycle.

A feature to change the width of the address registers was discussed in the Design section, but never implemented due to a lack of time. This feature could have allowed some more space to be saved on the FPGA.

The main problem with the processor is that large programs tend to use so many different features that most of the features are always needed. There is little to be gained by removing the few that aren't needed. Future work might look into the best way to avoid this. There are two main possibilities:

- Suppose that SMC had access to a profile of the program as well as a list of the opcodes within it. It could then replace an infrequently used opcode with a series of opcodes that do the same thing, but use less hardware. For instance, the LINK opcode could be replaced by a series of MOVEs. SMC would have to weigh up the advantages and disadvantages of including hardware support for each instruction or implementing it in software: a tricky problem because of the number of instructions involved.
- There might well be advantages to defining the state machine sequences in a register transfer language instead of VHDL. The obvious advantage is that the sequences would be described at a higher level, and would thus be easier to understand. Fewer mistakes would be made in writing them. But SMC would be able to understand the operations in each state without parsing VHDL. It would be able to remove unnecessary decisions, just as it did in the instruction decoder. Every `if` or `case` could be optimised. Furthermore, optimisations like those used in a compiler could be employed. Subroutines could be inlined, or removed if never called. A better optimised state machine could be much smaller.

In conclusion, the project has demonstrated that a modular processor can be built, but much more research is needed to make the modularity useful to larger programs.

Part VI. Appendices

A. Bibliography

References

[Ashenden 1998] P. Ashenden, *The Student's Guide to VHDL*, Morgan-Kaufmann (1998)

[Hennessy 1996] J. Hennessy, D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan-Kaufmann (1996)

[Herveille 2002] R. Herveille et al, *Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Revision B.3, (2002) *http://www.opencores.org/wishbone/*

[Motorola 1985] Motorola Inc, *MC68020 32-bit Microprocessor User's Manual, Second Edition*, Prentice-Hall (1985)

[Ponder 2001] J. Ponder, *Generator, a Sega MegaDrive/Genesis emulator*, (2001) *http://www.squish.net/generator/*

[Sasayama 2001] K. Sasayama, *Libvm68k, an M68000 virtual machine library written in C++*, (2001) *http://www.hypercore.co.jp/*

[Tredennick 1988] N. Tredennick, *Experiences in Commercial VLSI Microprocessor Design*, Microproc Microsyst 12n8: 419-432 (1988)

[Wallander 1998] A. Wallander, *MYRISC - A VHDL Implementation of a MIPS*, Luleå University of Technology (2000) *http://www.ludd.luth.se/~walle/projects/myrisc/*

[Wallner 2002] D. Wallner, *T80 - a Z80 microprocessor soft core*, (2002) *http://www.opencores.org/cvsweb.shtml/t80/*

[Xilinx 2002] Xilinx Inc., *Spartan-IIE FPGA Family documentation* (2002) *http://www.xilinx.com/*

B. Building-Block Hardware Components that appear in Diagrams

B.1. Multiplexers

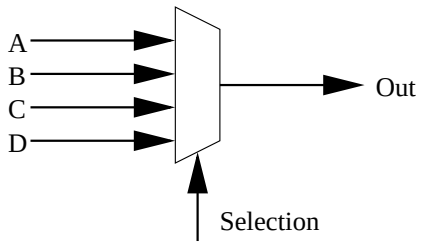


Figure 33: Multiplexer

Figure 33 shows a 4-to-1 multiplexer, where one of the four input lines (A, B, C or D) is selected by the "Selection" input and sent to the output: "Out". "Multiplexer" is commonly abbreviated to "Mux".

Multiplexers used in processors typically act on many data lines at the same time. A 32-bit 2-to-1 multiplexer, for example, connects one of two sets of 32 data lines on the input to the output. They allow a data source to be selected from more than one source.

Please note that in some diagrams, the Selection input is not shown. This is because the Selection input is a control line, and those diagrams omit control lines for clarity.

Here is the VHDL for a typical 16 bit 4-to-1 multiplexer:

```
    ...
    signal input_a      : std_logic_vector ( 15 downto 0 ) ; -- 16 bits
    signal input_b      : std_logic_vector ( 15 downto 0 ) ;
    signal input_c      : std_logic_vector ( 15 downto 0 ) ;
    signal input_d      : std_logic_vector ( 15 downto 0 ) ;
    signal output       : std_logic_vector ( 15 downto 0 ) ;
    signal mux_select   : std_logic_vector ( 1 downto 0 ) ;  -- 2 bits
begin
    ...
    process ( input_a , input_b , input_c , input_d , mux_select ) is
    begin
        case mux_select is
        when "00" =>        output <= input_a ;
        when "01" =>        output <= input_b ;
        when "10" =>        output <= input_c ;
        when others => -- i.e. "11"
                            output <= input_d ;
        end case ;
    end process ;
    ...
```

B.2. Links between Components

These are shown as lines connecting components in each diagram. Sometimes a link will be wider than one bit: in this case, the number of bits making up its width will be written in brackets somewhere along the line. The thickness of the line will also illustrate the number of bits, with thicker lines indicating a wider link.

B.3. Registers

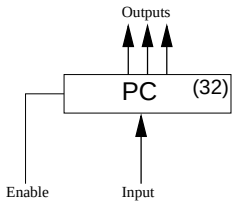


Figure 34: Register

Figure 34 shows a 32 bit register. Registers are a simple type of memory, provided by D-type flip flops. They store the data that is present on the input, if the "Enable" input is set, on a clock edge (i.e. during a transition from one clock state to the other). A 32 bit register stores the data present on 32 data lines.

Registers only have one input: if the register may have several sources, a multiplexer is essential. They may have any number of outputs. The stored data is always available on the outputs: no "fetch" is required, as there is no address. Some of the diagrams do not illustrate an Enable input. This is because the Enable input is a control line, and those diagrams omit control lines for clarity.

Here is the VHDL for a typical 32 bit register:

```
    ...
    signal input        : std_logic_vector ( 31 downto 0 ) ; -- 32 bits
    signal output       : std_logic_vector ( 31 downto 0 ) ;
    signal enable       : std_logic ;
begin
    ...
    process ( input , enable , clock ) is
    begin
        if ( clock = '1' )
        and ( clock'event )
        and ( enable = '1' )
        then
            output <= input ;
        end if ;
    end process ;
    ...
```

C. High-Level Register Transfers for Selected Instructions

In the following descriptions, these symbols are used:

- *instruction_register*$(a..b)$ refers to bits a through b in the current instruction word.
- x refers to *instruction_register*$(2..0)$ - the number at bit positions 2..0 in the instruction word.
- y refers to the number at bit positions 11..9 in the instruction word.
- D_n means Data Register n.
- A_n means Address Register n.
- IDR means Immediate Data Register.

- EA means the actual value of the effective address (an absolute memory location).
- $[EA]$ means the value stored in memory at location EA.
- $[A_n]$ means the value stored in memory at location A_n.

BCC: branch conditionally

if $instruction\_register(7..0) = 0$ then $DataSize \leftarrow Word$
    Fetch Immediate Data
    if $ConditionTrue$ then
        $PC \leftarrow PC + IDR$
else if $instruction\_register(7..0)$ =0xff then
    $DataSize \leftarrow DoubleWord$
    Fetch Immediate Data
    if $ConditionTrue$ then
        $PC \leftarrow PC + IDR$
else
    if $ConditionTrue$ then
        $PC \leftarrow PC + instruction\_register(7..0)$

DBCC: test condition, decrement and branch

$DataSize \leftarrow Word$
Fetch Immediate Data
if $ConditionTrue$ then $D_x \leftarrow D_x - 1$
    if $D_x \neq -1$ then
        $PC \leftarrow PC + IDR$

JMP: jump

$PC \leftarrow EA$

JSR: jump to subroutine (call)

(y is always 7, so A_y = Stack Pointer)
$A_y \leftarrow A_y - 4$
$[A_y] \leftarrow PC$
$PC \leftarrow EA$

MOVE: move

$[DestEA] \leftarrow [SourceEA]$

MOVEQ: move short immediate

$D_y \leftarrow instruction\_register(7..0)$

NEG:

Decode $DataSize$ from instruction_register(7..6)
$[EA] \leftarrow 0 - [EA]$

RTS: return from subroutine

(y is always 7, so A_y = Stack Pointer)
$PC \leftarrow [A_y]$
$A_y \leftarrow A_y + 4$

SCC: set according to condition codes

$DataSize \leftarrow Byte$
if $ConditionTrue$ then
    $[EA] \leftarrow$0xff
else
    $[EA] \leftarrow 0$

SUB: subtract

Decode $DataSize$ from $instruction\_register(7..6)$
if $instruction\_register(8) = 0$, then
    $D_y \leftarrow D_y - [EA]$
else
    $[EA] \leftarrow [EA] - D_y$

SUBA: subtract address registers

Decode $DataSize$ from $instruction\_register(7..6)$
$A_y \leftarrow A_y - [EA]$

SUBI: subtract immediate

Decode $DataSize$ from $instruction\_register(7..6)$
Fetch Immediate Data
$[EA] \leftarrow [EA] - IDR$

SUBQ: subtract short immediate

Decode $DataSize$ from $instruction\_register(7..6)$
$[EA] \leftarrow [EA] - y$

SUBX: subtract with extend

Decode $DataSize$ from $instruction\_register(7..6)$
if $instruction\_register(3) = 0$, then
    $D_y \leftarrow D_y - D_x - ExtendFlag$
else
    $A_x \leftarrow A_x - DataSize$
    $A_y \leftarrow A_y - DataSize$
    $A_y \leftarrow A_y - A_x - ExtendFlag$

D. Linker scripts and `crt0.s`

D.1. `crt0.s` file used for embedded applications

```
 1 # 1 "crt0.S"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "crt0.S"
 5 # 17 "crt0.S"
 6 # 1 "asm.h" 1
 7 # 18 "crt0.S" 2
 8
 9         .title "crt0.S for m68k-coff"
10
11         .data
12         .align 2
13 _environ:
14         .long 0
15
16         .align 2
17         .text
18
19         .extern _main
20         .extern _exit
21         .extern _hardware_init_hook
22         .extern _software_init_hook
23         .extern _atexit
24         .extern ___do_global_dtors
25
26         .extern __stack
27         .extern __bss_start
28         .extern _end
29
30         .global _start
31         .global ___main
32         .global _atexit
33
34 _start:
35
36         movel #__stack, a0
37         cmpl #0, a0
38         jbeq 1f
39         movel a0, sp
40 1:
41
42         link a6, #-8
43
44         movel #__bss_start, d1
45         movel #_end, d0
46         cmpl d0, d1
47         jbeq 3f
48         movl d1, a0
49         subl d1, d0
50         subql #1, d0
51 2:
52         clrb (a0)+
53
54         dbra d0, 2b
55         clrw d0
56         subql #1, d0
57         jbcc 2b
58
59 3:
60
61 #        lea _hardware_init_hook, a0
62 #        cmpl #0,a0
63 #        jbeq 4f
64 #        jsr (a0)
65 #4:
66 #
67 #        lea _software_init_hook, a0
68 #        cmpl #0,a0
69 #        jbeq 5f
```

```
70 #        jsr (a0)
71 #5:
72 # 126 "crt0.S"
73 #        movel #__FINI_SECTION__,(sp)
74 #        jsr _atexit
75 #
76 #        jsr __INIT_SECTION__
77
78         pea 0
79         pea _environ
80         pea sp@(4)
81         pea 0
82         jsr _main
83         # movel d0, sp@-
84
85 _halted_loop:
86         jmp  _halted_loop
87
88
89 ___main:
90 _atexit:
91         rts
92
93         # jsr _exit
94
```

D.2. `tiny.x` linker script used for the embedded applications

```
 1 /* linker script for my tiny binaries. */
 2
 3 OUTPUT_FORMAT("a.out-zero-big", "a.out-zero-big",
 4       "a.out-zero-big")
 5 OUTPUT_ARCH(m68k)
 6 SEARCH_DIR("/usr/jdw108/m68k-utils/m68k-linux-aout/lib");
 7 PROVIDE (__stack = 0x2000);
 8 SECTIONS
 9 {
10   . = 0;
11   .text :
12   {
13     CREATE_OBJECT_SYMBOLS
14     *(.text)
15     /* The next six sections are for SunOS dynamic linking.  The order
16        is important.  */
17     *(.dynrel)
18     *(.hash)
19     *(.dynsym)
20     *(.dynstr)
21     *(.rules)
22     *(.need)
23     _etext = .;
24     __etext = .;
25   }
26   . = ALIGN(0x0800);
27   .data :
28   {
29     /* The first three sections are for SunOS dynamic linking.  */
30     *(.dynamic)
31     *(.got)
32     *(.plt)
33     *(.data)
34     *(.linux-dynamic) /* For Linux dynamic linking.  */
35     CONSTRUCTORS
36   }
37   . = ALIGN(0x0c00);
38   .other :
39   {
40     _edata  =  .;
41     __edata  =  .;
42   }
43   .bss :
44   {
45     __bss_start = .;
```

```
46    *(.bss)
47    *(COMMON)
48    . = ALIGN(4);
49    _end = . ;
50    __end = . ;
51   }
52 }
```

E. VHDL sources

E.1. Source code of `alu.vhd`

```
 1
 2     carry <= ccr ( 0 ) ;
 3     overflow <= ccr ( 1 ) ;
 4     zero <= ccr ( 2 ) ;
 5     negative <= ccr ( 3 ) ;
 6     extend <= ccr ( 4 ) ;
 7
 8     -- Bits 0..6
 9     alu_seg_1 : entity alu_segment
10             generic map ( bits => 7 )
11             port map ( input_a => alu_input_a ( 6 downto 0 ) ,
12                 input_b => alu_input_b ( 6 downto 0 ) ,
13                 carry_in => alu_carry_in ,
14                 alu_internal_op => alu_internal_op ,
15                 carry_out => alu_carry_out_6 ,
16                 output => alu_int_output ( 6 downto 0 ) ) ;
17
18     -- Bit 7
19     alu_seg_2 : entity alu_segment
20             generic map ( bits => 1 )
21             port map ( input_a => alu_input_a ( 7 downto 7 ) ,
22                 input_b => alu_input_b ( 7 downto 7 ) ,
23                 carry_in => alu_carry_out_6 ,
24                 alu_internal_op => alu_internal_op ,
25                 carry_out => alu_carry_out_7 ,
26                 output => alu_int_output ( 7 downto 7 ) ) ;
27
28     -- Bits 8..14
29     alu_seg_3 : entity alu_segment
30             generic map ( bits => 7 )
31             port map ( input_a => alu_input_a ( 14 downto 8 ) ,
32                 input_b => alu_input_b ( 14 downto 8 ) ,
33                 carry_in => alu_carry_out_7 ,
34                 alu_internal_op => alu_internal_op ,
35                 carry_out => alu_carry_out_14 ,
36                 output => alu_int_output ( 14 downto 8 ) ) ;
37
38     -- Bit 15
39     alu_seg_4 : entity alu_segment
40             generic map ( bits => 1 )
41             port map ( input_a => alu_input_a ( 15 downto 15 ) ,
42                 input_b => alu_input_b ( 15 downto 15 ) ,
43                 carry_in => alu_carry_out_14 ,
44                 alu_internal_op => alu_internal_op ,
45                 carry_out => alu_carry_out_15 ,
46                 output => alu_int_output ( 15 downto 15 ) ) ;
47
48     -- Bits 16..30
49     alu_seg_5 : entity alu_segment
50             generic map ( bits => 15 )
51             port map ( input_a => alu_input_a ( 30 downto 16 ) ,
52                 input_b => alu_input_b ( 30 downto 16 ) ,
53                 carry_in => alu_carry_out_15 ,
54                 alu_internal_op => alu_internal_op ,
55                 carry_out => alu_carry_out_30 ,
56                 output => alu_int_output ( 30 downto 16 ) ) ;
57
58     -- Bit 31
59     alu_seg_6 : entity alu_segment
```

```
60              generic map ( bits => 1 )
61              port map ( input_a => alu_input_a ( 31 downto 31 ) ,
62                  input_b => alu_input_b ( 31 downto 31 ) ,
63                  carry_in => alu_carry_out_30 ,
64                  alu_internal_op => alu_internal_op ,
65                  carry_out => alu_carry_out_31 ,
66                  output => alu_int_output ( 31 downto 31 ) ) ;
67
68
69      alu : process ( alu_mode , extend , alu_internal_op ,
70                  alu_input_b , alu_int_output ) is
71          variable cin : std_logic ;
72      begin
73          -- Should alu_carry_in be set?
74          -- First we look at the extend bit in the CCR. In one
75          -- mode this is taken into account.
76          if ( alu_mode = ALU_X_FAMILY )
77          and ( extend = '1' )
78          then
79              cin := '1' ;
80          else
81              cin := '0' ;
82          end if ;
83
84          -- Then, carry_in is set. The state is inverted if we are
85          -- doing a subtraction or comparison, because subtractions
86          -- are actually performed as additions with the 2's complement
87          -- of input b. Inverting the carry input is necessary to get
88          -- the correct 2's complement value.
89          if ( alu_internal_op = ALU_INT_CMP )
90          or ( alu_internal_op = ALU_INT_SUB )
91          or ( alu_internal_op = ALU_INT_REV_SUB )
92          or ( alu_internal_op = ALU_INT_REV_CMP )
93          then
94              alu_carry_in <= not cin ;
95          else
96              alu_carry_in <= cin ;
97          end if ;
98
99          alu_output <= alu_int_output ;
100     end process alu ;
101
102     alu_ccr : process ( alu_internal_op , operation_size ,
103                 alu_carry_out_6 , alu_carry_out_7 ,
104                 alu_carry_out_14 , alu_carry_out_15 ,
105                 alu_carry_out_30 , alu_carry_out_31 ,
106                 alu_int_output , alu_modify_ccrs , ccr ) is
107         variable bits_0_to_7_are_zero   : std_logic ;
108         variable bits_0_to_15_are_zero  : std_logic ;
109         variable bits_0_to_31_are_zero  : std_logic ;
110     begin
111         -- Set the condition code registers, if the ALU mode
112         -- is one in which they may be modified (all internal
113         -- operations, PC <- PC + 1 etc, are non-CCR modifying).
114         if ( alu_modify_ccrs = '1' )
115         then
116             if ( alu_int_output ( 7 downto 0 ) =
117                             conv_std_logic_vector ( 0 , 8 ) )
118             then
119                 bits_0_to_7_are_zero := '1' ;
120             else
121                 bits_0_to_7_are_zero := '0' ;
122             end if ;
123
124             if ( alu_int_output ( 15 downto 8 ) =
125                             conv_std_logic_vector ( 0 , 8 ) )
126             and ( bits_0_to_7_are_zero = '1' )
127             then
128                 bits_0_to_15_are_zero := '1' ;
129             else
130                 bits_0_to_15_are_zero := '0' ;
131             end if ;
132
133             if ( alu_int_output ( 31 downto 16 ) =
```

```
134                             conv_std_logic_vector ( 0 , 16 ) )
135             and ( bits_0_to_15_are_zero = '1' )
136             then
137                 bits_0_to_31_are_zero := '1' ;
138             else
139                 bits_0_to_31_are_zero := '0' ;
140             end if ;
141
142             case operation_size is
143             when BYTE =>
144                     alu_ccr_output ( 0 ) <= alu_carry_out_7 ;
145                     alu_ccr_output ( 1 ) <= alu_carry_out_7
146                                             xor alu_carry_out_6 ;
147                     alu_ccr_output ( 2 ) <= bits_0_to_7_are_zero ;
148                     alu_ccr_output ( 3 ) <= alu_int_output ( 7 ) ;
149             when WORD =>
150                     alu_ccr_output ( 0 ) <= alu_carry_out_15 ;
151                     alu_ccr_output ( 1 ) <= alu_carry_out_15
152                                             xor alu_carry_out_14 ;
153                     alu_ccr_output ( 2 ) <= bits_0_to_15_are_zero ;
154                     alu_ccr_output ( 3 ) <= alu_int_output ( 15 ) ;
155             when others => -- DWORD =>
156                     alu_ccr_output ( 0 ) <= alu_carry_out_31 ;
157                     alu_ccr_output ( 1 ) <= alu_carry_out_31
158                                             xor alu_carry_out_30 ;
159                     alu_ccr_output ( 2 ) <= bits_0_to_31_are_zero ;
160                     alu_ccr_output ( 3 ) <= alu_int_output ( 31 ) ;
161             end case ;
162         else
163             alu_ccr_output ( 3 downto 0 ) <= ccr ( 3 downto 0 ) ;
164         end if ;
165         alu_ccr_output ( 15 downto 4 ) <= ccr ( 15 downto 4 ) ;
166     end process alu_ccr ;
167
168     ccr_update_process : process ( clock , alu_ccr_output ) is
169     begin
170         if ( clock = '1' )
171         and ( clock'event )
172         then
173             ccr <= alu_ccr_output ;
174         end if ;
175     end process ccr_update_process ;
176
177     dbcc_monitor : process ( alu_int_output ) is
178         constant minus_one : word_register := ( others => '1' ) ;
179     begin
180         -- This process is solely for the use of DBcc.
181         -- If the ALU output is -1 (word), then alu_output_is_minus_one is
182         -- set to 1.
183         if ( alu_int_output ( 15 downto 0 ) = minus_one )
184         then
185             alu_output_is_minus_one <= '1' ;
186         else
187             alu_output_is_minus_one <= '0' ;
188         end if ;
189     end process dbcc_monitor ;
190
```

E.2. Source code of `alu_muxes.vhd`

```
1
2
3
4     alu_input_muxes : process ( alu_source_a , alu_source_b ,
5                 pc_register ,
6                 register_file_address_out_x , register_file_address_out_y ,
7                 register_file_data_out_x , register_file_data_out_y ,
8                 operand_value , operand_address ,
9                 alu_input_small_number ,
10                immediate_data_reg ,
11                instruction_register ) is
12    begin
13        -- MUX for alu_input_a:-
```

```
14          case alu_source_a is
15          when ALU_A_PC =>
16                      alu_input_a <= pc_register ;
17          when ALU_A_ADDRESS_X =>
18                      alu_input_a <= register_file_address_out_x ;
19          when ALU_A_DATA_X =>
20                      alu_input_a <= register_file_data_out_x ;
21          when ALU_A_OPERAND_VALUE =>
22                      alu_input_a <= operand_value ;
23          when others => -- ALU_A_PGI =>
24                      alu_input_a ( 3 downto 0 ) <= alu_input_small_number ;
25                      alu_input_a ( 31 downto 4 ) <= ( others => '0' ) ;
26          end case ;
27
28          -- MUX for alu_input_b:-
29          case alu_source_b is
30          when ALU_B_PGI =>
31                      -- A small number from 0 to 15. We do it with a separate
32                      -- signal, alu_input_small_number, so that a large
33                      -- 32 bit mux will not be synthesised just to switch
34                      -- between 4 bit numbers.
35                      alu_input_b ( 3 downto 0 ) <= alu_input_small_number ;
36                      alu_input_b ( 31 downto 4 ) <= ( others => '0' ) ;
37          when ALU_B_IDR =>
38                      alu_input_b <= immediate_data_reg ;
39          when ALU_B_ADDRESS_Y =>
40                      alu_input_b <= register_file_address_out_y ;
41          when ALU_B_DATA_Y =>
42                      alu_input_b <= register_file_data_out_y ;
43          when ALU_B_OA =>
44                      alu_input_b <= operand_address ;
45          when others => -- ALU_B_LOW_BYTE_OF_IR =>
46                      -- IR(7..0) sign extended to 32 bits - used for branches
47                      alu_input_b ( 7 downto 0 ) <=
48                              instruction_register ( 7 downto 0 ) ;
49                      alu_input_b ( 31 downto 8 ) <=
50                              ( others => instruction_register ( 7 ) ) ;
51          end case ;
52      end process alu_input_muxes ;
53
54      alu_input_muxes_2 : process ( pgi_source , operation_size ,
55                      instruction_register , ea_reg ) is
56          variable qim        : std_logic_vector ( 4 downto 0 ) ;
57          variable postinc    : std_logic_vector ( 4 downto 0 ) ;
58      begin
59          -- MUX for alu_input_small_number:-
60          case pgi_source is
61          when PGI_ZERO =>
62                      alu_input_small_number <= "0000" ;
63          when PGI_ONE =>
64                      alu_input_small_number <= "0001" ;
65          when PGI_TWO =>
66                      alu_input_small_number <= "0010" ;
67          when PGI_THREE =>
68                      alu_input_small_number <= "0011" ;
69          when PGI_FOUR =>
70                      alu_input_small_number <= "0100" ;
71          when PGI_QUICK_IMMEDIATE =>
72                      -- This is meaningful only for ADDQ instructions.
73                      if ( instruction_register ( 11 downto 9 ) = "000" )
74                      then
75                          alu_input_small_number <= "1000" ;
76                      else
77                          alu_input_small_number ( 3 ) <= '0' ;
78                          alu_input_small_number ( 2 downto 0 ) <=
79                                  instruction_register ( 11 downto 9 ) ;
80                      end if ;
81          when PGI_POSTINC_PREDEC =>
82                      -- Calculate modifier for postinc/predec
83                      -- effective addresses.
84                      case operation_size is
85                      when BYTE =>
86                              -- Byte operations on register 7 are treated
87                              -- as word operations so as to preserve
```

```
88                          -- stack alignment.
89                          if ( ea_reg = "111" )
90                          then
91                              alu_input_small_number <= "0010" ;
92                          else
93                              alu_input_small_number <= "0001" ;
94                          end if ;
95                  when WORD =>
96                          alu_input_small_number <= "0010" ;
97                  when others => -- DWORD
98                          alu_input_small_number <= "0100" ;
99                  end case ;
100         end case ;
101     end process alu_input_muxes_2 ;
102
103     alu_control_mux : process ( alu_mode ,
104             alu_reverse_operands , instruction_register ) is
105         variable sub_op : alu_internal_op_type ;
106         variable cmp_op : alu_internal_op_type ;
107         variable op     : alu_internal_op_type ;
108     begin
109         if ( alu_reverse_operands = '1' )
110         then
111             sub_op := ALU_INT_REV_SUB ;
112             cmp_op := ALU_INT_REV_CMP ;
113         else
114             sub_op := ALU_INT_SUB ;
115             cmp_op := ALU_INT_CMP ;
116         end if ;
117
118         -- The aim of this process is to translate an arithmetic
119         -- operation type described in the opcode into something that
120         -- the ALU can understand. There is a slightly different
121         -- translation for each type of instruction.
122         case alu_mode is
123         when ALU_I_FAMILY =>
124                     case instruction_register ( 11 downto 9 ) is
125                     when "000" =>
126                             op := ALU_INT_OR ;
127                     when "001" =>
128                             op := ALU_INT_AND ;
129                     when "010" =>
130                             op := sub_op ;
131                     when "011" =>
132                             op := ALU_INT_ADD ;
133                     when "101" =>
134                             op := ALU_INT_EOR ;
135                     when "110" =>
136                             op := cmp_op ;
137                     when others => -- op is (really) undefined.
138                             op := ALU_INT_ADD ;
139                     end case ;
140                     alu_modify_ccrs <= '1' ;
141         when ALU_Q_FAMILY|ALU_CLR_FAMILY =>
142                     case instruction_register ( 8 ) is
143                     when '0' =>
144                             op := ALU_INT_ADD ;
145                     when others =>
146                             op := sub_op ;
147                     end case ;
148                     alu_modify_ccrs <= '1' ;
149         when ALU_NO_FAMILY|ALU_A_FAMILY|ALU_X_FAMILY =>
150                     case instruction_register ( 14 downto 12 ) is
151                     when "000" =>
152                             op := ALU_INT_OR ;
153                     when "001" =>
154                             op := sub_op ;
155                     when "011" =>
156                             if ( instruction_register ( 8 ) = '1' )
157                             and ( alu_mode = ALU_NO_FAMILY )
158                             then
159                                 op := ALU_INT_EOR ;
160                             else
161                                 op := cmp_op ;
```

```
162                             end if ;
163                     when "100" =>
164                             op := ALU_INT_AND ;
165                     when "101" =>
166                             op := ALU_INT_ADD ;
167                     when others => -- op is (really) undefined.
168                             op := ALU_INT_ADD ;
169                     end case ;
170                     alu_modify_ccrs <= '1' ;
171         when ALU_ADD_UPDATE_CCRS =>
172                     op := ALU_INT_ADD ;
173                     alu_modify_ccrs <= '1' ;
174         when ALU_ADD =>
175                     op := ALU_INT_ADD ;
176                     alu_modify_ccrs <= '0' ;
177         when ALU_SUBTRACT =>
178                     op := sub_op ;
179                     alu_modify_ccrs <= '0' ;
180         when others => -- op is (really) undefined.
181                     op := ALU_INT_ADD ;
182                     alu_modify_ccrs <= '0' ;
183         end case ;
184         alu_internal_op <= apply_alu_internal_op ( op ) ;
185
186     end process alu_control_mux ;
187
```

E.3. Source code of alu_segment.vhd

```
1  -- alu_segment.vhd
2  --
3  -- The entity here implements an n-bit wide slice of the ALU.
4  -- The width is set by the 'bits' generic parameter.
5  --
6
7  library ieee ;
8  use ieee . std_logic_1164 . all ;
9  use ieee . std_logic_arith . all ;
10 use ieee . std_logic_unsigned . all ;
11 use m68k_types . all ;
12
13 entity alu_segment is
14     generic ( bits      : integer ) ;
15     port (  input_b     : in std_logic_vector (( bits - 1 ) downto 0 ) ;
16             input_a     : in std_logic_vector (( bits - 1 ) downto 0 ) ;
17             output      : out std_logic_vector (( bits - 1 ) downto 0 ) ;
18             carry_in    : in std_logic ;
19             carry_out   : out std_logic ;
20             alu_internal_op
21                         : in alu_internal_op_type ) ;
22 end entity alu_segment ;
23
24 architecture basic of alu_segment is
25 begin
26
27     -- ALU internal operation codes
28     -- ALU_INT_OR       output <- A or B
29     -- ALU_INT_AND      output <- A and B
30     -- ALU_INT_SUB      output <- A - B
31     -- ALU_INT_ADD      output <- A + B
32     -- ALU_INT_EOR      output <- A xor B
33     -- ALU_INT_CMP      output <- B - A
34     -- ALU_INT_REV_SUB  output <- B - A
35     -- ALU_INT_REV_CMP  output <- B - A
36
37     process ( input_b , input_a , alu_internal_op , carry_in ) is
38         variable int_input_b    : std_logic_vector
39                                     (( bits + 1 ) downto 0 ) ;
40         variable int_input_a    : std_logic_vector
41                                     (( bits + 1 ) downto 0 ) ;
42         variable int_output     : std_logic_vector
43                                     (( bits + 1 ) downto 0 ) ;
44     begin
```

```
45          int_input_a ( bits + 1 ) := '0' ;
46          int_input_b ( bits + 1 ) := '0' ;
47
48          if ( alu_internal_op = ALU_INT_REV_SUB )
49          or ( alu_internal_op = ALU_INT_REV_CMP )
50          then
51              -- Reverse subtraction. The result is B - A.
52              -- 2's complement of A is obtained:
53              int_input_b ( 0 ) := carry_in ;
54              int_input_b ( bits downto 1 ) := input_b ;
55
56              int_input_a ( 0 ) := '1' ;
57              int_input_a ( bits downto 1 ) := not input_a ;
58          else
59              int_input_a ( 0 ) := carry_in ;
60              int_input_a ( bits downto 1 ) := input_a ;
61
62              int_input_b ( 0 ) := '1' ;
63              if ( alu_internal_op = ALU_INT_SUB )
64              or ( alu_internal_op = ALU_INT_CMP )
65              then
66                  -- For subtraction operations, 2's complement of B
67                  -- is obtained.
68                  int_input_b ( bits downto 1 ) := not input_b ;
69              else
70                  int_input_b ( bits downto 1 ) := input_b ;
71              end if ;
72          end if ;
73
74          case alu_internal_op is
75          when ALU_INT_AND =>
76                  int_output := ( int_input_a and int_input_b ) ;
77          when ALU_INT_EOR =>
78                  int_output := ( int_input_a xor int_input_b ) ;
79          when ALU_INT_OR =>
80                  int_output := ( int_input_a or int_input_b ) ;
81          when others => -- ADD, SUB, CMP
82                  int_output := ( int_input_a + int_input_b ) ;
83          end case ;
84
85          carry_out <= int_output ( bits + 1 ) ;
86          output <= int_output ( bits downto 1 ) ;
87      end process ;
88
89  end architecture basic ;
90
91
```

E.4. Source code of `clock.vhd`

```
 1
 2  -- Clock processes
 3
 4  -- The clock controller is part of the debugging system.
 5  -- The clock can be run slowly: the switch settings control the speed.
 6      clock_controller : process ( switches , fast_clock ,
 7              button_clock_event , slow_clock ) is
 8      begin
 9          if ( fast_clock = '1' )
10          and ( fast_clock'event )
11          then
12              button_clock_event_clear_1 <= '0' ;
13              reset <= '0' ;
14              run_single_instruction <= '0' ;
15
16              case switches ( 6 downto 5 ) is
17              when "00" =>
18                  -- Reset the processor.
19                  reset <= '1' ;
20                  clock <= not clock ;
21              when "01" =>
22                  -- Advance to the next clock edge on each button press
23                  if ( button_clock_event = '1' )
```

```
24                 then
25                     clock <= not clock ;
26                     button_clock_event_clear_1 <= '1' ;
27                 end if ;
28             when others =>
29                 -- Run continuously at full-ish speed.
30                 clock <= slow_clock ( 12 ) ;
31             end case ;
32             slow_clock <= slow_clock + 1 ;
33         end if ;
34     end process ;
35
36 -- This process debounces the button.
37     button_debouncer : process ( fast_clock , button_clock_event_clear_1 ,
38             button_clock_event_clear_2 , button , last_button ) is
39     begin
40         if ( fast_clock = '0' )
41         and ( fast_clock'event )
42         then
43             if ( button_clock_event_clear_1 = '1' )
44             or ( button_clock_event_clear_2 = '1' )
45             then
46                 button_clock_event <= '0' ;
47             end if ;
48
49             if (( button xor last_button ) = '1' )
50             then
51                 -- Button just changed.
52                 if ( button_state_stable (
53                     button_state_stable'length - 1 ) = '1' )
54                 then
55                     -- button was stable. generate a clock event.
56                     button_clock_event <= '1' ;
57                 end if ;
58                 button_state_stable <= ( others => '0' ) ;
59             else
60                 if ( button_state_stable (
61                     button_state_stable'length - 1 ) = '0' )
62                 then
63                     button_state_stable <= button_state_stable + 1 ;
64                 end if ;
65             end if ;
66             last_button <= button ;
67         end if ;
68     end process ;
69
```

E.5. Source code of `debugging.vhd`

```
 1
 2     led_display : entity seven_segment_driver(basic)
 3         port map ( clock => fast_clock ,
 4                     -- byte_to_output ( 4 downto 0 ) => state ,
 5                     -- byte_to_output ( 7 downto 5 ) => "000" ,
 6                     -- byte_to_output =>
 7                             -- data_register_2 ( 31 downto 24 ) ,
 8                     byte_to_output => led_display_word ( 15 downto 8 ) ,
 9                     blank_display => '0' ,
10                     enable => '1' ,
11                     led_output_pins => led_display_output ) ;
12
13     right_led_display : entity seven_segment_driver(basic)
14         port map ( clock => fast_clock ,
15                     -- byte_to_output ( 7 downto 4 ) =>
16                             -- call_stack_pointer ,
17                     -- byte_to_output ( 3 downto 0 ) =>
18                             -- data_register_2 ( 3 downto 0 ) ,
19                     byte_to_output => led_display_word ( 7 downto 0 ) ,
20                     blank_display => '0' ,
21                     enable => '1' ,
22                     led_output_pins => right_led_display_output ) ;
23
24     led_display_mux : process (
```

```
25            switches , operand_address , operand_value ,
26            pc_register , instruction_register ,
27            register_file_address_out_x , register_file_address_out_y ,
28            register_file_data_out_x , register_file_data_out_y ,
29            immediate_data_reg , register_file_data_out_y ,
30            state , call_stack_at_ptr_minus_one ,
31            call_stack_pointer , clock ,
32            ea_move_destination_control , ccr , condition_true ,
33            last_output , debug_memory_out ,
34            operation_size ) is
35    begin
36        case switches ( 4 downto 0 ) is
37        when "00000" =>
38            led_display_word <= operand_address ( 15 downto 0 ) ;
39        when "00001" =>
40            led_display_word <= operand_value ( 15 downto 0 ) ;
41        when "00010" =>
42            led_display_word <= pc_register ( 15 downto 0 ) ;
43        when "00011" =>
44            led_display_word <= instruction_register ;
45        when "00100" =>
46            led_display_word <= register_file_address_out_x ( 15 downto 0 ) ;
47        when "00101" =>
48            led_display_word <= register_file_data_out_x ( 15 downto 0 ) ;
49        when "00110" =>
50            led_display_word <= register_file_address_out_y ( 15 downto 0 ) ;
51        when "00111" =>
52            led_display_word <= register_file_address_out_y ( 31 downto 16 ) ;
53        when "01000" =>
54            led_display_word <= register_file_data_out_y ( 15 downto 0 ) ;
55        when "01001" =>
56            led_display_word <= register_file_data_out_y ( 31 downto 16 ) ;
57        when "01010" =>
58            led_display_word <= immediate_data_reg ( 15 downto 0 ) ;
59        when "01011" =>
60            led_display_word <= immediate_data_reg ( 31 downto 16 ) ;
61        when "01100" =>
62            led_display_word ( state_register'length - 1 downto 0 ) <= state ;
63            led_display_word ( 15 downto state_register'length ) <=
64                                        ( others => '0' ) ;
65        when "01101" =>
66            led_display_word ( state_register'length - 1 downto 0 ) <=
67                        call_stack_at_ptr_minus_one ;
68            led_display_word ( 15 downto state_register'length ) <=
69                                        ( others => '0' ) ;
70        when "01110" =>
71            led_display_word ( stack_pointer_register'length - 1 downto 0 ) <=
72                        call_stack_pointer ;
73            led_display_word ( 7 downto stack_pointer_register'length ) <=
74                                        ( others => '0' ) ;
75            led_display_word ( 15 downto 8 ) <= ccr ( 7 downto 0 ) ;
76        when "01111" =>
77            led_display_word ( 15 downto 0 ) <= ( others => '0' ) ;
78
79            led_display_word ( 15 ) <= condition_true ;
80            led_display_word ( 14 ) <=
81                        ea_move_destination_control ;
82
83            case operation_size is
84            when BYTE =>
85                led_display_word ( 10 downto 8 ) <= "001" ;
86            when WORD =>
87                led_display_word ( 10 downto 8 ) <= "010" ;
88            when others =>
89                led_display_word ( 10 downto 8 ) <= "100" ;
90            end case ;
91
92            led_display_word ( 7 downto 0 ) <= last_output ;
93        when others =>
94            led_display_word ( 15 downto 12 ) <= switches ( 3 downto 0 ) ;
95            led_display_word ( 11 downto 8 ) <= ( others => '0' ) ;
96            led_display_word ( 7 downto 0 ) <= debug_memory_out ;
97        end case ;
98    end process ;
```

```
 99
100     light_1 <= '1' ;
101     light_2 <= clock ;
102     light_3 <= clock ;
103
```

E.6. Source code of do_branch_process.vhd

```
 1
 2     do_branch_process : process ( instruction_register ,
 3                     carry , overflow , negative , zero ) is
 4         variable db : std_logic ;
 5     begin
 6         -- Examine the condition code. Only the top 3 bits need
 7         -- to be examined because the lowest bit of the code means
 8         -- "complement result".
 9         case instruction_register ( 11 downto 9 ) is
10         when "001" =>   -- BHI, high
11                         db := ( not carry ) and ( not zero ) ;
12         when "010" =>   -- BCC, carry clear
13                         db := not carry ;
14         when "011" =>   -- BNE, not equal
15                         db := not zero ;
16         when "100" =>   -- BVC, overflow clear
17                         db := not overflow ;
18         when "101" =>   -- BPL, plus
19                         db := not negative ;
20         when "110" =>   -- BGE, greater than or equal
21                         db := ( negative and overflow )
22                                 or (( not negative ) and ( not overflow )) ;
23         when "111" =>   -- BGT, greater than
24                         db := ( negative and overflow and ( not zero ))
25                                 or (( not negative ) and ( not overflow )
26                                         and ( not zero )) ;
27         when others =>  -- BRA, branch always
28                         db := '1' ;
29         end case ;
30
31         if ( instruction_register ( 8 ) = '1' )
32         then
33             -- Complement result of examination above.
34             db := not db ;
35         end if ;
36
37         condition_true <= db ;
38     end process do_branch_process ;
39
```

E.7. Source code of input.vhd

```
 1 -- input.vhd
 2 --
 3 -- $Id: input.vhd,v 1.1 2003/01/17 15:18:39 jdw108 Exp jwhitham $
 4
 5 library ieee ;
 6 use ieee . std_logic_1164 . all ;
 7 use ieee . std_logic_arith . all ;
 8 use ieee . std_logic_unsigned . all ;
 9 use m68k_types . all ;
10
11 entity state_machine is
12     port ( fast_clock : in std_logic ;
13             button : in std_logic ;
14             led_display_output : out std_logic_vector ( 6 to 19 ) ;
15             right_led_display_output : out std_logic_vector ( 6 to 19 ) ;
16             switches : in std_logic_vector ( 6 downto 0 ) ;
17             light_1 : out std_logic ;
18             light_2 : out std_logic ;
19             light_3 : out std_logic ) ;
20 end entity state_machine ;
21
```

```
22 architecture basic of state_machine is
23
24 -- Include signal definition code
25
26 INCLUDE clock.s.vhd
27 INCLUDE alu.s.vhd
28 INCLUDE alu_muxes.s.vhd
29 INCLUDE debugging.s.vhd
30 INCLUDE do_branch_process.s.vhd
31 INCLUDE ea_mode_mux_process.s.vhd
32 INCLUDE memory.s.vhd
33 INCLUDE operation_size_control_process.s.vhd
34 INCLUDE register_file.s.vhd
35 INCLUDE state_machine_controller.s.vhd
36 INCLUDE defaults.s.vhd
37 INCLUDE restore_pc_after_immediate_fetch.s.vhd
38
39 -- Optimisation code
40 INSERT OPTIMISATION alu_internal_op
41 INSERT OPTIMISATION ea_mode
42 INSERT OPTIMISATION ea_reg
43
44 begin
45
46 -- Code for CPU components
47
48 INCLUDE clock.vhd
49 INCLUDE alu.vhd
50 INCLUDE alu_muxes.vhd
51 INCLUDE debugging.vhd
52 INCLUDE do_branch_process.vhd
53 INCLUDE ea_mode_mux_process.vhd
54 INCLUDE memory.vhd
55 INCLUDE operation_size_control_process.vhd
56 INCLUDE register_file.vhd
57 INCLUDE restore_pc_after_immediate_fetch.vhd
58
59 state_machine_process : process (
60
61 INCLUDE state_machine_sensitivity_list.vhd
62
63 state , clock ) is
64
65 begin
66
67 INCLUDE defaults.vhd
68
69 INSERT STATE MACHINE
70
71 end process state_machine_process ;
72
73 instruction_decoder : process ( instruction_register ) is
74     variable ir : word_register ;
75 begin
76     ir := instruction_register ;
77     INSERT INSTRUCTION DECODER
78 end process instruction_decoder ;
79
80 INCLUDE state_machine_controller.vhd
81
82 end architecture basic ;
83
```

E.8. Source code of `memory.vhd`

```
1
2     rom : entity program_rom
3         port map ( address => memory_address ,
4                    data => rom_output ,
5                    clock => clock ) ;      -- memory access in +ve cycle
6
7     ram : entity dp_ram
8         generic map ( address_width => 12 , -- 4K RAM
```

```
 9                   data_width => 8 )
10       port map ( clock => clock ,         -- memory access in +ve cycle
11                   write => ram_write_enable ,
12                   address1 => memory_address ( 11 downto 0 ) ,
13                   address2 ( 3 downto 0 ) => switches ( 3 downto 0 ) ,
14                   address2 ( 11 downto 4 ) => "00000000" ,
15                   data_in => memory_input ,
16                   data_out1 => ram_output ,
17                   data_out2 => debug_memory_out ) ;
18
19    -- Memory map:
20    -- 0000-0fff    ROM
21    -- 1000-1fff    RAM
22    -- 8000         Output
23
24    memory_map_process  : process ( rom_output ,
25                ram_output , memory_address , clock ,
26                memory_write_enable , memory_input ) is
27    begin
28        case memory_address ( 15 downto 12 ) is
29        when "0000" =>  memory_output <= rom_output ;
30                        ram_write_enable <= '0' ;
31
32        when "0001" =>  memory_output <= ram_output ;
33                        ram_write_enable <= memory_write_enable ;
34
35        when "1000" =>  memory_output <= ( others => '0' ) ;
36                        if ( memory_write_enable = '1' )
37                        and ( memory_address ( 11 downto 0 ) =
38                                conv_std_logic_vector ( 0 , 12 ))
39                        then
40                            -- memory_input contains the
41                            -- data to be output.
42                            if ( clock = '1' )
43                            and ( clock'event )
44                            then
45                                last_output <= memory_input ;
46                            end if ;
47                        end if ;
48                        ram_write_enable <= '0' ;
49
50        when others =>  memory_output <= ( others => '0' ) ;
51                        ram_write_enable <= '0' ;
52        end case ;
53    end process memory_map_process ;
54
55    memory_address_mux : process ( mar_source ,
56            pc_register , operand_address ) is
57    begin
58        case mar_source is
59        when PC_TO_MAR =>
60                memory_address <= pc_register ;
61        when others => -- OA_TO_MAR
62                memory_address <= operand_address ;
63        end case ;
64    end process memory_address_mux ;
65
66    memory_output_latch : process ( clock , memory_output ) is
67    begin
68        if ( clock = '0' )
69        and ( clock'event )
70        then
71            last_memory_output <= memory_output ;
72        end if ;
73    end process memory_output_latch ;
74
75    memory_input_mux : process ( mdr_source ,
76            operand_value ) is
77    begin
78        case mdr_source is
79        when OV_0_TO_MDR =>
80                    memory_input <= operand_value ( 7 downto 0 ) ;
81                    memory_write_enable <= '1' ;
82        when OV_1_TO_MDR =>
```

```
83                  memory_input <= operand_value ( 15 downto 8 ) ;
84                  memory_write_enable <= '1' ;
85          when OV_2_TO_MDR =>
86                  memory_input <= operand_value ( 23 downto 16 ) ;
87                  memory_write_enable <= '1' ;
88          when OV_3_TO_MDR =>
89                  memory_input <= operand_value ( 31 downto 24 ) ;
90                  memory_write_enable <= '1' ;
91          when others =>
92                  -- Writing is turned off, so the source of memory_input
93                  -- is unimportant. Choose something that has been used
94                  -- before to simplify this mux.
95                  memory_input <= operand_value ( 31 downto 24 ) ;
96                  memory_write_enable <= '0' ;
97          end case ;
98      end process memory_input_mux ;
99
100     register_transfers : process ( ir_source ,
101             clock , last_memory_output , operand_value_source ,
102             alu_output , immediate_data_source , pc_source ,
103             operand_address , operand_address_source ) is
104     begin
105         -- The data that is transfered from memory MUST have been
106         -- fetched on the previous clock cycle. If it was fetched
107         -- earlier or later than that, it won't be available.
108         if ( clock = '1' )
109         and ( clock'event )
110         then
111             case ir_source is
112             when MDR_TO_IR_1 =>
113                         instruction_register ( 15 downto 8 )
114                                         <= last_memory_output ;
115             when MDR_TO_IR_0 =>
116                         instruction_register ( 7 downto 0 )
117                                         <= last_memory_output ;
118             when others => null ;
119             end case ;
120
121             case operand_value_source is
122             when MDR_TO_OV_3 =>
123                         operand_value ( 31 downto 24 )
124                                         <= last_memory_output ;
125             when MDR_TO_OV_2 =>
126                         operand_value ( 23 downto 16 )
127                                         <= last_memory_output ;
128             when MDR_TO_OV_1 =>
129                         operand_value ( 15 downto 8 )
130                                         <= last_memory_output ;
131             when MDR_TO_OV_0 =>
132                         operand_value ( 7 downto 0 )
133                                         <= last_memory_output ;
134             when ALU_TO_OV =>
135                         operand_value <= alu_output ;
136             when others => null ;
137             end case ;
138
139             case immediate_data_source is
140             when MDR_TO_IDR_3 =>
141                         immediate_data_reg ( 31 downto 24 )
142                                         <= last_memory_output ;
143             when MDR_TO_IDR_2 =>
144                         immediate_data_reg ( 23 downto 16 )
145                                         <= last_memory_output ;
146             when MDR_TO_IDR_1 =>
147                         immediate_data_reg ( 15 downto 8 )
148                                         <= last_memory_output ;
149             when MDR_TO_IDR_0 =>
150                         immediate_data_reg ( 7 downto 0 )
151                                         <= last_memory_output ;
152             when MDR_TO_IDR_1_SE =>
153                         immediate_data_reg ( 31 downto 16 )
154                             <= ( others => last_memory_output ( 7 ) ) ;
155                         immediate_data_reg ( 15 downto 8 )
156                                         <= last_memory_output ;
```

```
157             when MDR_TO_IDR_0_SE =>
158                         immediate_data_reg ( 31 downto 8 )
159                             <= ( others => last_memory_output ( 7 ) ) ;
160                         immediate_data_reg ( 7 downto 0 )
161                                     <= last_memory_output ;
162             when others => null ;
163             end case ;
164
165             case pc_source is
166             when ALU_TO_PC =>
167                         pc_register <= alu_output ;
168             when others => null ;
169             end case ;
170
171             case operand_address_source is
172             when ALU_TO_OA =>
173                         operand_address <= alu_output ;
174             when MDR_TO_OA_3 =>
175                         operand_address ( 31 downto 24 ) <= last_memory_output ;
176             when MDR_TO_OA_2 =>
177                         operand_address ( 23 downto 16 ) <= last_memory_output ;
178             when MDR_TO_OA_1_SE =>
179                         operand_address ( 31 downto 16 ) <=
180                                 ( others => last_memory_output ( 7 )) ;
181                         operand_address ( 15 downto 8 ) <= last_memory_output ;
182             when MDR_TO_OA_1 =>
183                         operand_address ( 15 downto 8 ) <= last_memory_output ;
184             when MDR_TO_OA_0 =>
185                         operand_address ( 7 downto 0 ) <= last_memory_output ;
186             when others => null ;
187             end case ;
188         end if ;
189     end process register_transfers ;
190
```

E.9. Source code of operation_size_control_process.vhd

```
1
2     operation_size_control_process : process ( clock ,
3                 operation_size_control , instruction_register ) is
4     begin
5         if ( clock = '1' )
6         and ( clock'event )
7         then
8             case operation_size_control is
9             when SET_TO_BYTE =>
10                    operation_size <= BYTE ;
11            when SET_TO_WORD =>
12                    operation_size <= WORD ;
13            when SET_TO_DWORD =>
14                    operation_size <= DWORD ;
15            when SET_TO_IR =>
16                    case instruction_register ( 7 downto 6 ) is
17                    when "00" =>    -- Byte size.
18                            operation_size <= BYTE ;
19                    when "01" =>    -- Word size.
20                            operation_size <= WORD ;
21                    when others =>  -- DWord size.
22                            operation_size <= DWORD ;
23                    end case ;
24            when others =>
25                    null ;
26            end case ;
27        end if ;
28    end process operation_size_control_process ;
29
```

E.10. Source code of register_file.vhd

```
1
2
```

```
 3
 4    data_register_file : entity dp_ram
 5        generic map ( address_width => 3 ,      -- 8 x 32
 6                    data_width => 32 )
 7        port map ( clock => clock ,
 8                    write => reg_update_data_x ,
 9                    address1 => register_file_address_x ,
10                    address2 => register_file_address_y ,
11                    data_in => data_regs_in ,
12                    data_out1 => register_file_data_out_x ,
13                    data_out2 => register_file_data_out_y ) ;
14
15    address_register_file : entity dp_ram
16        generic map ( address_width => 3 ,      -- 8 x 32
17                    data_width => 32 )
18        port map ( clock => clock ,
19                    write => reg_update_address_x ,
20                    address1 => register_file_address_x ,
21                    address2 => register_file_address_y ,
22                    data_in => address_regs_in ,
23                    data_out1 => register_file_address_out_x ,
24                    data_out2 => register_file_address_out_y ) ;
25
26    -- The register numbers, x and y, are chosen here. #
27    register_file_source_mux : process ( register_file_source_x ,
28                    register_file_source_y ,
29                    instruction_register , ea_reg ) is
30    begin
31        case register_file_source_x is
32        when RF_X_EA_REG =>
33                register_file_address_x <= ea_reg ;
34        when RF_X_FORCE_TO_SP =>
35                register_file_address_x <= "111" ;
36        when RF_X_11_TO_9_FIELD =>
37                register_file_address_x <=
38                            instruction_register ( 11 downto 9 ) ;
39        when others => -- RF_X_2_TO_0_FIELD =>
40                register_file_address_x <=
41                            instruction_register ( 2 downto 0 ) ;
42        end case ;
43
44        case register_file_source_y is
45        when RF_Y_FORCE_TO_SP =>
46                register_file_address_y <= "111" ;
47        when RF_Y_11_TO_9_FIELD =>
48                register_file_address_y <=
49                            instruction_register ( 11 downto 9 ) ;
50        when others => -- RF_Y_2_TO_0_FIELD =>
51                register_file_address_y <=
52                            instruction_register ( 2 downto 0 ) ;
53        end case ;
54    end process register_file_source_mux ;
55
56    -- The input to the register file is basically alu_output. However,
57    -- if a single word is written to a register, the high word is unchanged.
58    -- The following code supports this
59    register_file_input : process ( alu_output , operation_size ,
60                    reg_update_override_size ,
61                    register_file_address_out_x ,
62                    register_file_data_out_x ) is
63    begin
64        if ( reg_update_override_size = '1' )
65        or ( operation_size = DWORD )
66        then
67            -- Treat as a DWORD
68            address_regs_in <= alu_output ;
69            data_regs_in <= alu_output ;
70        else
71            if ( operation_size = BYTE )
72            then
73                address_regs_in ( 31 downto 8 ) <=
74                                ( others => alu_output ( 7 ) ) ;
75                address_regs_in ( 7 downto 0 ) <=
76                                alu_output ( 7 downto 0 ) ;
```

```
77                 data_regs_in ( 31 downto 8 ) <=
78                                 ( others => alu_output ( 7 ) ) ;
79                 data_regs_in ( 7 downto 0 ) <=
80                                 alu_output ( 7 downto 0 ) ;
81             else
82                 address_regs_in ( 31 downto 16 ) <=
83                                 ( others => alu_output ( 15 ) ) ;
84                 address_regs_in ( 15 downto 0 ) <=
85                                 alu_output ( 15 downto 0 ) ;
86                 data_regs_in ( 31 downto 16 ) <=
87                                 ( others => alu_output ( 15 ) ) ;
88                 data_regs_in ( 15 downto 0 ) <=
89                                 alu_output ( 15 downto 0 ) ;
90             end if ;
91         end if ;
92     end process register_file_input ;
93
```

E.11. Source code of seven_segment_driver.vhd

```
 1
 2 library ieee ;
 3 use ieee . std_logic_1164 . all ;
 4
 5 entity seven_segment_driver is
 6     port ( clock : in std_logic ;
 7            byte_to_output : in std_logic_vector ( 7 downto 0 ) ;
 8            blank_display : in std_logic ;
 9            enable : in std_logic ;
10            led_output_pins : out std_logic_vector ( 6 to 19 ) ) ;
11 end entity seven_segment_driver ;
12
13 architecture basic of seven_segment_driver is
14     subtype ad is std_logic_vector ( 1 to 4 ) ;
15     subtype dt is std_logic_vector ( 1 to 7 ) ;
16 begin
17     ssd_process : for digit in 0 to 1 generate
18         process ( byte_to_output , clock , blank_display , enable ) is
19             variable uout : dt ;
20             variable nibble : ad ;
21         begin
22             if ( enable = '1' )
23             and ( clock = '1' )
24             and ( clock'event )
25             then
26                 if ( blank_display = '1' )
27                 then
28                     uout := dt'("0000000" ) ;
29                 else
30                     nibble := byte_to_output(
31                         (( digit * 4 ) + 3 ) downto ( digit * 4 ) ) ;
32                     case nibble is
33                                     --                   edcgfab
34                         when ad'("0000") => uout := dt'("1110111") ; -- 0
35                         when ad'("0001") => uout := dt'("0010001") ;
36                         when ad'("0010") => uout := dt'("1101011") ;
37                         when ad'("0011") => uout := dt'("0111011") ;
38                         when ad'("0100") => uout := dt'("0011101") ; -- 4
39                         when ad'("0101") => uout := dt'("0111110") ;
40                         when ad'("0110") => uout := dt'("1111110") ;
41                         when ad'("0111") => uout := dt'("0010011") ;
42                         when ad'("1000") => uout := dt'("1111111") ; -- 8
43                         when ad'("1001") => uout := dt'("0011111") ;
44                         when ad'("1010") => uout := dt'("1011111") ;
45                         when ad'("1011") => uout := dt'("1111100") ;
46                         when ad'("1100") => uout := dt'("1101000") ; -- c
47                         when ad'("1101") => uout := dt'("1111001") ;
48                         when ad'("1110") => uout := dt'("1101110") ;
49                         when others =>      uout := dt'("1001110") ; -- f
50                     end case ;
51                 end if ;
52                 if ( digit = 0 )
53                 then
```

```
54                      -- less significant digit, i.e. digit on the right
55                      led_output_pins ( 13 to 19 ) <= uout ;
56                  else
57                      -- more significant digit, i.e. digit on the left
58                      led_output_pins ( 6 to 12 ) <= uout ;
59                  end if ;
60              end if ;
61          end process ;
62      end generate ;
63
64 -- pin arrangements:
65 -- e1 - 6  -- e2 - 13
66 -- d1 - 7  -- d2 - 14
67 -- c1 - 8  -- c2 - 15
68 -- g1 - 9  -- g2 - 16
69 -- f1 - 10 -- f2 - 17
70 -- a1 - 11 -- a2 - 18
71 -- b1 - 12 -- b2 - 19
72
73 end architecture basic ;
74
```

E.12. Source code of state_machine_controller.vhd

```
 1 -- state_machine_controller.vhd
 2 --
 3 --
 4
 5 call_stack_pointer_plus_one <= call_stack_pointer + 1 ;
 6 call_stack_pointer_minus_one <= call_stack_pointer - 1 ;
 7
 8 state_plus_one <= state + 1 ;
 9
10 call_stack_ram : entity dp_ram
11     generic map ( address_width => stack_pointer_register'length ,
12             data_width => state_register'length )
13     port map ( clock => clock , -- done in +ve cycle.
14             write => write_enable ,
15             address1 => call_stack_pointer_minus_one ,
16             address2 => call_stack_pointer_minus_one ,
17             data_in => value_to_be_stacked ,
18             data_out1 => call_stack_at_ptr_minus_one ) ;
19
20 state_machine_controller : process (
21         call_requested , reset , return_requested , clock ,
22         state_plus_one , call_stack_at_ptr_minus_one ,
23         call_stack_pointer_minus_one ,
24         call_stack_pointer_plus_one ,
25         call_state ) is
26     constant zero_call_state : state_register := ( others => '0' ) ;
27 begin
28     if ( clock'event )
29     and ( clock = '0' )
30     then
31         if ( reset = '1' )
32         then
33             state <= ( others => '0' ) ;
34             value_to_be_stacked <= ( others => '0' ) ;
35             write_enable <= '0' ;
36             call_stack_pointer <= ( others => '0' ) ;
37         else
38             -- CALL stacking operations
39             value_to_be_stacked <= state_plus_one ;
40
41             if ( call_requested = '0' )
42             and ( return_requested = '0' )
43             then
44                 -- CLOCK - just go to next state.
45                 state <= state_plus_one ;
46                 write_enable <= '0' ;
47             elsif ( call_requested = '0' )
48             and ( return_requested = '1' )
49             then
```

```
50                 -- Return
51                 state <= call_stack_at_ptr_minus_one ;
52                 call_stack_pointer <= call_stack_pointer_minus_one ;
53                 write_enable <= '0' ;
54             elsif ( call_requested = '1' )
55             and ( return_requested = '0' )
56             then
57                 -- a CALL
58                 if ( call_state = zero_call_state )
59                 then
60                     -- This is a CALL NOTHING statement.
61                     -- This is a dummy call that means "just go to
62                     -- the next state".
63                     state <= state_plus_one ;
64                     write_enable <= '0' ;
65                 else
66                     -- This is a CALL to a real state machine.
67                     state <= call_state ;
68                     call_stack_pointer <= call_stack_pointer_plus_one ;
69                     write_enable <= '1' ;
70                 end if ;
71             else
72                 -- a JUMP. This is a call without a stack.
73                 -- In the positive going clock cycle, we change
74                 -- the state.
75                 state <= call_state ;
76                 write_enable <= '0' ;
77             end if ;
78         end if ;
79     end if ;
80 end process state_machine_controller ;
81
```

E.13. Source code of types.vhd

```
1 -- types.vhd
2 --
3 -- Global type and constant definitions.
4 --
5
6 library ieee ;
7 use ieee . std_logic_1164 . all ;
8 use ieee . std_logic_arith . all ;
9 use ieee . std_logic_unsigned . all ;
10
11 package m68k_types is
12
13     -- Register types
14     subtype byte_register is std_logic_vector ( 7 downto 0 ) ;
15     subtype word_register is std_logic_vector ( 15 downto 0 ) ;
16     subtype dword_register is std_logic_vector ( 31 downto 0 ) ;
17
18     -- ALU internal operation codes
19     type alu_internal_op_type is
20         ( ALU_INT_OR , ALU_INT_AND , ALU_INT_SUB , ALU_INT_ADD ,
21         ALU_INT_EOR , ALU_INT_CMP , ALU_INT_REV_SUB , ALU_INT_REV_CMP ) ;
22
23 end package m68k_types ;
24
```

E.14. Source code of xilinx_dp_ram.vhd

```
1
2 library ieee ;
3 use ieee . std_logic_1164 . all ;
4 use ieee . std_logic_arith . all ;
5 use ieee . std_logic_unsigned . all ;
6
7 entity dp_ram is
8     generic ( address_width : integer := 5 ;
9               data_width : integer := 4 ) ;
```

```
10     port ( clock : in std_logic ;
11      write : in std_logic ;
12      address1 : in std_logic_vector (( address_width - 1 ) downto 0 ) ;
13      address2 : in std_logic_vector (( address_width - 1 ) downto 0 ) ;
14      data_in : in std_logic_vector (( data_width - 1 ) downto 0 ) ;
15      data_out1 : out std_logic_vector (( data_width - 1 ) downto 0 ) ;
16      data_out2 : out std_logic_vector (( data_width - 1 ) downto 0 ) ) ;
17 end dp_ram ;
18
19 architecture syn of dp_ram is
20     constant max_address : integer := ( 2 ** address_width ) - 1 ;
21
22     type ram_type is array ( max_address downto 0 )
23                     of std_logic_vector (( data_width - 1 ) downto 0 ) ;
24
25     signal RAM : ram_type ;
26     signal read_address1 : std_logic_vector (( address_width - 1 ) downto 0 ) ;
27     signal read_address2 : std_logic_vector (( address_width - 1 ) downto 0 ) ;
28 begin
29     ram_process : process ( clock ) is
30     begin
31         if ( clock'event )
32         and ( clock = '1' )
33         then
34             if ( write = '1' )
35             then
36                 RAM ( conv_integer ( address1 ) ) <= data_in ;
37             end if;
38             read_address1 <= address1 ;
39             read_address2 <= address2 ;
40         end if ;
41     end process ;
42
43     data_out1 <= RAM ( conv_integer ( read_address1 ) ) ;
44     data_out2 <= RAM ( conv_integer ( read_address2 ) ) ;
45 end syn ;
46
```

F. Test Program sources

F.1. Source code of `fib.c`

```
 1
 2 int main ()
 3 {
 4     char    * ptr = (char *) 0x8000 ;
 5
 6     while ( 1 )
 7     {
 8         int     a = 0 ;
 9         int     b = 1 ;
10         int     current = 0 ;
11
12         do {
13             (* ptr) = (char) ( current & 0xff ) ;
14
15             current = a + b ;
16             a = b ;
17             b = current ;
18         } while ( current <= 0x10000000 ) ;
19     }
20
21 }
22
```

F.2. Source code of `fvt.s`

```
 1
 2 #20
 3 clrtestmemsize = 0x100
```

```
 4 initialstack = 0x2000
 5 outputptr = 0x8000
 6 perbyte = 0x42
 7
 8 .text
 9
10 # a0 = output ptr
11 movel #outputptr,%a0
12
13 moveb #10,repflag
14
15 repeat:
16
17 moveb #0x1,(%a0)
18 # tests for Link
19 movew #initialstack,%a7
20 movew #0x1234,%d2
21 movel %d2,%a2
22 movel #0x12345678,initialstack-4
23
24 moveb #0x2,(%a0)
25
26 # sp = initialstack.  frame ptr = 0x1234
27 linkw %a2,#-40
28
29 moveb #0x3,(%a0)
30
31 # Check SP = ( initialstack - 4 ) - 40
32 movel %a7,%d3
33 cmpl #initialstack-44,%d3
34 bne fail
35
36 moveb #0x4,(%a0)
37
38 # Check (initialstack-4) = 0x1234
39 cmpl (initialstack-4),%d2
40 bne fail
41
42 moveb #0x5,(%a0)
43
44 # Check frame ptr = initialstack-4
45 movel %a2,%d3
46 cmpl #initialstack-4,%d3
47 bne  fail
48
49 # The Link instruction worked correctly!
50
51 moveb #0x6,(%a0)
52
53 # Now, unlink
54
55 unlk %a2
56
57 moveb #0x7,(%a0)
58
59 # Has SP been restored?
60 movel #initialstack,%d3
61 cmpl %a7,%d3
62 bne  fail
63
64 moveb #0x8,(%a0)
65
66 # Has the frame ptr been restored?
67 cmpl %a2,%d2
68 bne  fail
69
70 # Now, JSR and RTS
71 moveb #0x9,(%a0)
72 jsr testprocedure
73
74 returnpoint:
75
76 # And JMP
77 jmp ooh
```

```
 78
 79 # Should never get here:
 80 moveb #0x13,(%a0)
 81 bra  fail
 82
 83 ooh:
 84
 85 moveb #0x14,(%a0)
 86
 87 # Now test DBCC and postinc
 88
 89 move #clrtestmem,%a5
 90 move #clrtestmemsize-1,%d3
 91
 92 moveb #0x15,(%a0)
 93
 94 filler:
 95 moveb #perbyte,(%a5)+
 96 dbra %d3,filler
 97
 98 moveb #0x16,(%a0)
 99
100 # Is %d3 = -1?
101 cmpw #-1,%d3
102 bne fail
103
104 moveb #0x17,(%a0)
105
106 # Is %a5 correct?
107
108 movel %a5,%d4
109 cmpl #clrtestmem+clrtestmemsize,%d4
110 bne fail
111
112 moveb #0x18,(%a0)
113
114 # Is every value in the range correct?
115 jsr checkrange
116
117 # sum is in d7
118
119 moveb #0x19,(%a0)
120
121 cmpl #( perbyte * clrtestmemsize ),%d7
122 bne fail
123
124 moveb #0x20,(%a0)
125
126 # What about after a CLRB?
127
128 clrb clrtestmem+3
129
130 moveb #0x21,(%a0)
131
132 jsr checkrange
133
134 moveb #0x22,(%a0)
135 cmpl #(( perbyte * clrtestmemsize ) - perbyte ),%d7
136 bnes fail
137
138 # A CLRL?
139 clrl clrtestmem+16
140
141 moveb #0x23,(%a0)
142
143 jsr checkrange
144
145 moveb #0x24,(%a0)
146 cmpl #(( perbyte * clrtestmemsize ) - ( perbyte * 5 )),%d7
147 bnes fail
148
149 moveb #0x25,(%a0)
150
151 # Test LEA
```

```
152 # .word 0x4de8
153 # .word 28
154 # 0100 1101 1110 1000
155
156 lea %a0@(28),%a6
157
158 moveb #0x26,(%a0)
159
160 movel #( outputptr + 28 ),%d2
161 cmpl %a6,%d2
162 bnes fail
163
164 moveb #0x27,(%a0)
165
166 # Test PEA
167 pea %a0@(28)
168
169 moveb #0x28,(%a0)
170
171 # Check SP
172 movel #( initialstack - 4 ),%d3
173 cmpl %a7,%d3
174 bnes fail
175
176 moveb #0x29,(%a0)
177
178 # Check (SP)
179 cmpl (%a7),%d2
180 bnes fail
181
182 moveb #0x30,(%a0)
183
184 # Subtract 2 from SP.
185
186 moveq #2,%d2
187 suba %d2,%a7
188
189 # Test SCC
190 seq (%a7)
191 moveq #0,%d2
192 cmpb (%a7),%d2
193 bnes fail
194
195 moveb #0x31,(%a0)
196
197 sne (%a7)
198 moveq #-1,%d2
199 cmpb (%a7),%d2
200 bnes fail
201
202 moveb #0x32,(%a0)
203
204 # Has this been run before?
205 # Repeat the test until repflag = 0
206 subqb #1,repflag
207 bge repeat
208
209 moveb #0xa5,(%a0)
210
211 # This is actually a pass.
212 # But we go into the fail loop anyway with a success code (a5)
213
214 fail:
215
216 movel %a7,spout
217 movel %a6,fpout
218 movel %d3,d3out
219 movel %d4,d4out
220 failloop:
221 bras failloop
222
223 testprocedure:
224 moveb #0x10,(%a0)
225
```

```
226 # Check the stack pointer is correct
227 movel %a7,%d3
228 cmpl #initialstack-4,%d3
229 bnes fail
230
231 moveb #0x11,(%a0)
232
233 # Check the return point has been stacked correctly
234 movel (%a7),%d3
235 cmpl #returnpoint,%d3
236 bnes fail
237
238 moveb #0x12,(%a0)
239 rts
240
241 checkrange:
242 moveql #0,%d7
243 moveql #0,%d5
244 move #clrtestmemsize+clrtestmem,%d6
245 move #clrtestmem,%a6
246 checkrange_loop:
247 moveb (%a6),%d5
248 addl %d5,%d7
249 addq #1,%a6
250 cmpl %a6,%d6
251 bgts checkrange_loop
252 rts
253
254 .bss
255 regdumpspace:
256 spout:
257 .space 4
258 fpout:
259 .space 4
260 d3out:
261 .space 4
262 d4out:
263 .space 4
264 clrtestmem:
265 .space clrtestmemsize
266 repflag:
267 .space 1
268
```

F.3. Source code of 23instructions.s

```
 1 _start:
 2 addw a1,a2
 3 bra _start
 4 subl d6,d6
 5 adda a1,a2
 6 cmpa a2,a3
 7 addi #127,d4
 8 cmpi #222,_start
 9 cmpb d5,d6
10 addq #2,d7
11 clr d1
12 dbcc d4,_start
13 jmp _start
14 jsr _start
15 lea _start,a4
16 link a5,#8
17 unlk a1
18 nop
19 pea _start
20 rts
21 scc d6
22 tst d1
23 move %a2,%a3
24 moveq #0,%d3
25
```

G. State Machine Compiler sources

G.1. Source code of alu_optimisation.cc

```
 1
 2 #include "alu_optimisation.h"
 3
 4 /** ALU_Optimisation constructor
 5  *
 6  */
 7 ALU_Optimisation :: ALU_Optimisation () :
 8             Basic_Optimisation ( "alu_internal_op" , "alu_internal_op_type" )
 9 {
10     /* These ALU modes are always needed */
11     Add_Assert ( "ALU_INT_ADD" ) ;
12     Add_Assert ( "ALU_INT_SUB" ) ;
13     Add_Assert ( "ALU_INT_REV_SUB" ) ;
14 }
15
16 /** Notify
17  *
18  * This method is called with every possible opcode containing an
19  * ALU operation within the program. It allows the ALU to be optimised,
20  * removing unnecessary operations.
21  */
22 void ALU_Optimisation :: Notify ( unsigned opcode , char subtype )
23 {
24     switch ( subtype )
25     {
26         case '+' :
27         case '-' :      /* These are always needed. */
28                         break ;
29         case 'c' :      Add_Assert ( "ALU_INT_CMP" ) ;
30                         Add_Assert ( "ALU_INT_REV_CMP" ) ;
31                         break ;
32         case '^' :      Add_Assert ( "ALU_INT_EOR" ) ;
33                         break ;
34         case '&' :      Add_Assert ( "ALU_INT_AND" ) ;
35                         break ;
36         case '|' :      Add_Assert ( "ALU_INT_OR" ) ;
37                         break ;
38         default :       assert ( 0 ) ;
39     }
40 }
41
```

G.2. Source code of alu_optimisation.h

```
 1 #ifndef ALU_OPTIMISATION_H
 2 #define ALU_OPTIMISATION_H
 3
 4 #include "basic_optimisation.h"
 5
 6 class ALU_Optimisation : public Basic_Optimisation
 7 {
 8 public:
 9     ALU_Optimisation () ;
10
11     virtual void Notify ( unsigned opcode , char subtype ) ;
12 } ;
13
14 #endif
15
```

G.3. Source code of control.cc

```
 1
 2 #include <stdio.h>
 3 #include <regex.h>
 4 #include <string.h>
 5 #include <stdlib.h>
```

```
 6 #include <assert.h>
 7 #include <limits.h>
 8
 9 #include "utils.h"
10 #include "control.h"
11 #include "definitions.h"
12
13 /****** Control public methods ******/
14
15 /** Control constructor
16  *
17  * A new object is created to control the various components of the program.
18  * This object is fed a series of settings from a configuration, and
19  * when Generate_VHDL is called, it produces the VHDL to represent the
20  * processor.
21  */
22 Control :: Control ( const char * opcode_map_file ,
23                      const char * state_machine_directory ,
24                      const char * root_vhdl_input_file ,
25                      const char * vhdl_input_directory ,
26                      const char * initial_state_machine ,
27                      const char * required_opcode_list_file )
28 {
29     /* First compile the regular expressions used in parsing the VHDL */
30     int rc = regcomp ( & include_regex ,
31         "^[\t ]*INCLUDE +([^ \t]+)[ \t]*" , REG_EXTENDED ) ;
32     assert ( rc == 0 ) ;
33
34     rc = regcomp ( & insert_subtypes_regex ,
35         "^[\t ]*INSERT +SUBTYPES" , REG_EXTENDED ) ;
36     assert ( rc == 0 ) ;
37
38     rc = regcomp ( & insert_state_machine_regex ,
39         "^[\t ]*INSERT +STATE +MACHINE" , REG_EXTENDED ) ;
40     assert ( rc == 0 ) ;
41
42     rc = regcomp ( & insert_instruction_decoder_regex ,
43         "^[\t ]*INSERT +INSTRUCTION +DECODER" , REG_EXTENDED ) ;
44     assert ( rc == 0 ) ;
45
46     rc = regcomp ( & insert_optimisation_regex ,
47         "^[\t ]*INSERT +OPTIMI[ZS]ATION +([^ \t]+)[ \t]*" , REG_EXTENDED ) ;
48     assert ( rc == 0 ) ;
49
50     /* This regex is used for parsing objdump/opcode list
51      * files (Require_Opcodes_In_File) */
52     rc = regcomp ( & require_opcode_regex ,
53             require_opcode_expression , REG_EXTENDED | REG_ICASE ) ;
54     assert ( rc == 0 ) ;
55
56
57
58     opcode_database . Read_Opcode_Map ( opcode_map_file ) ;
59     optimisation_manager . Read_Opcode_Map ( opcode_map_file ) ;
60
61     /* Read in state machines */
62     sm_loader . Add_State_Machine_Directory ( state_machine_directory ) ;
63
64     Require_Opcodes_In_File ( required_opcode_list_file ) ;
65     /* This will have to change: */
66     //opcode_database . Require_Opcode ( 0x6000 ) ; // BRA
67     //opcode_database . Require_Opcode ( 0x5280 ) ; // ADDQ
68     //opcode_database . Require_Opcode ( 0x2001 ) ; // some move
69
70     /* Use dependencies to calculate which microcode is needed to
71      * run the program */
72     sm_loader . Require_Microsubs
73                     ( opcode_database . List_Required_Micro_Subroutines () ) ;
74
75     /* Build the master state machine for use as microcode. */
76     master_sm = sm_loader . Build_Master_Machine ( initial_state_machine ) ;
77
78     /* Finalise the opcode database, linking the instruction decoder
79      * to the microcode */
```

```
 80     opcode_database . Finalise_DFA ( master_sm ) ;
 81
 82     this -> vhdl_input_directory = Copy_String ( vhdl_input_directory ) ;
 83     this -> root_vhdl_input_file = Copy_String ( root_vhdl_input_file ) ;
 84 }
 85
 86 /** Control destructor
 87  */
 88 Control :: ~Control ()
 89 {
 90     regfree ( & include_regex ) ;
 91     regfree ( & insert_subtypes_regex ) ;
 92     regfree ( & insert_state_machine_regex ) ;
 93     regfree ( & insert_instruction_decoder_regex ) ;
 94     regfree ( & insert_optimisation_regex ) ;
 95     regfree ( & require_opcode_regex ) ;
 96
 97     delete [] root_vhdl_input_file ;
 98     delete [] vhdl_input_directory ;
 99 }
100
101 /** Generate_VHDL
102  *
103  * VHDL is generated and sent to the specified file.
104  */
105 void Control :: Generate_VHDL ( FILE * output )
106 {
107     Add_VHDL_Source_File ( root_vhdl_input_file , output ) ;
108 }
109
110 /****** Control private methods ******/
111
112 /** Add_VHDL_Source_File
113  *
114  * Recursively process the given VHDL source file, inserting the
115  * appropriate information in the right places. This procedure
116  * must be passed a finalised master state machine and opcode database.
117  */
118 void Control :: Add_VHDL_Source_File ( const char * filename , FILE * output )
119 {
120     FILE *      input ;
121
122     regmatch_t  matches [ 3 ] ;
123     char        str [ MAX_LINE_LEN + 1 ] ;
124
125     /* First try to open the filename using the default path
126      * provided to the constructor */
127     char *      abs_filename = new char [ strlen ( vhdl_input_directory ) +
128                         strlen ( filename ) + 2 ] ;
129     strcpy ( abs_filename , vhdl_input_directory ) ;
130     strcat ( abs_filename , "/" ) ;
131     strcat ( abs_filename , filename ) ;
132     input = fopen ( abs_filename , "rt" ) ;
133
134     if ( input == 0L )
135     {
136         /* Now try reading it from the current directory */
137         strcpy ( abs_filename , filename ) ;
138         input = fopen ( abs_filename , "rt" ) ;
139     }
140
141     MESSAGE2 ( "Reading VHDL source '%s'.\n" , abs_filename ) ;
142
143     if ( input == 0L )
144     {
145         throw new File_Access_Exception ( filename ) ;
146     }
147
148     while ( fgets ( str , MAX_LINE_LEN , input ) != NULL )
149     {
150         Remove_Trailing_Newlines ( str ) ;
151
152         if ( regexec ( & include_regex , str , 2 , matches , 0 ) == 0 )
153         {
```

```
154             /* An include statement. Read the specified file. */
155             char * new_filename = Get_Regex_Match ( str , & matches [ 1 ] ) ;
156 
157             Add_VHDL_Source_File ( new_filename , output ) ;
158 
159             delete [] new_filename ;
160         } else if ( regexec ( & insert_subtypes_regex , str , 1 ,
161                     matches , 0 ) == 0 )
162         {
163             /* This means that two subtype definitions should
164             be added to the VHDL. They define state_register
165             and stack_pointer_register */
166 
167             fprintf ( output ,
168                 "\tsubtype state_register is std_logic_vector "
169                         "( %d downto 0 ) ;\n"
170                 "\tsubtype stack_pointer_register is std_logic_vector "
171                         "( %d downto 0 ) ;\n" ,
172                 master_sm -> Get_Width_Of_State_Number () - 1 ,
173                 Get_Number_Of_Bits_Needed_For ( STACK_SIZE - 1 ) - 1 ) ;
174         } else if ( regexec ( & insert_optimisation_regex , str , 2 ,
175                     matches , 0 ) == 0 )
176         {
177             /* An optimisation statement. */
178             char * ot = Get_Regex_Match ( str , & matches [ 1 ] ) ;
179 
180             optimisation_manager . Generate_VHDL ( output ,
181                                 Optimisation_Record ( ot ) ) ;
182 
183             delete [] ot ;
184         } else if ( regexec ( & insert_state_machine_regex , str , 1 ,
185                     matches , 0 ) == 0 )
186         {
187             /* This means that the state machine defintion
188             should be added to the VHDL. */
189 
190             master_sm -> Compile_Machine ( output ) ;
191         } else if ( regexec ( & insert_instruction_decoder_regex , str , 1 ,
192                     matches , 0 ) == 0 )
193         {
194             /* This means that the instruction decoder
195             should be added to the VHDL. */
196 
197             opcode_database . Generate_VHDL ( output ) ;
198         } else {
199             /* The line will be added to the outgoing VHDL if it
200             has any non-whitespace characters on it. */
201 
202             if ( String_Contains_Non_Whitespace ( str ) )
203             {
204                 fputs ( str , output ) ;
205                 fputs ( "\n" , output ) ;
206             }
207         }
208     }
209     fclose ( input ) ;
210 
211     delete [] abs_filename ;
212 }
213 
214 /** Require_Opcodes_In_File
215  *
216  * For each opcode listed in the file, run Require_Opcode.
217  * The file may take one of the following formats:
218  *    GNU objdump output
219  *    a list of opcodes, one per line, in hex format.
220  */
221 void Control :: Require_Opcodes_In_File ( const char * file )
222 {
223     FILE *      input ;
224     regmatch_t  matches [ 4 ] ;
225     char        str [ MAX_LINE_LEN + 1 ] ;
226     int         count = 0 ;
227     int         line_no = 0 ;
```

```
228
229     input = fopen ( file , "rt" ) ;
230
231     if ( input == 0L )
232     {
233         throw new File_Access_Exception ( file ) ;
234     }
235
236     MESSAGE ( "Reading required opcodes list file '%s'\n" , file ) ;
237     while ( fgets ( str , MAX_LINE_LEN , input ) != NULL )
238     {
239         line_no ++ ;
240
241         Remove_Trailing_Newlines ( str ) ;
242         Remove_Whitespace_From_Ends ( str ) ;
243
244         if ( regexec ( & require_opcode_regex , str , 3 , matches , 0 ) == 0 )
245         {
246             /* Ah, a match. The 2nd field should be the opcode */
247             char *  opcode_str = Get_Regex_Match ( str , & matches [ 2 ] ) ;
248             int     opcode = strtol ( opcode_str , 0L , 16 ) ;
249
250             opcode_database . Require_Opcode ( opcode ) ;
251
252             optimisation_manager . Notify ( opcode ) ;
253
254             count ++ ;
255
256             delete [] opcode_str ;
257         } else {
258             MESSAGE2 ( "Unable to decode %s line %d: %s\n" ,
259                         file , line_no , str ) ;
260         }
261     }
262     fclose ( input ) ;
263
264     MESSAGE ( "%d opcodes read.\n" , count ) ;
265     if ( count == 0 )
266     {
267         throw new No_Opcodes_Read_Exception ( file ) ;
268     }
269 }
270
271 const char * Control :: require_opcode_expression =
272                 "^([0-9a-f]+:[\t ]+|0x|)([0-9a-f]{4})[ \t]" ;
273
```

G.4. Source code of `control.h`

```
 1 #ifndef CONTROL_H
 2 #define CONTROL_H
 3
 4 #include <stdio.h>
 5 #include <regex.h>
 6
 7 #include "state_machine.h"
 8 #include "opcode_database.h"
 9 #include "state_machine_loader.h"
10 #include "optimisation.h"
11
12 class Control
13 {
14 public:
15     Control ( const char * opcode_map_file ,
16                     const char * state_machine_directory ,
17                     const char * root_vhdl_input_file ,
18                     const char * vhdl_input_directory ,
19                     const char * initial_state_machine ,
20                     const char * required_opcode_list_file ) ;
21     virtual ~Control () ;
22     void Generate_VHDL ( FILE * output ) ;
23
24 private:
```

```
25      void Add_VHDL_Source_File ( const char * filename , FILE * output ) ;
26      void Require_Opcodes_In_File ( const char * file ) ;
27
28      Opcode_Database opcode_database ;
29      State_Machine * master_sm ;
30      State_Machine_Loader
31                      sm_loader ;
32      Optimisation_Manager
33                      optimisation_manager ;
34      regex_t         include_regex ;
35      regex_t         insert_subtypes_regex ;
36      regex_t         insert_state_machine_regex ;
37      regex_t         insert_instruction_decoder_regex ;
38      regex_t         insert_optimisation_regex ;
39      regex_t         require_opcode_regex ;
40      const char *    root_vhdl_input_file ;
41      const char *    vhdl_input_directory ;
42
43      static const char * require_opcode_expression ;
44  } ;
45
46  #endif
47
```

G.5. Source code of `main.cc`

```
 1
 2  #include "opcode_database.h"
 3  #include "state_machine_loader.h"
 4  #include "exceptions.h"
 5  #include "control.h"
 6  #include "utils.h"
 7  #include "settings.h"
 8
 9  int main ( int argc , char * argv [] )
10  {
11
12      try {
13          Settings * s ;
14
15          g_verbose_setting = VERBOSE_MEDIUM ;
16
17          MESSAGE ( "  ** State Machine Compiler **\n"
18                  "Binary build time: " __TIME__ " " __DATE__ "\n\n" ) ;
19
20          /* Read the settings */
21          if ( argc > 1 )
22          {
23              s = new Settings ( argv [ 1 ] ) ;
24              if ( argc > 2 )
25              {
26                  s -> Set_Required_Opcode_File ( argv [ 2 ] ) ;
27              }
28          } else {
29              s = new Settings ( "smc.ini" ) ;
30          }
31
32          g_verbose_setting = s -> Get_Verbose_Level () ;
33
34          Control c ( s -> Get_Opcode_Map_Filename () ,
35                      s -> Get_State_Machine_Directory () ,
36                      s -> Get_Root_VHDL_Input_Filename () ,
37                      s -> Get_VHDL_Input_Directory () ,
38                      s -> Get_Initial_State_Machine () ,
39                      s -> Get_Required_Opcode_File () ) ;
40
41          FILE * fd = fopen ( s -> Get_Output_Filename () , "wt" ) ;
42          if ( fd == 0L )
43          {
44              throw new File_Access_Exception ( s -> Get_Output_Filename () ) ;
45          }
46          c . Generate_VHDL ( fd ) ;
47          fclose ( fd ) ;
```

```
48
49          MESSAGE ( "Completed successfully.\n" ) ;
50
51          delete s ;
52
53          return 0 ;
54      } catch ( Basic_Exception * e )
55      {
56          MESSAGE ( "An exception was thrown:\n" ) ;
57          e -> PrintMessage () ;
58
59          return 1 ;
60      }
61  }
62
```

G.6. Source code of ndfa_dag.cc

```
 1
 2
 3  #include <stdio.h>
 4  #include <assert.h>
 5
 6  #include <set>
 7  #include <algorithm>
 8
 9
10  #include "ndfa_dag.h"
11  #include "utils.h"
12
13
14
15  /****** NDFA_DAG public methods ******/
16
17  /** NDFA_DAG constructor
18   *
19   * This constructor produces an empty NDFA.
20   */
21  NDFA_DAG :: NDFA_DAG () : NDFA_Node ( 0 )
22  {
23      MESSAGE4 ( "Creating empty NDFA.\n" ) ;
24      is_accept_all_ndfa = false ;
25  }
26
27  /** NDFA_DAG constructor
28   *
29   * This constructor produces an NDFA that accepts all bit patterns.
30   * The accept state information is the one provided.
31   */
32  NDFA_DAG :: NDFA_DAG ( Accept_State * asi ) : NDFA_Node ( 0 )
33  {
34      MESSAGE4 ( "Creating accept-all NDFA.\n" ) ;
35
36      NDFA_Node *     current_node ;
37
38      current_node = this ;
39
40      for ( int i = 0 ; i < BITS_PER_OPCODE ; i ++ )
41      {
42          accept_all_list [ i ] = current_node ;
43          current_node = current_node -> Add_Transition ( true , true ) ;
44      }
45
46      current_node -> Make_Accept_State ( asi ) ;
47
48      is_accept_all_ndfa = true ;
49  }
50
51
52
53  /** Reject_Pattern
54   *
55   * Makes the NDFA reject a particular bit pattern.
```

```
 */
void NDFA_DAG :: Reject_Pattern ( int start_bit , const char * pattern )
{
    MESSAGE4 ( "Rejecting bit pattern '%s' start %d.\n" ,
                        pattern , start_bit ) ;

    assert (( start_bit >= 0 )
            && ( start_bit < BITS_PER_OPCODE ) ) ;

    NDFA_Node *     current_node ;
    unsigned        i ;

    /* First add a series of transitions from the root to start_bit - 1,
     * that will accept any pattern. As a special case, if this is an
     * "accept-all" NDFA (created by the 2nd constructor), this set of
     * transitions already exists, so we can use an entry in the
     * accept_all_list. */

    if ( is_accept_all_ndfa )
    {
        current_node = accept_all_list [ start_bit ] ;
    } else {
        current_node = this ;

        for ( i = 0 ; i < (unsigned) start_bit ; i ++ )
        {
            current_node = current_node -> Add_Transition ( true , true ) ;
        }
    }

    /* Now add a series of transitions from start_bit - 1 to the end
     * of the pattern that reject that specific pattern */
    for ( i = 0 ; i < strlen ( pattern ) ; i ++ )
    {
        if ( pattern [ i ] == '0' )
        {
            current_node = current_node -> Add_Transition ( true , false ) ;
        } else if ( pattern [ i ] == '1' )
        {
            current_node = current_node -> Add_Transition ( false , true ) ;
        } else {
            assert ( 0 ) ;
        }
    }

    current_node -> Make_Reject_State () ;
}


/** Get_Accept_State
 *
 * Translate an opcode number into an accept state.
 * If translation fails, 0 is returned.
 */
Accept_State * NDFA_DAG :: Get_Accept_State ( unsigned opcode )
{
    assert ( Is_Deterministic () ) ;

    NDFA_Node * node = this ;

    while (( node != 0L )
    && ( node -> Get_Accept_State () == 0L ))
    {
        if (( opcode >> (( BITS_PER_OPCODE - 1 ) -
                    node -> Get_Test_Bit_Number () ) ) & 1 )
        {
            node = node -> DFA_Transition ( 1 ) ;
        } else {
            node = node -> DFA_Transition ( 0 ) ;
        }
    }
    if ( node == 0L )
    {
        return 0L ;
```

```
130     } else {
131         return node -> Get_Accept_State () ;
132     }
133 }
134
135 /** Enable_Accept_State
136  *
137  * Translate an opcode number into an accept state. During the tree
138  * traversal, every tree node visited has it's "visit" counter incremented.
139  * This tells the compression routines that this tree path will be needed.
140  * The accept state's Enable method is called when it is reached.
141  * If no accept state is found, 0 is returned.
142  */
143 Accept_State * NDFA_DAG :: Enable_Accept_State ( unsigned opcode )
144 {
145     assert ( Is_Deterministic () ) ;
146
147     NDFA_Node * node = this ;
148
149     while (( node != 0L )
150     && ( node -> Get_Accept_State () == 0L ))
151     {
152         node -> Visit () ;
153
154         if (( opcode >> (( BITS_PER_OPCODE - 1 ) -
155                     node -> Get_Test_Bit_Number () ) ) & 1 )
156         {
157             node = node -> DFA_Transition ( 1 ) ;
158         } else {
159             node = node -> DFA_Transition ( 0 ) ;
160         }
161     }
162     if ( node == 0L )
163     {
164         return 0L ;
165     } else {
166         node -> Visit () ;
167         node -> Get_Accept_State () -> Enable () ;
168         return node -> Get_Accept_State () ;
169     }
170 }
171
172
173 /** Generate_VHDL
174  *
175  * Produces VHDL, sent to the specified device, for the DFA,
176  * which is first checked for determinism.
177  */
178 void NDFA_DAG :: Generate_VHDL ( FILE * fd )
179 {
180     assert ( Is_Deterministic () ) ;
181     DFA_To_VHDL ( this , fd , "" ) ;
182 }
183
184
185
186
187
188 /****** NDFA_DAG private methods ******/
189
190 void NDFA_DAG :: DFA_To_VHDL ( NDFA_Node * root_node , FILE * fd ,
191                                    const char * indent )
192 {
193     if ( root_node == 0L )
194     {
195         printf ( "Fault 1\n" ) ;
196     }
197     assert ( root_node != 0L ) ;
198
199     char *  new_indent = new char [ strlen ( indent ) + 4 ] ;
200     int     bit_num = ( BITS_PER_OPCODE - 1 ) -
201                         root_node -> Get_Test_Bit_Number () ;
202
203     strcpy ( new_indent , indent ) ;
```

```
204     strcat ( new_indent , "  " ) ;
205
206     if ( root_node -> Get_Accept_State () != 0L )
207     {
208         root_node -> Get_Accept_State () ->
209                             Accept_State_To_VHDL ( fd , indent ) ;
210     } else {
211         /* Compressed DFA nodes always have two children */
212         fprintf ( fd , "%sif ir ( %d ) = '1' then\n" , indent , bit_num ) ;
213         DFA_To_VHDL ( root_node -> DFA_Transition ( 1 ) , fd , new_indent ) ;
214         fprintf ( fd , "%selse --ir(%d)= '0'\n" , indent , bit_num ) ;
215         DFA_To_VHDL ( root_node -> DFA_Transition ( 0 ) , fd , new_indent ) ;
216         fprintf ( fd , "%send if ;\n" , indent ) ;
217     }
218     delete [] new_indent ;
219 }
220
221
```

G.7. Source code of ndfa_dag.h

```
 1
 2
 3 #ifndef NDFA_TREE_H
 4 #define NDFA_TREE_H
 5
 6
 7
 8 #include "ndfa_node.h"
 9 #include "ndfa_accept_state.h"
10
11
12 class NDFA_DAG : public NDFA_Node
13 {
14 public:
15     NDFA_DAG () ;
16     NDFA_DAG ( Accept_State * accept_state ) ;
17
18
19     void Reject_Pattern ( int start_bit , const char * pattern ) ;
20
21     Accept_State * Get_Accept_State ( unsigned opcode ) ;
22     Accept_State * Enable_Accept_State ( unsigned opcode ) ;
23
24     void Generate_VHDL ( FILE * fd ) ;
25
26
27 private:
28     void DFA_To_VHDL ( NDFA_Node * root_node , FILE * fd ,
29                                     const char * indent ) ;
30
31
32     NDFA_Node * actual_root ;
33     NDFA_Node * accept_all_list [ BITS_PER_OPCODE ] ;
34     bool        is_accept_all_ndfa ;
35 } ;
36
37
38
39
40 #endif
41
```

G.8. Source code of ndfa_node.cc

```
 1
 2 #include <stdio.h>
 3 #include <assert.h>
 4
 5 #include <set>
 6 #include <algorithm>
```

```
 7
 8 #include "ndfa_node.h"
 9 #include "utils.h"
10
11 /****** NDFA_Node public methods ******/
12
13 /** NDFA_Node constructor
14  *
15  * A new node for an NDFA is produced. It is neither an accept nor
16  * reject state, and has no transitions. The bit number to be tested
17  * for the transitions is given as a parameter to the constructor.
18  */
19 NDFA_Node :: NDFA_Node ( int test_bit_number )
20 {
21     assert (( test_bit_number >= 0 )
22             && ( test_bit_number < ( BITS_PER_OPCODE + 1 ))) ;
23
24     accept_state = 0L ;
25     is_accept_state = false ;
26     is_reject_state = false ;
27     this -> test_bit_number = test_bit_number ;
28
29     for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
30     {
31         transitions [ ts ] . clear () ;
32     }
33
34     is_deterministic = false ;
35
36     visits = 0 ;
37 }
38
39 /** NDFA_Node destructor
40  *
41  * The NDFA node is deleted along with all its children. All NDFA-specific
42  * memory is freed:- no attempt is made to free the accept state memory if
43  * any.
44  */
45 NDFA_Node :: ~NDFA_Node ()
46 {
47     Delete_Children () ;
48 }
49
50 /** Add_Transition
51  *
52  * Adds a transition from this node to a new one. The new node is
53  * returned. Parameters specify whether the transition is on '0',
54  * '1', or both.
55  */
56 NDFA_Node * NDFA_Node :: Add_Transition ( bool transition_on_zero ,
57                                           bool transition_on_one )
58 {
59     assert ( transition_on_zero || transition_on_one ) ;
60
61     NDFA_Node * nn = new NDFA_Node ( test_bit_number + 1 ) ;
62
63     if ( transition_on_zero )
64     {
65         transitions [ 0 ] . insert ( nn ) ;
66     }
67     if ( transition_on_one )
68     {
69         transitions [ 1 ] . insert ( nn ) ;
70     }
71
72     Check_For_Determinism () ;
73     return nn ;
74 }
75
76 /** Make_Reject_State
77  *
78  * Makes this node a reject state.
79  */
80 void NDFA_Node :: Make_Reject_State ()
```

```
 81 {
 82     is_reject_state = true ;
 83     is_accept_state = false ;
 84     Delete_Children () ;
 85     Check_For_Determinism () ;
 86 }
 87
 88 /** Make_Accept_State
 89  *
 90  * Makes this node an accept state.
 91  * If this node is already an accept state, the new accept state
 92  * information must be the same as the old. If it is not, an exception
 93  * (SharedAcceptStateException) will be thrown.
 94  * If this node is already a reject state, nothing happens. (Reject
 95  * takes priority over accept).
 96  */
 97 void NDFA_Node :: Make_Accept_State ( Accept_State * accept_info )
 98 {
 99     if ( is_reject_state )
100     {
101         return ;
102     }
103
104     if ( is_accept_state )
105     {
106         if ( accept_state != accept_info )
107         {
108             assert ( accept_state != 0L ) ;
109             assert ( accept_info != 0L ) ;
110
111             /* An NDFA node cannot possibly be two different types of
112             accept states. This would mean that two opcodes had the
113             same bit pattern but a different meaning. */
114             throw new SharedAcceptStateException (
115                         accept_state , accept_info ) ;
116         }
117     }
118     is_accept_state = true ;
119     is_reject_state = false ;
120     accept_state = accept_info ;
121
122     Check_For_Determinism () ;
123 }
124
125 /** Merge_In
126  *
127  * The provided NDFA is copied into this one. A deep copy is made of all
128  * nodes in the 2nd NDFA, meaning it can be freely deleted. However,
129  * shallow copies are made of any accept_state records.
130  */
131 void NDFA_Node :: Merge_In ( NDFA_Node * source_root ,
132                                 bool copy_reject_states )
133 {
134     NDFA_Node   * new_ndfa_node ;
135
136     /* Copy an accept state from source to this, checking to ensure
137      * that an existing accept state is not overwritten. */
138     if ( source_root -> is_accept_state )
139     {
140         Make_Accept_State ( source_root -> accept_state ) ;
141     }
142
143     /* Copy all transitions from source to target */
144     for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
145     {
146         NDFA_Node_Set_Iterator  iter ;
147         NDFA_Node *             item ;
148
149         iter = source_root -> transitions [ ts ] . begin () ;
150
151         while ( iter != source_root -> transitions [ ts ] . end () )
152         {
153             item = (* iter) ;
154
```

```
155             new_ndfa_node = new NDFA_Node
156                                 ( source_root -> test_bit_number + 1 ) ;
157
158             new_ndfa_node -> Merge_In ( item , copy_reject_states ) ;
159
160             this -> transitions [ ts ] . insert ( new_ndfa_node ) ;
161
162             iter ++ ;
163         }
164     }
165
166     if (( source_root -> is_reject_state )
167     && ( copy_reject_states ))
168     {
169         this -> is_reject_state = true ;
170         this -> is_accept_state = false ;
171     }
172
173     Check_For_Determinism () ;
174 }
175
176 /** Delete_Dead_Branches
177  *
178  * A dead branch is one with either:
179  * - no _enabled_ accept states amongst any of its descendants.
180  * - a zero visit count (indicating that it will never be used in practice)
181  * This function tests to see if "this"
182  * is the parent of any dead branches.  Any descendant dead branches are
183  * removed and deallocated.
184  *
185  * It returns TRUE if children were removed.
186  */
187 bool NDFA_Node :: Delete_Dead_Branches ()
188 {
189     NDFA_Node_Set_Iterator  iter ;
190     NDFA_Node *             item ;
191     bool                    is_dead_branch = false ;
192     bool                    no_children = true ;
193
194     /* First, eliminate dead branches from the children */
195     for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
196     {
197         iter = this -> transitions [ ts ] . begin () ;
198         while ( iter != this -> transitions [ ts ] . end () )
199         {
200             item = (* iter) ;
201
202             if ( item -> Delete_Dead_Branches () )
203             {
204                 iter ++ ;
205                 Delete_Item_Before ( & this -> transitions [ ts ] ,
206                                      iter ) ;
207             } else {
208                 iter ++ ;
209             }
210         }
211
212         if ( ! this -> transitions [ ts ] . empty () )
213         {
214             no_children = false ;
215         }
216     }
217
218     /* This is a dead branch if both the branches off it are NULL
219     (no children - i.e. all_sub_branches_dead == true) and either :
220         1. it is not an accept state
221      or 2. it is an accept state, but the accept state has been disabled
222      or 3. the visit counter is zero (in which case, the same will have been
223             true for the children, so there won't be any children)
224
225     Return false iff this is a dead branch.*/
226
227     if ( no_children )
228     {
```

```
229         if ( visits == 0 )
230         {
231             is_dead_branch = true ;
232         }
233         if ( this -> is_accept_state )
234         {
235             if ( ! this -> accept_state -> Is_Enabled () )
236             {
237                 is_dead_branch = true ;
238             }
239         } else {
240             is_dead_branch = true ;
241         }
242     }
243 
244     return is_dead_branch ;
245 }
246 
247 /** Compress
248  *
249  * Several algorithms are applied recursively to the NDFA tree to reduce the
250  * number of nodes while maintaining equivalence.
251  *
252  * 1. Nodes with one child are replaced by the child.
253  * 2. Nodes with two children are replaced by either child if the
254  *    children are clearly equivalent (both the same accept state,
255  *    or both reject states)
256  *
257  * These are surprisingly effective. Unfortunately, once compressed,
258  * the tree cannot be converted to a DFA again, because
259  * the Make_Deterministic algorithm is unable to handle the idea that
260  * nodes at the same tree depth might be testing different bits.
261  *
262  * The tree must be a DFA before it is compressed.
263  */
264 void NDFA_Node :: Compress ()
265 {
266     NDFA_Node *             compressed_out_node ;
267     NDFA_Node_Set_Iterator  iter ;
268     NDFA_Node *             item ;
269     int                     ts ;
270 
271     assert ( is_deterministic ) ;
272 
273     MESSAGE4 ( "Compress() begins bit = %d\n" , test_bit_number ) ;
274 
275     /* Recurse down and do the children if any. */
276     for ( ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
277     {
278         iter = this -> transitions [ ts ] . begin () ;
279         while ( iter != this -> transitions [ ts ] . end () )
280         {
281             item = (* iter) ;
282 
283             /* Is this child a reject state? If so, it can be removed. */
284             if ( item -> is_reject_state )
285             {
286                 /* a quick sanity check assures that the tree is
287                  * sane - no state is a reject and accept state */
288                 assert ( ( ! item -> is_accept_state )
289                     && item -> transitions [ 0 ] . empty ()
290                     && item -> transitions [ 1 ] . empty () ) ;
291                 iter ++ ;
292                 Delete_Item_Before ( & this -> transitions [ ts ] ,
293                                      iter ) ;
294                 MESSAGE4 ( "Compress() removed reject state bit = %d\n" ,
295                                           test_bit_number ) ;
296             } else {
297                 /* Attempt to compress it */
298                 item -> Compress () ;
299 
300                 iter ++ ;
301             }
302         }
```

```
303     }
304
305     NDFA_Node_Set * tset0 = & ( this -> transitions [ 0 ] ) ;
306     NDFA_Node_Set * tset1 = & ( this -> transitions [ 1 ] ) ;
307
308     /* Compressions are possible if a node has only one child. */
309     if ( tset1 -> empty ()
310     && ( tset0 -> size () == 1 ))
311     {
312         MESSAGE4 ( "Compress() node compressed out (type 1) bit = %d\n" ,
313                                     test_bit_number ) ;
314
315         compressed_out_node = DFA_Transition ( 0 ) ;
316     } else if ( tset0 -> empty ()
317     && ( tset1 -> size () == 1 ))
318     {
319         MESSAGE4 ( "Compress() node compressed out (type 2) bit = %d\n" ,
320                                     test_bit_number ) ;
321
322         compressed_out_node = DFA_Transition ( 1 ) ;
323     /* or if there are two children, but they are both
324     accept states of the same type. */
325     } else if ( (( tset1 -> size () == 1 )
326     && ( tset0 -> size () == 1 )
327     && ( DFA_Transition ( 1 ) -> is_accept_state )
328     && ( DFA_Transition ( 0 ) -> is_accept_state )
329     && ( DFA_Transition ( 1 ) -> accept_state ==
330                         DFA_Transition ( 0 ) -> accept_state )
331     && ( DFA_Transition ( 1 ) -> test_bit_number ==
332                         DFA_Transition ( 0 ) -> test_bit_number ))
333     /* or if there are two children, but they are both reject states. */
334     || (( tset1 -> size () == 1 )
335     && ( tset0 -> size () == 1 )
336     && ( DFA_Transition ( 1 ) -> is_reject_state )
337     && ( DFA_Transition ( 0 ) -> is_reject_state )) )
338     {
339         MESSAGE4 ( "Compress() node compressed out (type 3) bit = %d\n" ,
340                                     test_bit_number ) ;
341
342         compressed_out_node = DFA_Transition ( 1 ) ;
343                 /* or tset0.. doesn't matter, they're the same. */
344
345         if ( compressed_out_node != DFA_Transition ( 0 ) )
346         {
347             /* The two nodes are equivalent but are at different
348              * memory locations, so the 0 node must be deleted.
349              * (It will soon be inaccessible). */
350             delete DFA_Transition ( 0 ) ;
351         }
352     } else {
353         compressed_out_node = 0L ;
354     }
355
356     if ( compressed_out_node != 0L )
357     {
358         /* Move the grandchildren up one generation.
359         The node we are looking at, this, Takes on the
360         properties of the compressed_out_node. */
361         Replace_With ( compressed_out_node ) ;
362     }
363     MESSAGE4 ( "Compress() ends bit = %d\n" , test_bit_number ) ;
364 }
365
366 /** Print_NDFA_DAG
367  *
368  * The NDFA tree is drawn out in ASCII format to the specified device.
369  */
370 void NDFA_Node :: Print_NDFA_DAG ( bool with_pointers , FILE * fd ,
371                                       const char * old_tab_str )
372 {
373     if ( is_accept_state )
374     {
375         assert ( accept_state != 0L ) ;
376
```

```
377         accept_state -> Print_Info ( fd , old_tab_str ) ;
378         if ( with_pointers )
379         {
380             fprintf ( fd , "%s     ptr=%p\n" , old_tab_str , this ) ;
381         }
382     } else if ( is_reject_state )
383     {
384         if ( with_pointers )
385         {
386             fprintf ( fd , "%s   REJECT ptr=%p\n" , old_tab_str , this ) ;
387         } else {
388             fprintf ( fd , "%s   REJECT\n" , old_tab_str ) ;
389         }
390     } else {
391         NDFA_Node_Set_Iterator  iter ;
392         NDFA_Node *             item ;
393         char *                  tab_str =
394                                     new char [ strlen ( old_tab_str ) + 4 ] ;
395 
396         strcpy ( tab_str , old_tab_str ) ;
397         strcat ( tab_str , "1  " ) ;
398         fprintf ( fd , "%s+-ir(%d)=one:\n" , tab_str ,
399                 BITS_PER_OPCODE - 1 - test_bit_number ) ;
400 
401         iter = transitions [ 1 ] . begin () ;
402         while ( iter != transitions [ 1 ] . end () )
403         {
404             item = (* iter) ;
405 
406             item -> Print_NDFA_DAG ( with_pointers , fd , tab_str ) ;
407             iter ++ ;
408         }
409         fprintf ( fd , "%s\n" , tab_str ) ;
410 
411         strcpy ( tab_str , old_tab_str ) ;
412         strcat ( tab_str , "0  " ) ;
413         fprintf ( fd , "%s+-ir(%d)=zero:\n" , tab_str ,
414                 BITS_PER_OPCODE - 1 - test_bit_number ) ;
415 
416         iter = transitions [ 0 ] . begin () ;
417         while ( iter != transitions [ 0 ] . end () )
418         {
419             item = (* iter) ;
420 
421             item -> Print_NDFA_DAG ( with_pointers , fd , tab_str ) ;
422             iter ++ ;
423         }
424         fprintf ( fd , "%s\n" , tab_str ) ;
425 
426         delete [] tab_str ;
427     }
428 }
429 
430 /** Get_Size
431  *
432  * Returns the number of nodes in the tree.
433  */
434 int NDFA_Node :: Get_Size ()
435 {
436     NDFA_Node_Set_Iterator  iter ;
437     NDFA_Node *             item ;
438     int                     count = 1 ;
439 
440     if (( ! is_reject_state )
441     && ( ! is_accept_state ))
442     {
443         for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
444         {
445             iter = transitions [ ts ] . begin () ;
446             while ( iter != transitions [ ts ] . end () )
447             {
448                 item = (* iter) ;
449 
450                 count += item -> Get_Size () ;
```

```
451                 iter ++ ;
452             }
453         }
454     }
455 
456     return count ;
457 }
458 
459 /** Make_Deterministic
460  *
461  * The NDFA is converted to a DFA. A depth-bounded depth first tree
462  * algorithm is used - Transform_NDFA_To_DFA is the procedure that
463  * actually does the work. This algorithm is used because it avoids
464  * the possibility that a very long branch of the NDFA may be uselessly
465  * converted to a DFA before it is realised that much of it will be
466  * rejected.
467  *
468  * Shallow copies of accept state pointers in the NDFA are made. Checking
469  * is done to ensure there are no ambiguous accept states. If a state is
470  * both an accept and a reject state, it is assumed to be a reject state.
471  *
472  * The return value is the number of nodes in the DFA.
473  *
474  * This will not work correctly on all compressed trees - an assertion
475  * will fail if the tree has been compressed. See the comment for
476  * Compress().
477  */
478 void NDFA_Node :: Make_Deterministic ()
479 {
480     MESSAGE4 ( "Make_Deterministic ()\n" ) ;
481 
482     /* Create a new root for the DFA */
483     NDFA_Node * dfa_root = new NDFA_Node ( test_bit_number ) ;
484 
485     /* Transform the NDFA into a DFA. This is a depth-bounded
486      * depth first tree operation - the depth is increased until
487      * the operation finishes. */
488     for ( int i = test_bit_number ; i <= BITS_PER_OPCODE ; i ++ )
489     {
490         Transform_NDFA_To_DFA ( dfa_root , i ) ;
491     }
492 
493     /* Remove children of the NDFA tree root */
494     Delete_Children () ;
495 
496     /* Destroy the NDFA tree, replacing it with the DFA tree. */
497     Replace_With ( dfa_root ) ;
498 
499     Check_For_Determinism () ;
500     assert ( is_deterministic ) ;
501 }
502 
503 /** Get_Accept_State
504  *
505  * Returns the accept state information for the node (if any) or 0.
506  */
507 Accept_State * NDFA_Node :: Get_Accept_State ()
508 {
509     if ( is_accept_state )
510     {
511         return accept_state ;
512     } else {
513         return 0L ;
514     }
515 }
516 
517 /** DFA_Transition
518  *
519  * Returns the node reached by a transition ts. It returns 0L if
520  * no node is reached.
521  */
522 NDFA_Node * NDFA_Node :: DFA_Transition ( int ts )
523 {
524     NDFA_Node_Set_Iterator  iter ;
```

```
525
526     if ( transitions [ ts ] . empty () )
527     {
528         return 0L ;
529     } else {
530         iter = transitions [ ts ] . begin () ;
531         return (* iter) ;
532     }
533 }
534
535 /****** NDFA_Node private methods ******/
536
537 /** Delete_Item_Before
538  *
539  * This convenience procedure finds the NDFA node before the current item
540  * in "set" (i.e. "iter" - 1). This NDFA node is removed from the set and
541  * deleted.
542  */
543 void NDFA_Node :: Delete_Item_Before ( NDFA_Node_Set * set ,
544                                        NDFA_Node_Set_Iterator iter )
545 {
546     NDFA_Node *             item ;
547
548     iter -- ;
549     item = (* iter) ;
550
551     delete item ;
552
553     set -> erase ( iter ) ;
554 }
555
556 /** Check_For_Determinism
557  *
558  * Examines the current node (no recursion) and its children
559  * and sets the deterministic flags appropriately.
560  *
561  * Node is deterministic if there are 0 or 1 children
562  * for each type of transition and those children are deterministic too.
563  */
564 void NDFA_Node :: Check_For_Determinism ()
565 {
566     is_deterministic = true ;
567     for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
568     {
569         if ( transitions [ ts ] . size () > 1 )
570         {
571             // not deterministic - 2 or more children
572             is_deterministic = false ;
573         } else if ( ! transitions [ ts ] . empty () )
574         {
575             if ( ! DFA_Transition ( ts ) -> is_deterministic )
576             {
577                 // child is not deterministic
578                 is_deterministic = false ;
579             }
580         }
581     }
582 }
583
584 /** Delete_Children
585  *
586  * Used for the conversion of a node into a rejecting node, or for
587  * freeing the data structures.
588  */
589 void NDFA_Node :: Delete_Children ()
590 {
591     NDFA_Node_Set_Iterator  remove_iter ;
592
593     /* Unfortunately a child node may be in both ndfa_root -> transitions[1]
594     and ndfa_root -> transitions[0]. So items from transitions[0] are moved
595     to transitions[1] before deletion */
596     remove_iter = transitions [ 0 ] . begin () ;
597     while ( remove_iter != transitions [ 0 ] . end () )
598     {
```

```
599         NDFA_Node * n = (* remove_iter) ;
600
601         transitions [ 1 ] . insert ( n ) ;
602         remove_iter ++ ;
603     }
604
605     remove_iter = transitions [ 1 ] . begin () ;
606     while ( remove_iter != transitions [ 1 ] . end () )
607     {
608         NDFA_Node * n = (* remove_iter) ;
609
610         delete n ;
611
612         remove_iter ++ ;
613     }
614     transitions [ 0 ] . clear () ;
615     transitions [ 1 ] . clear () ;
616 }
617
618 /** Transform_NDFA_To_DFA
619  *
620  * Does the work for Make_Deterministic(). The NDFA (this) is
621  * converted to a DFA (dfa_root) by a depth bounded depth first
622  * tree operation.
623  *
624  * This will not work correctly on all compressed trees - an assertion
625  * will fail if the tree has been compressed. See the comment for
626  * Compress().
627  */
628 void NDFA_Node :: Transform_NDFA_To_DFA ( NDFA_Node * dfa_root ,
629                                           int depth_remaining )
630 {
631     NDFA_Node_Set_Iterator  iter ;
632     NDFA_Node *             item ;
633     int                     ts ;
634
635     if ( depth_remaining < 0 )
636     {
637         /* Implement the depth bound */
638         return ;
639     }
640
641     if (( dfa_root -> is_reject_state )
642     || ( dfa_root -> is_accept_state ))
643     {
644         /* The DFA node is an accept/reject state. The children of
645          * the node, if any, are not important. */
646         return ;
647     }
648
649     /* If any NDFA nodes are in each transition set, check to
650      * see if any of them are accept or reject states... */
651     for ( ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
652     {
653         if ( ! this -> transitions [ ts ] . empty () )
654         {
655             NDFA_Node * dfa_child = dfa_root -> DFA_Transition ( ts ) ;
656
657             if ( dfa_child == 0L )
658             {
659                 dfa_child = new NDFA_Node
660                                 ( dfa_root -> test_bit_number + 1 ) ;
661                 dfa_root -> transitions [ ts ] . insert ( dfa_child ) ;
662             }
663
664             iter = this -> transitions [ ts ] . begin () ;
665             while ( iter != this -> transitions [ ts ] . end () )
666             {
667                 item = (* iter) ;
668                 if ( item -> is_reject_state )
669                 {
670                     /* The NDFA node is a reject state. Go no further: the
671                     DFA node must also be a reject state. */
672                     dfa_child -> Make_Reject_State () ;
```

```
673                 } else if ( item -> is_accept_state )
674                 {
675                     /* The NDFA node is an accept state. Ensure that, if
676                      * the DFA node is already known to be an accept state,
677                      * that it is the SAME accept state. Otherwise,
678                      * it is non-deterministic and there is nothing
679                      * we can do to change that. */
680
681                     dfa_child -> Make_Accept_State ( item -> accept_state ) ;
682                 } else {
683                     /* Regular transition node.
684                      * Ensure that all children reached by transition ts in
685                      * the NDFA are testing the same bit number. */
686                      assert ( dfa_child -> test_bit_number ==
687                                       item -> test_bit_number ) ;
688                 }
689                 iter ++ ;
690             }
691         }
692     }
693
694     if (( dfa_root -> is_reject_state )
695     || ( dfa_root -> is_accept_state ))
696     {
697         /* The DFA node has become, as a result of the nodes just added
698          * to it, an accept/reject state. The children of
699          * the node, if any, are not important. */
700         return ;
701     }
702
703     /* Transform the child subtrees */
704     for ( ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
705     {
706         NDFA_Node * dfa_child = dfa_root -> DFA_Transition ( ts ) ;
707
708         if ( dfa_child != 0L )
709         {
710             iter = this -> transitions [ ts ] . begin () ;
711             while ( iter != this -> transitions [ ts ] . end () )
712             {
713                 item = (* iter) ;
714
715                 item -> Transform_NDFA_To_DFA ( dfa_child ,
716                                       depth_remaining - 1 ) ;
717                 iter ++ ;
718             }
719         }
720     }
721     dfa_root -> Check_For_Determinism () ;
722 }
723
724 /** Replace_With
725  *
726  * The important fields of n are copied to this. Then, n is deleted.
727  * The aim is to allow the node <this> to be replaced by n, so that
728  * n is moved up the tree.
729  */
730 void NDFA_Node :: Replace_With ( NDFA_Node * n )
731 {
732     this -> accept_state = n -> accept_state ;
733     this -> is_accept_state = n -> is_accept_state ;
734     this -> is_reject_state = n -> is_reject_state ;
735     this -> test_bit_number = n -> test_bit_number ;
736
737     for ( int ts = 0 ; ts < TRANSITION_TYPES ; ts ++ )
738     {
739         this -> transitions [ ts ] = n -> transitions [ ts ] ;
740         n -> transitions [ ts ] . clear () ;
741     }
742
743     delete n ;
744 }
745
```

G.9. Source code of ndfa_node.h

```
 1
 2
 3 #ifndef NDFA_NODE_H
 4 #define NDFA_NODE_H
 5
 6 #include <exception>
 7 #include <set>
 8
 9 #include "definitions.h"
10 #include "ndfa_accept_state.h"
11 #include "exceptions.h"
12
13
14 class NDFA_Node ;
15
16 typedef set<NDFA_Node *> NDFA_Node_Set ;
17 typedef NDFA_Node_Set::iterator NDFA_Node_Set_Iterator ;
18
19
20
21 class NDFA_Node
22 {
23 public:
24     NDFA_Node ( int test_bit_number ) ;
25     virtual ~NDFA_Node () ;
26
27
28     bool Is_Deterministic ()
29             { return is_deterministic ; } ;
30
31     void Merge_In ( NDFA_Node * to_be_merged , bool cpr ) ;
32
33     void Compress () ;
34
35     bool Delete_Dead_Branches () ;
36
37     void Print_NDFA_DAG ( bool with_pointers , FILE * fd ,
38                           const char * old_tab_str = "" ) ;
39
40     int Get_Size () ;
41
42     NDFA_Node * Add_Transition ( bool transition_on_zero ,
43                                  bool transition_on_one ) ;
44     void Make_Reject_State () ;
45     void Make_Accept_State ( Accept_State * accept_info ) ;
46
47     void Make_Deterministic () ;
48
49     NDFA_Node * DFA_Transition ( int ts ) ;
50
51     int Get_Test_Bit_Number ()
52                 { return test_bit_number ; } ;
53
54     Accept_State * Get_Accept_State () ;
55
56     void Visit ()
57                 { visits ++ ; } ;
58     int Get_Number_Of_Visits ()
59                 { return visits ; } ;
60
61
62
63
64 private:
65     void Delete_Item_Before ( NDFA_Node_Set * set ,
66                               NDFA_Node_Set_Iterator iter ) ;
67     void Check_For_Determinism () ;
68     void Delete_Children () ;
69     void Transform_NDFA_To_DFA ( NDFA_Node * dfa_root , int depth_remaining ) ;
70     void Replace_With ( NDFA_Node * n ) ;
71
72
```

```
73
74      Accept_State *      accept_state ;
75      bool                is_accept_state ;
76      bool                is_reject_state ;
77      bool                is_deterministic ;
78      NDFA_Node_Set       transitions [ TRANSITION_TYPES ] ;
79      int                 test_bit_number ;
80      int                 visits ;
81
82
83  } ;
84
85
86
87
88  #endif
89
```

G.10. Source code of opcode_map_reader.cc

```
 1  #include <stdio.h>
 2
 3  #include <sys/types.h>
 4  #include <regex.h>
 5  #include <assert.h>
 6  #include <stdlib.h>
 7  #include <string.h>
 8  #include <ctype.h>
 9
10  #include "opcode_map_reader.h"
11  #include "utils.h"
12  #include "definitions.h"
13  #include "ndfa_dag.h"
14  #include "ndfa_accept_state.h"
15  #include "state.h"
16  #include "state_machine.h"
17
18  /****** Opcode_Map_Reader public methods ******/
19
20  /** Read_Opcode_Map
21   *
22   * Read in an opcode map file, updating the ndfa structure.
23   * Accept state information is obtained from New_Accept_State().
24   */
25  int Opcode_Map_Reader :: Read_Opcode_Map ( const char * filename )
26  {
27      const char *    read_regex_str =
28              "^(.... ... ... ......)  (.{8})(.{7})(.{16})(.{8})(.*)$" ;
29      const int       NUMFIELDS = 6 ; /* number of regular expression fields */
30
31      char            str [ MAX_LINE_LEN + 1 ] ;
32      FILE *          fd = fopen ( filename , "rt" ) ;
33      int             rc ;
34      regex_t         read_regex ;
35      int             opcodes_read = 0 ;
36
37      if ( fd == 0 )
38      {
39          return 0 ;
40      }
41
42      MESSAGE ( "Reading opcode map from %s\n" , filename ) ;
43
44      rc = regcomp ( & read_regex , read_regex_str , REG_EXTENDED ) ;
45      assert ( rc == 0 ) ;
46
47      /* read in a line from the map file */
48      while ( fgets ( str , MAX_LINE_LEN , fd ) != NULL )
49      {
50          Remove_Trailing_Newlines ( str ) ;
51
52          regmatch_t      matches [ NUMFIELDS + 1 ] ;
53
```

```
54          if (( str [ 0 ] == '#' )     /* comment */
55          || ( ! String_Contains_Non_Whitespace ( str ) ))    /* blank line */
56          {
57              continue ;
58          }
59
60          if ( regexec ( & read_regex , str , NUMFIELDS + 1 ,
61                             matches , 0 ) != 0 )
62          {
63              /* line not recognised */
64              throw new Unrecognised_Opcode_Map_Entry ( str ) ;
65          }
66
67          char *      micro_sub_name =
68                                  Get_Regex_Match ( str , & matches [ 4 ] ) ;
69
70          if ( ! String_Contains_Non_Whitespace ( micro_sub_name ) )
71          {
72              MESSAGE3 ( "Skipped '%s' - no micro sub name.\n" , str ) ;
73              delete [] micro_sub_name ;
74              continue ;
75          }
76
77          Remove_Whitespace_From_Ends ( micro_sub_name ) ;
78
79          char *      bit_pattern =
80                                  Get_Regex_Match ( str , & matches [ 1 ] ) ;
81          char *      opcode_name =
82                                  Get_Regex_Match ( str , & matches [ 2 ] ) ;
83          char *      dea =       Get_Regex_Match ( str , & matches [ 3 ] ) ;
84          char *      optimisation_info =
85                                  Get_Regex_Match ( str , & matches [ 5 ] ) ;
86          char *      comment =   Get_Regex_Match ( str , & matches [ 6 ] ) ;
87          char        cmp_bit_pattern [ BITS_PER_OPCODE + 1 ] ;
88          Accept_State *
89                      accept_state ;
90          int         i , j ;
91
92          Remove_Whitespace_From_Ends ( opcode_name ) ;
93          Remove_Whitespace_From_Ends ( comment ) ;
94
95          MESSAGE2 ( "Adding %s %s\n" , opcode_name , comment ) ;
96
97          /* Remove all spaces from the bit pattern */
98          for ( i = 0 , j = 0 ; i < (int) strlen ( bit_pattern ) ; i ++ )
99          {
100             if ( ! isspace ( bit_pattern [ i ] ) )
101             {
102                 assert ( j < BITS_PER_OPCODE ) ;
103                 cmp_bit_pattern [ j ] = bit_pattern [ i ] ;
104                 j ++ ;
105             }
106         }
107         assert ( j == BITS_PER_OPCODE ) ;
108         cmp_bit_pattern [ j ] = '\0' ;
109
110         /* Get an accept state for this line. */
111         accept_state = New_Accept_State ( cmp_bit_pattern , opcode_name ,
112                                           dea , micro_sub_name ,
113                                           optimisation_info , comment ) ;
114         assert ( accept_state != 0L ) ;
115
116         /* Create a temporary NDFA that accepts any bit sequence
117          * and has an accept state with information from "accept_state" */
118         NDFA_DAG   temporary_ndfa ( accept_state ) ;
119
120         /* Begin parsing the bit pattern to determine which
121          * patterns are NOT acceptable for this opcode. This
122          * bit is highly specific to the format of the map file. */
123         for ( i = 0 ; i < BITS_PER_OPCODE ; i ++ )
124         {
125             switch ( cmp_bit_pattern [ i ] )
126             {
127                 case '0' :
```

```
128                 temporary_ndfa . Reject_Pattern ( i , "1" ) ;
129                 break ;
130             case '1' :
131                 temporary_ndfa . Reject_Pattern ( i , "0" ) ;
132                 break ;
133             case 'R' :
134             case 'r' :
135             case 'Z' :
136             case 'D' :
137             case 'e' :
138             case 'a' :
139             case 'I' :
140             case 'X' :
141             case 'C' :
142                 /* These always match. They may be 1 or 0 */
143                 break ;
144             default :
145                 if ( strncmp ( & cmp_bit_pattern [ i ] ,
146                                             "SS" , 2 ) == 0 )
147                 {
148                     /* a size, that can be anything but 11 */
149                     temporary_ndfa . Reject_Pattern ( i , "11" ) ;
150                     i ++ ;
151                 } else if ( strncmp ( & cmp_bit_pattern [ i ] ,
152                                             "ss" , 2 ) == 0 )
153                 {
154                     /* another sort of size, that can be
155                      * anything but 00 */
156                     temporary_ndfa . Reject_Pattern ( i , "00" ) ;
157                     i ++ ;
158                 } else if ( strncmp ( & cmp_bit_pattern [ i ] ,
159                                             "TT" , 2 ) == 0 )
160                 {
161                     /* a bit operation type. Doesn't
162                     match if it is 00 */
163                     temporary_ndfa . Reject_Pattern ( i , "00" ) ;
164                     i ++ ;
165                 } else if ( strncmp ( & cmp_bit_pattern [ i ] ,
166                                             "cccc" , 4 ) == 0 )
167                 {
168                     /* Condition code.. may be anything apart
169                     from 1. */
170                     temporary_ndfa . Reject_Pattern ( i , "0001" ) ;
171                     i += 3 ;
172                 } else if ( strncmp ( & cmp_bit_pattern [ i ] ,
173                                             "EEEEEE" , 6 ) == 0 )
174                 {
175                     /* An effective address field. */
176                     Add_Reject_EAs ( & temporary_ndfa , i ,
177                                     dea , false ) ;
178                     i += 5 ;
179                 } else if ( strncmp ( & cmp_bit_pattern [ i ] ,
180                                             "MMMMMM" , 6 ) == 0 )
181                 {
182                     /* Another sort of effective address field. */
183                     Add_Reject_EAs ( & temporary_ndfa , i ,
184                                     dea , true ) ;
185                     i += 5 ;
186                 } else {
187                     /* character not recognised */
188                     assert ( 0 ) ;
189                 }
190                 break ;
191         }
192     }
193
194     MESSAGE3 ( "  (NDFA size for '%s' = %d)\n" ,
195                     opcode_name , temporary_ndfa . Get_Size () ) ;
196
197     /* Turn the NDFA that accepts this opcode into a DFA.
198      * The procedure assumes that any state that is both a
199      * reject state and an accept state is just a reject state.
200      * In this way, the resulting DFA accepts exactly the opcode
201      * and rejects everything else. */
```

```
202         temporary_ndfa . Make_Deterministic () ;
203
204         MESSAGE3 ( "  (DFA size for '%s' = %d)\n" ,
205                         opcode_name , temporary_ndfa . Get_Size () ) ;
206
207         /* Can't compress the DFA here. It will compress to one accept
208          * state! */
209
210         /* And merge it into the main opcode
211          * We do not set any of the states in _this_ NDFA as REJECT,
212          * (last parameter is false) because the reject states
213          * of one opcode may well be accept states of another. */
214         ndfa . Merge_In ( & temporary_ndfa , false ) ;
215
216         opcodes_read ++ ;
217
218         MESSAGE3 ( "  (%d opcodes: master NDFA size is now %d)\n" ,
219                         opcodes_read , ndfa . Get_Size () ) ;
220
221         /* Free up memory used to store regex fields */
222         delete [] bit_pattern ;
223         delete [] opcode_name ;
224         delete [] dea ;
225         delete [] micro_sub_name ;
226         delete [] comment ;
227     }
228     regfree ( & read_regex ) ;
229
230     MESSAGE ( "Making decoding DFA..\n" ) ;
231
232     ndfa . Make_Deterministic () ;
233
234     MESSAGE ( "Decoding DFA has %d nodes.\n" , ndfa . Get_Size () ) ;
235
236     return opcodes_read ;
237 }
238
239 /****** Opcode_Map_Reader private methods ******/
240
241 /** Add_Reject_EAs
242  *
243  * A helper function for adding a series of rejection states to
244  * an NDFA, to represent an effective address field.
245  */
246 void Opcode_Map_Reader :: Add_Reject_EAs ( NDFA_DAG * n , int offset ,
247                 const char * dea , bool move_destination_ea )
248 {
249     /* parse the list of disallowed address modes */
250     if ( index ( dea , 'i' ) != NULL )
251     {
252         /* Immediates are not allowed. */
253         n -> Reject_Pattern ( offset ,
254             move_destination_ea ? "100111" : "111100" ) ;
255     }
256     if ( index ( dea , 'p' ) != NULL )
257     {
258         /* PC-relative is not allowed.
259          not indirect with displacement: */
260         n -> Reject_Pattern ( offset ,
261             move_destination_ea ? "010111" : "111010" ) ;
262         /* nor memory indirect with index */
263         n -> Reject_Pattern ( offset ,
264             move_destination_ea ? "011111" : "111011" ) ;
265     }
266     if ( index ( dea , 'a' ) != NULL )
267     {
268         /* Address Register Direct: not allowed */
269         n -> Reject_Pattern (
270             move_destination_ea ? ( offset + 3 ) : offset , "001" ) ;
271     }
272     if ( index ( dea , 'd' ) != NULL )
273     {
274         /* Data Register Direct: not allowed */
275         n -> Reject_Pattern (
```

```
276             move_destination_ea ? ( offset + 3 ) : offset , "000" ) ;
277     }
278     if (( index ( dea , 'r' ) != NULL )
279     || ( index ( dea , '+' ) != NULL ))
280     {
281         /* Register modifying modes not allowed...
282         no postinc */
283         n -> Reject_Pattern (
284             move_destination_ea ? ( offset + 3 ) : offset , "011" ) ;
285     }
286     if (( index ( dea , 'r' ) != NULL )
287     || ( index ( dea , '-' ) != NULL ))
288     {
289         /* Register modifying modes not allowed...
290         no predec */
291         n -> Reject_Pattern (
292             move_destination_ea ? ( offset + 3 ) : offset , "100" ) ;
293     }
294
295     /* These three modes are reserved by Motorola
296     and are thus never valid. */
297     n -> Reject_Pattern ( offset ,
298         move_destination_ea ? "101111" : "111101" ) ;
299     n -> Reject_Pattern ( offset ,
300         move_destination_ea ? "110111" : "111110" ) ;
301     n -> Reject_Pattern ( offset ,
302         move_destination_ea ? "111111" : "111111" ) ;
303 }
304
```

G.11. Source code of opcode_map_reader.h

```
 1
 2 #ifndef OPCODE_MAP_READER_H
 3 #define OPCODE_MAP_READER_H
 4
 5 #include <set>
 6
 7 #include "ndfa_dag.h"
 8
 9 class Opcode_Map_Reader
10 {
11 public:
12     Opcode_Map_Reader () { } ;
13     virtual ~Opcode_Map_Reader () { } ;
14
15     int Read_Opcode_Map ( const char * filename ) ;
16
17     virtual Accept_State * New_Accept_State (
18                 const char * cmp_bit_pattern ,
19                 const char * opcode_name ,
20                 const char * dea ,
21                 const char * micro_sub_name ,
22                 const char * optimisation_info ,
23                 const char * comment ) { return 0L ; } ;
24
25 protected:
26     NDFA_DAG       ndfa ;
27
28 private:
29
30     void Add_Reject_EAs ( NDFA_DAG * ndfa , int offset ,
31                 const char * dea , bool move_destination_ea ) ;
32
33 } ;
34
35 #endif
36
```

G.12. Source code of optimisation.cc

```
 1
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <ctype.h>
 5
 6 #include "optimisation.h"
 7 #include "exceptions.h"
 8 #include "utils.h"
 9
10
11 /****** Optimisation_Record static data ******/
12
13 const Optimisation_Type_Data Optimisation_Record :: opt_data [] = {
14         { EA , "ea_mode" , "EM" } ,
15         { EAREG , "ea_reg" , "em" } ,
16         { ALUOP , "alu_internal_op" , "+-|&^c" } } ;
17
18 /****** Optimisation_Record public methods ******/
19
20 /** Optimisation_Record constructor
21  *
22  * A new optimisation record is created for the named optimisation type.
23  */
24 Optimisation_Record :: Optimisation_Record ( const char * n )
25 {
26     /* We don't know what subtype the optimisation will take.. */
27     subtype = '\0' ;
28
29     /* Search the opt_data table for the optimisation type */
30     for ( int i = 0 ; i < NUMBER_OF_OPTIMISATIONS ; i ++ )
31     {
32         if ( strcasecmp ( n , opt_data [ i ] . name ) == 0 )
33         {
34             type = opt_data [ i ] . type ;
35             return ;
36         }
37     }
38     throw new Unrecognised_Optimisation_Exception ( n ) ;
39 }
40
41 /** Optimisation_Record constructor
42  *
43  * A new optimisation record is created for the named optimisation type.
44  * Here, the name is specified as a code character. These are the same
45  * characters that are used in the optimisation field of the opcode map.
46  */
47 Optimisation_Record :: Optimisation_Record ( char c )
48 {
49     /* Make a note of the optimisation's subtype */
50     subtype = c ;
51
52     /* Search the opt_data table for the optimisation type */
53     for ( int i = 0 ; i < NUMBER_OF_OPTIMISATIONS ; i ++ )
54     {
55         if ( index ( opt_data [ i ] . codes , c ) != 0L )
56         {
57             type = opt_data [ i ] . type ;
58             return ;
59         }
60     }
61     char name [ 2 ] = { c , '\0' } ;
62
63     throw new Unrecognised_Optimisation_Exception ( name ) ;
64 }
65
66 /** Print_Info
67  *
68  * Print the type name and subtype character of this optimisation record.
69  */
70 void Optimisation_Record :: Print_Info ()
71 {
72     MESSAGE3 ( "%s(%c) " , opt_data [ type ] . name , subtype ) ;
```

```
 73 }
 74
 75 /****** Optimisation_NDFA_Accept_State public methods ******/
 76
 77 /** Optimisation_NDFA_Accept_State destructor
 78  *
 79  * The list of optimisation records is deleted.
 80  */
 81 Optimisation_NDFA_Accept_State :: ~Optimisation_NDFA_Accept_State ()
 82 {
 83     Optimisation_Record_Set_Iter        iter ;
 84
 85     iter = opts . begin () ;
 86     while ( iter != opts . end () )
 87     {
 88         Optimisation_Record * item = (* iter) ;
 89
 90         delete item ;
 91
 92         iter ++ ;
 93     }
 94 }
 95
 96 /****** Optimisation_Manager public methods ******/
 97
 98 /** Optimisation_Manager constructor
 99  *
100  * The various optimisation classes are created.
101  */
102 Optimisation_Manager :: Optimisation_Manager ()
103                         : Opcode_Map_Reader ()
104 {
105     optimisation_types [ EA ] = new EA_Optimisation () ;
106     optimisation_types [ EAREG ] = new EA_Reg_Optimisation () ;
107     optimisation_types [ ALUOP ] = new ALU_Optimisation () ;
108 }
109
110 /** Optimisation_Manager destructor
111  *
112  * The various optimisation classes are deleted.
113  */
114 Optimisation_Manager :: ~Optimisation_Manager ()
115 {
116     for ( int i = 0 ; i < NUMBER_OF_OPTIMISATIONS ; i ++ )
117     {
118         delete optimisation_types [ i ] ;
119     }
120 }
121
122 /** Notify
123  *
124  * All the optimisation classes that are available for this opcode are
125  * notified that it will appear in the program. They use this information
126  * to work out what optimisations can be applied.
127  */
128 void Optimisation_Manager :: Notify ( unsigned opcode )
129 {
130     MESSAGE2 ( "Adding optimisation info for opcode 0x%04x..\n" ,
131                         opcode ) ;
132
133     Optimisation_NDFA_Accept_State *    accept_state =
134         (Optimisation_NDFA_Accept_State *) ndfa . Get_Accept_State ( opcode ) ;
135
136     if ( accept_state == 0L )
137     {
138         throw new UnavailableOpcodeException () ;
139     }
140
141     Optimisation_Record_Set             o_list = accept_state ->
142                                             Get_Optimisation_Record_Set () ;
143
144     Optimisation_Record_Set_Iter        iter ;
145
146     iter = o_list . begin () ;
```

```
147     while ( iter != o_list . end () )
148     {
149         Optimisation_Record * item = (* iter) ;
150 
151         optimisation_types [ item -> Get_Optimisation_Type () ] ->
152                             Notify ( opcode , item -> Get_Subtype () ) ;
153         iter ++ ;
154     }
155 }
156 
157 /** New_Accept_State
158  *
159  * Create a new accept state. This method is called from inherited method
160  * Read_Opcode_Map as data is read from the file.
161  */
162 Accept_State * Optimisation_Manager :: New_Accept_State (
163                 const char * cmp_bit_pattern ,
164                 const char * opcode_name ,
165                 const char * dea ,
166                 const char * micro_sub_name ,
167                 const char * optimisation_info ,
168                 const char * comment )
169 {
170     Optimisation_NDFA_Accept_State * accept_state ;
171 
172     /* Create the new accept state */
173     accept_state = new Optimisation_NDFA_Accept_State ;
174 
175     /* Now, what optimisations should be present for this opcode?
176      * Check the optimisation_info field */
177 
178     MESSAGE3 ( "Adding optimisations for %s: " , opcode_name ) ;
179     for ( int i = 0 ; i < (int) strlen ( optimisation_info ) ; i ++ )
180     {
181         if ( ! isspace ( optimisation_info [ i ] ) )
182         {
183             /* Create a new optimisation and add it to the accept state. */
184             Optimisation_Record * new_opt =
185                     new Optimisation_Record ( optimisation_info [ i ] ) ;
186 
187             new_opt -> Print_Info () ;
188 
189             accept_state -> Add_Optimisation ( new_opt ) ;
190         }
191     }
192 
193     MESSAGE3 ( "\n" ) ;
194 
195     return accept_state ;
196 }
197 
198 /** Generate_VHDL
199  *
200  * The given optimisation type is asked for the VHDL that implements the
201  * optimisation.
202  */
203 void Optimisation_Manager :: Generate_VHDL (
204                         FILE * output , Optimisation_Record o_type )
205 {
206     optimisation_types [ o_type . Get_Optimisation_Type () ] ->
207                                             Generate_VHDL ( output ) ;
208 }
209 
```

G.13. Source code of optimisation.h

```
1 #ifndef OPTIMISATION_H
2 #define OPTIMISATION_H
3 
4 #include <stdio.h>
5 #include <set>
6 
7 #include "basic_optimisation.h"
```

```
 8 #include "ea_optimisation.h"
 9 #include "ea_reg_optimisation.h"
10 #include "alu_optimisation.h"
11 #include "opcode_map_reader.h"
12
13 enum Optimisation_Type { EA = 0 , EAREG ,
14                 ALUOP , NUMBER_OF_OPTIMISATIONS } ;
15
16 struct Optimisation_Type_Data
17 {
18     Optimisation_Type   type ;
19     const char          name [ 16 ] ;
20     const char          codes [ 16 ] ;
21 } ;
22
23 class Optimisation_Record
24 {
25 public:
26     Optimisation_Record ( const char * name ) ;
27     Optimisation_Record ( char c ) ;
28
29     Optimisation_Type Get_Optimisation_Type () const
30                 { return type ; } ;
31     char Get_Subtype () const
32                 { return subtype ; } ;
33
34     void Print_Info () ;
35
36 private:
37     Optimisation_Type   type ;
38     char                subtype ;
39
40     static const Optimisation_Type_Data opt_data [ NUMBER_OF_OPTIMISATIONS ] ;
41 } ;
42
43 struct Optimisation_Record_Compare
44 {
45     bool operator () ( const Optimisation_Record * r1 ,
46                        const Optimisation_Record * r2 ) const
47     {
48         if ( r1 -> Get_Optimisation_Type () ==
49                 r2 -> Get_Optimisation_Type () )
50         {
51             return ( r1 -> Get_Subtype () < r2 -> Get_Subtype () ) ;
52         } else {
53             return ( r1 -> Get_Optimisation_Type () ) <
54                     ( r2 -> Get_Optimisation_Type () ) ;
55         }
56     }
57 } ;
58
59 typedef set<Optimisation_Record *, Optimisation_Record_Compare>
60                                             Optimisation_Record_Set ;
61 typedef Optimisation_Record_Set::iterator Optimisation_Record_Set_Iter ;
62
63 class Optimisation_NDFA_Accept_State : public Accept_State
64 {
65 public:
66     virtual ~Optimisation_NDFA_Accept_State () ;
67
68     Optimisation_Record_Set & Get_Optimisation_Record_Set ()
69             { return opts ; } ;
70     void Add_Optimisation ( Optimisation_Record * r )
71             { opts . insert ( r ) ; } ;
72
73 private:
74     Optimisation_Record_Set opts ;
75 } ;
76
77 class Optimisation_Manager : public Opcode_Map_Reader
78 {
79 public:
80     Optimisation_Manager () ;
81     virtual ~Optimisation_Manager () ;
```

```
 82
 83     void Notify ( unsigned opcode ) ;
 84
 85     void Generate_VHDL ( FILE * output , Optimisation_Record o_type ) ;
 86
 87     virtual Accept_State * New_Accept_State (
 88                 const char * cmp_bit_pattern ,
 89                 const char * opcode_name ,
 90                 const char * dea ,
 91                 const char * micro_sub_name ,
 92                 const char * optimisation_info ,
 93                 const char * comment ) ;
 94
 95 private:
 96     Basic_Optimisation * optimisation_types [ NUMBER_OF_OPTIMISATIONS ] ;
 97 } ;
 98
 99 #endif
100
```

G.14. Source code of programram.cc

```
 1
 2 #include <stdio.h>
 3
 4 #include "programram.hh"
 5
 6 using namespace vm68k ;
 7
 8 ProgramRAM :: ProgramRAM ( uint32_type startAddr , uint32_type memorySize ,
 9                     const char * filename , bool debug )
10                 throw ( memory_exception , file_reading_exception )
11                     : RAM ( startAddr , memorySize , debug )
12 {
13     FILE * ihex = fopen ( filename , "rt" ) ;
14
15     if ( ihex == NULL )
16     {
17         throw file_reading_exception () ;
18     }
19
20     startPC = 0 ;
21     lowestAddrUsed = highestAddrUsed = 0 ;
22
23     while ( ! feof ( ihex ) )
24     {
25         const uint32_type
26                         bytesLimit = 32 ;
27         int             bytes [ bytesLimit ] ;
28         uint32_type     numberOfBytes , lineAddr , recordType ;
29         char            buffer [ 128 ] ;
30
31         fgets ( buffer , 127 , ihex ) ;
32
33         if (( buffer [ 0 ] == ':' )
34         && ( sscanf ( buffer , ":%02x%04x%02x" ,
35                 & numberOfBytes , & lineAddr , & recordType ) == 3 )
36         && ( numberOfBytes <= bytesLimit ))
37         {
38             /* scan data bytes */
39             for ( uint32_type i = 0 ; i < numberOfBytes ; i ++ )
40             {
41                 uint32_type startChar = 9 + ( (int) i * 2 ) ;
42
43                 if ( ! (( startChar < strlen ( buffer ))
44                     && ( sscanf ( & buffer [ startChar ] , "%02x" ,
45                             & bytes [ i ] ) == 1 )) )
46                 {
47                     throw file_reading_exception () ;
48                 }
49             }
50             if (( recordType == 0 ) // This is a data record
51             && ( numberOfBytes > 0 ))
```

```
52            {
53                uint32_type endOfLineAddr = ( lineAddr + numberOfBytes - 1 ) ;
54
55                if ( lineAddr < getLowestAddr ())
56                {
57                    throw ( address_error ( lineAddr , memory :: WRITE ) ) ;
58                } else if ( endOfLineAddr > getHighestAddr ())
59                {
60                    throw ( address_error ( lineAddr + numberOfBytes - 1 , memory :: WRITE ) ) ;
61                }
62
63                for ( uint32_type i = 0 ; i < numberOfBytes ; i ++ )
64                {
65                    put_8 ( (uint32_type) lineAddr + i , bytes [ i ] ,
66                            (function_code) memory :: WRITE ) ;
67                }
68
69                if ( endOfLineAddr > highestAddrUsed )
70                {
71                    highestAddrUsed = endOfLineAddr ;
72                } else if ( lineAddr < lowestAddrUsed )
73                {
74                    lowestAddrUsed = lineAddr ;
75                }
76            } else if (( recordType == 3 )
77            && ( numberOfBytes >= 4 ))
78            {
79                startPC = bytes [ 3 ] |
80                        ( bytes [ 2 ] << 8 ) |
81                        ( bytes [ 1 ] << 16 ) |
82                        ( bytes [ 0 ] << 24 ) ;
83            }
84        }
85    }
86    fclose ( ihex ) ;
87 }
88
89 ProgramRAM :: ~ProgramRAM ()
90 {
91 }
92
```

G.15. Source code of programram.hh

```
 1 #ifndef PROGRAM_RAM_HH
 2 #define PROGRAM_RAM_HH
 3
 4 #include "ram.hh"
 5
 6 #include <vm68k/types.h>
 7 #include <vm68k/memory.h>
 8 #include <vm68k/processor.h>
 9
10 using namespace vm68k ;
11
12 class ProgramRAM : public RAM
13 {
14 public:
15     struct file_reading_exception : exception
16     {
17     } ;
18
19     ProgramRAM ( uint32_type startAddr , uint32_type memorySize ,
20                     const char * filename , bool debug = false )
21             throw ( memory_exception , file_reading_exception ) ;
22     virtual ~ProgramRAM () ;
23
24     uint32_type getStartPC ()
25         { return startPC ; } ;
26     uint32_type getLowestAddrUsedByProgram ()
27         { return lowestAddrUsed ; } ;
28     uint32_type getHighestAddrUsedByProgram ()
29         { return highestAddrUsed ; } ;
```

```
30
31 private:
32     uint32_type startPC ;
33     uint32_type lowestAddrUsed , highestAddrUsed ;
34 } ;
35
36 #endif
37
```

G.16. Source code of `state.cc`

```
 1
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <stdlib.h>
 5 #include <assert.h>
 6 #include <limits.h>
 7
 8 #include <map>
 9 #include <list>
10 #include <iostream>
11
12 #include "utils.h"
13 #include "state.h"
14
15 /****** State public methods ******/
16
17 /** State destructor
18  *
19  * Frees the state and associated command list.
20  */
21 State :: ~State ()
22 {
23     /* Free the command list */
24     Command_List_Iterator   iter ;
25     Command *               item ;
26
27     iter = commands . begin () ;
28     while ( iter != commands . end () )
29     {
30         item = (* iter) ;
31         delete item ;
32         iter ++ ;
33     }
34
35     if ( state_name != 0L )
36     {
37         delete [] state_name ;
38     }
39 }
40
41 /** Add_Command
42  *
43  * Add an extra command to the end of the list of VHDL
44  * commands for the given state. */
45 void State :: Add_Command ( Command * cmd )
46 {
47     commands . push_back ( cmd ) ;
48 }
49
50 /** Depends_On
51  *
52  * Scan the command list for CALLs/JUMPs and return their labels as a set */
53 Label_Set State :: Depends_On ()
54 {
55     Label_Set               depends_on_set ;
56     Command_List_Iterator   iter ;
57     Command *               item ;
58
59     iter = commands . begin () ;
60     while ( iter != commands . end () )
61     {
62         item = (* iter) ;
```

```
63
64          if (( item -> Get_Type () == Command :: JUMP )
65          || ( item -> Get_Type () == Command :: CALL ))
66          {
67              depends_on_set . insert ( item -> Get_Data () ) ;
68          }
69          iter ++ ;
70      }
71      return depends_on_set ;
72  }
73
74  /** Generate_VHDL
75   *
76   * VHDL is generated for the state.
77   */
78  void State :: Generate_VHDL ( FILE * output , State_Map * definitions )
79  {
80      Command_List_Iterator   command_list_pos ;
81
82      fprintf ( output , "\nwhen " ) ;
83      Print_Binary ( output , abs_state_number , width_of_state_number ) ;
84      fprintf ( output , " => -- 0x%04x (%d) %s\n" ,
85                  abs_state_number , abs_state_number ,
86                  state_name == 0L ? "none" : state_name ) ;
87
88      command_list_pos = commands . begin () ;
89      while ( command_list_pos != commands . end () )
90      {
91          Command *           current_command = (* command_list_pos) ;
92          State *             target_mc ;
93
94          switch ( current_command -> Get_Type ())
95          {
96              case Command :: VHDL :
97                  fputs ( current_command -> Get_Data () , output ) ;
98                  fputs ( "\n" , output ) ;
99                  break ;
100             case Command :: IDECODE :
101                 /* Instruction decode. This is much like a CALL,
102                  * except the address to be called comes from the
103                  * instruction decoding logic. */
104                 fprintf ( output , "\n\t-- Instruction decode.\n"
105                         "\tcall_state <= decoded_state ;\n"
106                         "\tcall_requested <= '1' ;\n"
107                         "\treturn_requested <= '0' ;\n" ) ;
108                 break ;
109             case Command :: JUMP :
110             case Command :: CALL :
111                 /* The target machine should be part of this machine
112                  * already if the dependencies have been properly
113                  * satisfied. If it is not, we stop now. */
114                 assert ( definitions -> count
115                                 ( current_command -> Get_Data () ) != 0 ) ;
116
117                 target_mc = (* definitions) [ current_command -> Get_Data () ] ;
118
119                 assert ( target_mc != 0 ) ;
120
121                 fprintf ( output ,
122                     "\n\t-- %s %s:\n"
123                     "\tcall_state <= " ,
124                     ( current_command -> Get_Type () == Command :: CALL ) ?
125                                             "CALL" : "JUMP" ,
126                     current_command -> Get_Data () ) ;
127
128                 Print_Binary ( output ,
129                     target_mc -> abs_state_number , /* target state */
130                     width_of_state_number ) ;
131
132                 fprintf ( output , " ;\n"
133                         "\tcall_requested <= '1' ;\n"
134                         "\treturn_requested <= '%d' ;\n" ,
135                         ( current_command -> Get_Type () == Command :: CALL ) ?
136                                             0 : 1 ) ;
```

```
137                 break ;
138             case Command :: RETURN :
139                 fprintf ( output ,
140                         "\treturn_requested <= '1' ;\n" ) ;
141                 break ;
142             default :
143                 assert ( 0 ) ;
144         }
145         command_list_pos ++ ;
146     }
147 }
148
149 /** Generate_Link_To_State_VHDL
150  *
151  * Instruction decoder VHDL is generated to cause a jump to the state.
152  */
153 void State :: Generate_Link_To_State_VHDL ( FILE * output ,
154                                             const char * indent )
155 {
156     fprintf ( output , "%sdecoded_state <= " , indent ) ;
157
158     Print_Binary ( output , abs_state_number ,
159                 width_of_state_number ) ;
160
161     fprintf ( output , " ; -- (%s)\n" ,
162                 state_name == 0L ? "none" : state_name ) ;
163 }
164
165 /** Set_State_Name
166  *
167  * Changes the name of the state. If the name is already known,
168  * the space is freed before it is changed.
169  */
170 void State :: Set_State_Name ( const char * name )
171 {
172     if ( state_name != 0L )
173     {
174         delete [] state_name ;
175     }
176     state_name = Copy_String ( name ) ;
177 }
178
179 /****** Command public methods ******/
180
181 /** Command constructor
182  *
183  * Creates a new Command class.
184  */
185 Command :: Command ( Command_Type t , const char * cmd_data )
186 {
187     type = t ;
188     if ( cmd_data == 0L )
189     {
190         data = 0L ;
191     } else {
192         data = Copy_String ( cmd_data ) ;
193     }
194 }
195
196 /** Command destructor
197  *
198  * Deletes a Command class.
199  */
200 Command :: ~Command ()
201 {
202     /* Free the data associated with the command, if any */
203     if ( data != 0L )
204     {
205         delete [] data ;
206     }
207 }
208
```

G.17. Source code of state.h

```
 1 #ifndef STATE_H
 2 #define STATE_H
 3
 4 #include <stdio.h>
 5
 6 #include <set>
 7 #include <map>
 8 #include <list>
 9 #include <string>
10
11 #include "optimisation.h"
12
13 /* The label set is used for passing around lists of dependencies on
14  * particular state labels */
15 typedef set<const char *> Label_Set ;
16 typedef Label_Set :: iterator Label_Set_Iterator ;
17
18 class State ;
19
20 /* The Command class maintains details of a single command. There are
21  * 1 or more commands in a state. */
22
23 class Command
24 {
25 public:
26     enum Command_Type { UNKNOWN_TYPE , CALL , RETURN ,
27                 JUMP , VHDL , IDECODE } ;
28
29     Command ( Command_Type t , const char * cmd_data = 0L ) ;
30     virtual ~Command () ;
31
32     Command_Type Get_Type ()
33             { return type ; } ;
34     const char * Get_Data ()
35             { return data ; } ;
36
37 private:
38     Command_Type    type ;
39     const char *    data ;
40 } ;
41
42 typedef list<Command *> Command_List ;
43 typedef Command_List :: iterator Command_List_Iterator ;
44
45 /* The state map maintains a link between a label name and the state
46  * it refers to. */
47
48 struct Strings_Less
49 {
50     bool operator () ( string s1 , string s2 ) const
51     {
52         return strcmp ( s1 . c_str () , s2 . c_str () ) < 0 ;
53     }
54 } ;
55
56 typedef map<string , State * , Strings_Less> State_Map ;
57 typedef State_Map :: iterator State_Map_Iterator ;
58
59 /* The state command maintains details of a single state in the state
60  * machine. There are 1 or more states in a state machine. */
61
62 class State
63 {
64 public:
65     State () { state_name = 0L ; } ;
66     virtual ~State () ;
67
68     void Add_Command ( Command * cmd ) ;
69     Label_Set Depends_On () ;
70     void Generate_VHDL ( FILE * output , State_Map * definitions ) ;
71     void Generate_Link_To_State_VHDL ( FILE * output , const char * indent ) ;
72
```

```
73      void Set_Abs_State_Number ( int n )
74              { abs_state_number = n ; } ;
75      void Set_Width_Of_State_Number ( int n )
76              { width_of_state_number = n ; } ;
77      void Set_State_Name ( const char * name ) ;
78      bool Is_Empty ()
79              { return commands . empty () ; } ;
80
81  private:
82      Command_List    commands ;
83      int             abs_state_number ;
84      int             width_of_state_number ;
85      const char *    state_name ;
86  } ;
87
88  typedef list<State *> State_List ;
89  typedef State_List :: iterator State_List_Iterator ;
90
91  #endif
92
```

G.18. Source code of state_machine.cc

```
 1
 2  #include <stdio.h>
 3  #include <string.h>
 4  #include <stdlib.h>
 5  #include <assert.h>
 6  #include <limits.h>
 7
 8  #include <map>
 9  #include <list>
10  #include <iostream>
11
12  #include "utils.h"
13  #include "state_machine.h"
14
15  /****** State_Machine public methods ******/
16
17  /** State_Machine constructor
18   *
19   * Create a new state machine from the given source file.
20   */
21  State_Machine :: State_Machine ( const char * source_file )
22  {
23      const int       NUMFIELDS = 2 ;
24      regmatch_t      matches [ NUMFIELDS + 1 ] ;
25      State *         current_state = new State () ;
26      FILE *          input = fopen ( source_file , "rt" ) ;
27      char            str [ MAX_LINE_LEN + 1 ] ;
28      int             line_no = 0 ;
29      int             rc ;
30      regex_t         statement_with_parameter_regex ;
31      regex_t         statement_without_parameter_regex ;
32
33      /* Compile regular expressions */
34      rc = regcomp ( & statement_with_parameter_regex ,
35          "^[\t ]*(CALL|JUMP|LABEL) +([A-Za-z0-9_]+)[ \t]*$" ,
36              REG_EXTENDED ) ;
37      assert ( rc == 0 ) ;
38
39      rc = regcomp ( & statement_without_parameter_regex ,
40          "^[\t ]*(CLOCK|RETURN|IDECODE)[ \t]*$" , REG_EXTENDED ) ;
41      assert ( rc == 0 ) ;
42
43      /* Begin building the state list */
44      states . push_back ( current_state ) ;
45
46      if ( input == 0L )
47      {
48          throw new File_Access_Exception ( source_file ) ;
49      }
50
```

```
51      /* Read in lines from the machine definition. */
52      while ( fgets ( str , MAX_LINE_LEN , input ) != NULL )
53      {
54          line_no ++ ;
55
56          Remove_Trailing_Newlines ( str ) ;
57
58          if ( regexec ( & statement_with_parameter_regex ,
59                              str , NUMFIELDS + 1 , matches , 0 ) == 0 )
60          {
61              /* This is one of the state machine statements */
62              char *  statement_name = Get_Regex_Match ( str , & matches [ 1 ] ) ;
63              char *  parameter_name = Get_Regex_Match ( str , & matches [ 2 ] ) ;
64
65              if ( strcmp ( statement_name , "JUMP" ) == 0 )
66              {
67                  current_state -> Add_Command (
68                          new Command ( Command :: JUMP , parameter_name ) ) ;
69              } else if ( strcmp ( statement_name , "CALL" ) == 0 )
70              {
71                  current_state -> Add_Command (
72                          new Command ( Command :: CALL , parameter_name ) ) ;
73              } else if ( strcmp ( statement_name , "LABEL" ) == 0 )
74              {
75                  /* This state is being assigned a label. */
76
77                  char *  label_name = parameter_name ;
78
79                  /* We add the label to the label definitions table */
80                  if ( definitions . count ( string ( label_name ) ) > 0 )
81                  {
82                      throw new Duplicate_Label_Exception ( label_name ) ;
83                  }
84
85                  definitions [ string ( label_name ) ] = current_state ;
86
87                  /* For convenience, we also store the state name in the
88                   * state itself */
89                  current_state -> Set_State_Name ( label_name ) ;
90              } else {
91                  assert ( 0 ) ;
92              }
93              delete [] statement_name ;
94              delete [] parameter_name ;
95          } else if ( regexec ( & statement_without_parameter_regex , str , 2 ,
96                                              matches , 0 ) == 0 )
97          {
98              char *  statement_name = Get_Regex_Match ( str , & matches [ 1 ] ) ;
99
100             if ( strcmp ( statement_name , "RETURN" ) == 0 )
101             {
102                 current_state -> Add_Command (
103                         new Command ( Command :: RETURN ) ) ;
104             } else if ( strcmp ( statement_name , "IDECODE" ) == 0 )
105             {
106                 current_state -> Add_Command (
107                         new Command ( Command :: IDECODE ) ) ;
108             } else if ( strcmp ( statement_name , "CLOCK" ) == 0 )
109             {
110                 /* That's the end of state marker. Create a new state. */
111                 current_state = new State () ;
112                 /* and add it to the list of states */
113                 states . push_back ( current_state ) ;
114             } else {
115                 assert ( 0 ) ;
116             }
117             delete [] statement_name ;
118         } else if ( String_Contains_Non_Whitespace ( str ) )
119         {
120             /* This is a VHDL statement or a comment or something
121             like that. We don't care precisely what it is. */
122             current_state -> Add_Command (
123                     new Command ( Command :: VHDL , str ) ) ;
124         }
```

```
125     }
126     fclose ( input ) ;
127 
128     /* A common error is to omit the CLOCK from the last state in the
129     machine. We can check for that here: does the last state have
130     anything in it? */
131 
132     if ( ! current_state -> Is_Empty () )
133     {
134         throw new Missing_Clock_Exception ( source_file ) ;
135     }
136 
137     /* Remove last state, since it has no CLOCK */
138     states . pop_back () ;
139     delete current_state ;
140 
141     /* We also check that the machine has some states in it. An empty
142     machine is a mistake! */
143     if ( states . empty () )
144     {
145         throw new Empty_Machine_Exception ( source_file ) ;
146     }
147 
148     /* Store the name of the state machine */
149     name = Copy_String ( source_file ) ;
150 
151     /* Free the regular expressions */
152     regfree ( & statement_without_parameter_regex ) ;
153     regfree ( & statement_with_parameter_regex ) ;
154 
155     is_finalised = false ;
156 }
157 
158 /** State_Machine empty constructor
159  *
160  * A new empty state machine is created.
161  */
162 State_Machine :: State_Machine ()
163 {
164     is_finalised = false ;
165     name = Copy_String ( "unnamed" ) ;
166 }
167 
168 /** State_Machine destructor
169  *
170  * Destroys the state machine and _all_ related memory.
171  */
172 State_Machine :: ~State_Machine ()
173 {
174     /* Free the state list */
175     State_List_Iterator iter ;
176     State *             item ;
177 
178     iter = states . begin () ;
179     while ( iter != states . end () )
180     {
181         item = (* iter) ;
182         delete item ;
183         iter ++ ;
184     }
185 
186     delete [] name ;
187 }
188 
189 /** Incorporate_Sub_Machine
190  *
191  * The given state machine is incorporated into the current state machine.
192  * This is intended to be done in order to bring in other machines required
193  * to satisfy dependencies. The sub machine is left empty.
194  */
195 void State_Machine :: Incorporate_Sub_Machine ( State_Machine * sub_machine )
196 {
197     assert ( ! is_finalised ) ;
198 
```

```
199     /* Move the contents of the sub machine into this one:
200      * states and definitions */
201     MESSAGE2 ( "Incorporating sub machine %s.\n" , Get_Name () ) ;
202
203     /* Copy the states */
204     State_List_Iterator state_list_iter ;
205
206     state_list_iter = sub_machine -> states . begin () ;
207     while ( state_list_iter != sub_machine -> states . end () )
208     {
209         State * item = (* state_list_iter) ;
210
211         states . push_back ( item ) ;
212
213         state_list_iter ++ ;
214     }
215
216     /* Copy the definitions */
217     State_Map_Iterator  state_map_iter ;
218
219     state_map_iter = sub_machine -> definitions . begin () ;
220     while ( state_map_iter != sub_machine -> definitions . end () )
221     {
222         const char *    label = (* state_map_iter) . first . c_str () ;
223         State *         state = (* state_map_iter) . second ;
224
225         if ( definitions . count ( string ( label ) ) > 0 )
226         {
227             throw new Duplicate_Label_Exception ( label ) ;
228         }
229         definitions [ string ( label ) ] = state ;
230
231         state_map_iter ++ ;
232     }
233
234     /* Now delete the contents of the sub machine.. This is essential
235      * to prevent the memory that makes them up being deleted when
236      * the sub machine is deleted. */
237     sub_machine -> states . clear () ;
238     sub_machine -> definitions . clear () ;
239 }
240
241 /** Compile_Machine
242  *
243  * Generates the VHDL for the state machine, sending it to the given
244  * output device. This can only be done once all the dependencies
245  * for the machine, as listed by Depends_On(), have been satisfied.
246  * Assertions will fail if dependencies haven't been satisfied.
247  */
248 void State_Machine :: Compile_Machine ( FILE * output )
249 {
250     int                             width_of_state_number ;
251     State_List_Iterator             state_list_pos ;
252
253
254     if ( ! is_finalised )
255     {
256         /* Add absolute state numbers to all states */
257         Calculate_Abs_State_Numbers () ;
258     }
259
260     width_of_state_number = Get_Width_Of_State_Number () ;
261
262     MESSAGE2 ( "Number of states: %d\n"
263                 "Width of state number (bits): %d\n" ,
264                 states . size () , width_of_state_number ) ;
265
266     /* Must be in a process sensitive to 'state' and 'clock' */
267     fprintf ( output ,
268         "-- Start automatically generated state machine logic.\n"
269         "\tcall_requested <= '0' ;\n"
270         "\treturn_requested <= '0' ;\n"
271         "\tcall_state <= ( others => '0' ) ;\n\n"
272         "\tcase state is\n" ) ;
```

```
273
274     state_list_pos = states . begin () ;
275     while ( state_list_pos != states . end () )
276     {
277         State * current_state = (* state_list_pos) ;
278
279         current_state -> Generate_VHDL ( output , & definitions ) ;
280
281         state_list_pos ++ ;
282     }
283
284     fprintf ( output ,
285         "when others => null ;\n"
286         "\tend case ;\n\n"
287         "-- End automatically generated state machine logic.\n" ) ;
288 }
289
290 /** Depends_On
291  *
292  * Return a list of states that this machine depends upon.
293  * Essentially, this is the set of all states that are JUMPed
294  * or CALLed, minus the set of locations within this machine. */
295 Label_Set State_Machine :: Depends_On ()
296 {
297     Label_Set           depends_on_set ;
298     State_List_Iterator state_iter ;
299     State *             state ;
300
301     state_iter = states . begin () ;
302     while ( state_iter != states . end () )
303     {
304         state = (* state_iter) ;
305
306         Label_Set           state_depends_on_set = state -> Depends_On () ;
307         Label_Set_Iterator  label_iter ;
308
309         label_iter = state_depends_on_set . begin () ;
310         while ( label_iter != state_depends_on_set . end () )
311         {
312             const char *    label = (* label_iter) ;
313
314             depends_on_set . insert ( label ) ;
315
316             label_iter ++ ;
317         }
318
319         state_iter ++ ;
320     }
321
322     /* Now we have the set of all states reached by a JUMP or CALL
323      * from this machine. Remove all the states provided by this machine. */
324
325     Label_Set           provides_set = Provides () ;
326     Label_Set_Iterator  label_iter ;
327
328     label_iter = provides_set . begin () ;
329     while ( label_iter != provides_set . end () )
330     {
331         const char *    label = (* label_iter) ;
332
333         if ( depends_on_set . count ( label ) != 0 )
334         {
335             /* This label is depended upon by this machine, but it is
336              * also provided by this machine. */
337             label_iter ++ ;
338
339             depends_on_set . erase ( label ) ;
340         } else {
341             label_iter ++ ;
342         }
343     }
344
345     return depends_on_set ;
346 }
```

```
347
348 /** Provides
349  *
350  * Return the set of labels provided by this machine. This
351  * is simply the set of keys of the "definitions" map */
352 Label_Set State_Machine :: Provides ()
353 {
354     Label_Set           provides_set ;
355     State_Map_Iterator  iter ;
356
357     iter = definitions . begin () ;
358     while ( iter != definitions . end () )
359     {
360         const char * label = (* iter) . first . c_str () ;
361
362         provides_set . insert ( label ) ;
363
364         iter ++ ;
365     }
366
367     return provides_set ;
368 }
369
370 /** Finalise_SM
371  *
372  * The state machine is finalised - no more changes can be made to it.
373  */
374 void State_Machine :: Finalise_SM ()
375 {
376     assert ( ! is_finalised ) ;
377     Calculate_Abs_State_Numbers () ;
378     is_finalised = true ;
379 }
380
381 /** Get_State_For_Name
382  *
383  * Convert a state name into a State object, if possible.
384  */
385 State * State_Machine :: Get_State_For_Name ( const char * sub_name )
386 {
387     if ( definitions . count ( string ( sub_name ) ) > 0 )
388     {
389         return definitions [ string ( sub_name ) ] ;
390     } else {
391         return 0L ;
392     }
393 }
394
395 /****** State_Machine private methods ******/
396
397 /** Calculate_Abs_State_Numbers
398  *
399  * Work out the absolute state numbers of all the states in the database.
400  */
401 void State_Machine :: Calculate_Abs_State_Numbers ()
402 {
403     MESSAGE2 ( "Calculating absolute state numbers.\n" ) ;
404
405     State_List_Iterator pos ;
406     int                 abs_state_number = 0 ;
407
408     pos = states . begin () ;
409     while ( pos != states . end () )
410     {
411         State * state = (* pos) ;
412
413         state -> Set_Abs_State_Number ( abs_state_number ) ;
414
415         abs_state_number ++ ;
416
417         pos ++ ;
418     }
419
420     /* Program the width of each state number. */
```

```
421     int                 width_of_state_number = Get_Width_Of_State_Number () ;
422
423     pos = states . begin () ;
424     while ( pos != states . end () )
425     {
426         State * state = (* pos) ;
427
428         state -> Set_Width_Of_State_Number ( width_of_state_number ) ;
429
430         abs_state_number ++ ;
431
432         pos ++ ;
433     }
434 }
435
```

G.19. Source code of state_machine.h

```
 1 #ifndef STATE_MACHINE_H
 2 #define STATE_MACHINE_H
 3
 4 #include <stdio.h>
 5 #include <regex.h>
 6
 7 #include <set>
 8 #include <map>
 9 #include <list>
10
11 #include "exceptions.h"
12 #include "state.h"
13 #include "utils.h"
14 #include "optimisation.h"
15
16 /* The state machine class. This is able to read in a state machine
17  * description, produce the VHDL it represents, and calculate the
18  * dependencies of the machine. Other state machines can be merged in. */
19 class State_Machine
20 {
21 public:
22     State_Machine ( const char * source_file ) ;
23     State_Machine () ;
24     virtual ~State_Machine () ;
25
26     void Compile_Machine ( FILE * output ) ;
27
28     void Incorporate_Sub_Machine ( State_Machine * sub_machine ) ;
29
30     Label_Set Depends_On () ;
31     Label_Set Provides () ;
32
33     void Finalise_SM () ;
34
35     bool Is_Finalised ()
36             { return is_finalised ; } ;
37
38     State * Get_State_For_Name ( const char * sub_name ) ;
39
40     int Get_Width_Of_State_Number ()
41             { return Get_Number_Of_Bits_Needed_For
42                             ( states . size () - 1 ) ; } ;
43
44     const char * Get_Name ()
45             { return name ; } ;
46 private:
47     void Calculate_Abs_State_Numbers () ;
48
49     State_List      states ;
50     State_Map       definitions ;
51     bool            is_finalised ;
52     const char *    name ;
53 } ;
54
55 #endif
```

G.20. Source code of state_machine_loader.cc

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <assert.h>
 5 #include <limits.h>
 6
 7 #include <sys/types.h>
 8 #include <dirent.h>
 9 #include <fnmatch.h>
10
11 #include <map>
12 #include <list>
13 #include <iostream>
14
15 #include "utils.h"
16 #include "state_machine_loader.h"
17
18 /****** State_Machine public methods ******/
19
20 /** State_Machine_Loader destructor
21  *
22  * All state machines are deleted.
23  */
24 State_Machine_Loader :: ~State_Machine_Loader ()
25 {
26     State_Machine_Set_Iterator iter = machines . begin () ;
27
28     while ( iter != machines . end () )
29     {
30         State_Machine * item = (* iter) ;
31
32         delete item ;
33
34         iter ++ ;
35     }
36 }
37
38 /** Add_State_Machine
39  *
40  * A state machine is loaded from the given file, and added to the loader's
41  * internal database.
42  */
43 void State_Machine_Loader :: Add_State_Machine ( const char * filename )
44 {
45     assert ( ! is_finalised ) ;
46
47     MESSAGE2 ( "Loading state machine file %s\n" , filename ) ;
48
49     State_Machine * sm = new State_Machine ( filename ) ;
50
51     /* The new state machine is added to the list of machines */
52     machines . insert ( sm ) ;
53
54     /* The provides list of the machine is examined. */
55     Label_Set           provides_list = sm -> Provides () ;
56     Label_Set_Iterator  iter = provides_list . begin () ;
57
58     while ( iter != provides_list . end () )
59     {
60         const char * state_name = (* iter) ;
61
62         if ( provides_map . count ( string ( state_name ) ) > 0 )
63         {
64             throw new Duplicate_Label_Exception ( state_name ) ;
65         }
66         provides_map [ string ( state_name ) ] = sm ;
67
68         iter ++ ;
69     }
```

```
 70 }
 71
 72 /** Add_State_Machine_Directory
 73  *
 74  * All state machine (.sm) files are loaded from the given directory.
 75  */
 76 void State_Machine_Loader :: Add_State_Machine_Directory ( const char * dir )
 77 {
 78     assert ( ! is_finalised ) ;
 79
 80     DIR *           dird = opendir ( dir ) ;
 81     struct dirent * entry ;
 82
 83     if ( dird == 0L )
 84     {
 85         throw new Dir_Access_Exception ( dir ) ;
 86     }
 87     MESSAGE ( "Scanning state machine directory %s\n" , dir ) ;
 88
 89     while (( entry = readdir ( dird ) ) != NULL )
 90     {
 91         char * filename = new char [ strlen ( entry -> d_name )
 92                                          + strlen ( dir ) + 2 ] ;
 93         strcpy ( filename , dir ) ;
 94         strcat ( filename , "/" ) ;
 95         strcat ( filename , entry -> d_name ) ;
 96
 97         if ( fnmatch ( STATE_MACHINE_GLOB , filename , 0 ) == 0 )
 98         {
 99             Add_State_Machine ( filename ) ;
100         }
101
102         delete [] filename ;
103     }
104     closedir ( dird ) ;
105 }
106
107 /** Require_Microsub
108  *
109  * Indicate to the database that a particular microsubroutine is required.
110  */
111 void State_Machine_Loader :: Require_Microsub ( const char * sub_name )
112 {
113     assert ( ! is_finalised ) ;
114
115     /* The machine containing this subroutine will certainly be required. */
116     MESSAGE2 ( "Adding microsubroutine %s.." , sub_name ) ;
117     State_Machine * sm = Get_State_Machine_For_Name ( sub_name ) ;
118
119     /* Is sm already on the required list of machines? */
120     if ( required_machines . count ( sm ) == 0 )
121     {
122         /* No... add it now. */
123         required_machines . insert ( sm ) ;
124
125         MESSAGE2 ( "added\n" ) ;
126         /* Also add everything it depends on */
127         Require_Microsubs ( sm -> Depends_On () ) ;
128     } else {
129         MESSAGE2 ( "already present\n" ) ;
130     }
131 }
132
133 /** Require_Microsubs
134  *
135  * Indicate to the database that all the members of a set of
136  * microsubroutines are required.
137  */
138 void State_Machine_Loader :: Require_Microsubs ( Label_Set list )
139 {
140     assert ( ! is_finalised ) ;
141
142     Label_Set_Iterator  iter = list . begin () ;
143
```

```
144     while ( iter != list . end () )
145     {
146         const char * state_name = (* iter) ;
147         Require_Microsub ( state_name ) ;
148 
149         iter ++ ;
150     }
151 }
152 
153 /** Build_Master_Machine
154  *
155  * Creates a new state machine containing all the machines required
156  * to execute the program.
157  *
158  * Once called, no more changes can be made to the state machine loader
159  * database (they could invalidate the machine).
160  */
161 State_Machine * State_Machine_Loader :: Build_Master_Machine (
162                                         const char * root_sub_name )
163 {
164     if ( is_finalised )
165     {
166         /* machine already built */
167         return master_machine ;
168     }
169 
170     MESSAGE ( "Building master state machine..\n" ) ;
171 
172     /* The master machine is based on the machine containing the
173      * root subroutine named here.
174      * Ensure that all dependencies are met, and that the
175      * named subroutine exists. */
176     Require_Microsub ( root_sub_name ) ;
177 
178     master_machine = Get_State_Machine_For_Name ( root_sub_name ) ;
179 
180     /* Add all other required machines to the master machine. This
181      * will effectively concatenate them on the end. */
182     State_Machine_Set      required_machines_copy = required_machines ;
183     State_Machine_Set_Iterator iter ;
184 
185     required_machines_copy . erase ( master_machine ) ;
186 
187     iter = required_machines_copy . begin () ;
188     while ( iter != required_machines_copy . end () )
189     {
190         State_Machine * item = (* iter) ;
191 
192         master_machine -> Incorporate_Sub_Machine ( item ) ;
193 
194         iter ++ ;
195     }
196 
197     is_finalised = true ; /* No more changes can be made to the database */
198 
199     master_machine -> Finalise_SM () ;
200 
201     /* Debugging information */
202     MESSAGE2 ( "State machine report:\n" ) ;
203     iter = machines . begin () ;
204     while ( iter != machines . end () )
205     {
206         State_Machine * item = (* iter) ;
207 
208         if ( required_machines . count ( item ) > 0 )
209         {
210             MESSAGE2 ( "includes %s\n" , item -> Get_Name () ) ;
211         } else {
212             MESSAGE2 ( "   omits %s\n" , item -> Get_Name () ) ;
213         }
214         iter ++ ;
215     }
216     MESSAGE2 ( "Total: %d machines of %d included.\n" ,
217             required_machines . size () , machines . size () ) ;
```

```
218
219     return master_machine ;
220 }
221
222 /** Get_State_Machine_For_Name
223  *
224  * Finds the state machine which provides the given state/microsubroutine.
225  */
226 State_Machine * State_Machine_Loader ::
227                     Get_State_Machine_For_Name ( const char * sub_name )
228 {
229     if ( provides_map . count ( string ( sub_name ) ) == 0 )
230     {
231         /* This subroutine doesn't seem to exist! */
232         throw new Unavailable_Microsub_Exception ( sub_name ) ;
233     }
234     return provides_map [ string ( sub_name ) ] ;
235 }
236
```

G.21. Source code of state_machine_loader.h

```
 1
 2 #ifndef STATE_MACHINE_LOADER_H
 3 #define STATE_MACHINE_LOADER_H
 4
 5 #include <set>
 6
 7 #include "state.h"
 8 #include "state_machine.h"
 9
10 typedef set<State_Machine *> State_Machine_Set ;
11 typedef State_Machine_Set :: iterator State_Machine_Set_Iterator ;
12
13 typedef map<string , State_Machine * , Strings_Less> Provides_Map ;
14 typedef Provides_Map :: iterator Provides_Map_Iterator ;
15
16 class State_Machine_Loader
17 {
18 public:
19     State_Machine_Loader () { is_finalised = false ; } ;
20     virtual ~State_Machine_Loader () ;
21
22     void Require_Microsub ( const char * sub_name ) ;
23     void Require_Microsubs ( Label_Set list ) ;
24     void Add_State_Machine ( const char * filename ) ;
25     void Add_State_Machine_Directory ( const char * dir ) ;
26
27     State_Machine * Build_Master_Machine ( const char * root_sub_name ) ;
28
29     State_Machine * Get_State_Machine_For_Name ( const char * sub_name ) ;
30
31 private:
32     Provides_Map        provides_map ;
33     State_Machine_Set   machines ;
34     State_Machine_Set   required_machines ;
35     bool                is_finalised ;
36     State_Machine *     master_machine ;
37 } ;
38
39 #endif
40
```

G.22. Source code of utils.cc

```
 1
 2 #include <string.h>
 3 #include <regex.h>
 4 #include <assert.h>
 5 #include <ctype.h>
 6
```

```
 7 #include "utils.h"
 8
 9 int g_verbose_setting ;
10
11 /* This procedure deletes the newline character and everything
12  * after it from the supplied string. If no newline character
13  * is present, the string is unchanged. */
14 void Remove_Trailing_Newlines ( char * str )
15 {
16     char * newline_index = index ( str , '\n' ) ;
17
18     if ( newline_index != NULL )
19     {
20         newline_index [ 0 ] = '\0' ;
21     }
22 }
23
24 /* Extracts a particular regular expression match substring from
25  * match_str. An assertion fails if the regex function didn't find any
26  * substring to match the specified pattern.. i.e. if the match
27  * parameter is not valid.
28  *
29  * Note: a deep copy of the substring is made that will need to be freed
30  * later. */
31 char * Get_Regex_Match ( const char * match_str , regmatch_t * match )
32 {
33     int     length_of_string = ( match -> rm_eo - match -> rm_so ) ;
34     char *  substring ;
35
36     assert ( match -> rm_so >= 0 ) ;
37     assert ( length_of_string >= 0 ) ;
38
39     substring = new char [ sizeof ( char ) * ( length_of_string + 1 ) ] ;
40
41     memcpy ( substring , & match_str [ match -> rm_so ] ,
42                          length_of_string ) ;
43     substring [ length_of_string ] = '\0' ;
44
45     return substring ;
46 }
47
48 /* Compute the minimum number of bits required to represent
49  * the integer parameter. */
50 int Get_Number_Of_Bits_Needed_For ( int parameter )
51 {
52     if ( parameter <= 2 )
53     {
54         return 1 ;
55     }
56
57     int width = 0 ;
58
59     while ( parameter > 0 )
60     {
61         parameter = parameter >> 1 ;
62         width ++ ;
63     }
64
65     return width ;
66 }
67
68 /** Print_Binary
69  *
70  * Prints a number in binary. The given binary number is sent to the
71  * specified stream, surrounded by double quotes.
72  */
73 void Print_Binary ( FILE * fd , int number , int width )
74 {
75     const char * s = Binary_To_String ( number , width ) ;
76
77     fprintf ( fd , "\"%s\"" , s ) ;
78
79     delete [] s ;
80 }
```

```
 81
 82 /** Binary_To_String
 83  *
 84  * Creates a new string, of length 'width+1', and puts the
 85  * given binary number in the string.
 86  */
 87 const char * Binary_To_String ( int number , int width )
 88 {
 89     char * str      = new char [ width + 1 ] ;
 90
 91     str [ width ] = '\0' ;
 92     for ( int i = width - 1 ; i >= 0 ; i -- )
 93     {
 94         str [ ( width - 1 ) - i ] = (( number >> i ) & 1 ) ? '1' : '0' ;
 95     }
 96     return str ;
 97 }
 98
 99 /** Copy_String
100  *
101  * The same as 'strdup', but uses 'new', not 'malloc'.
102  */
103 const char * Copy_String ( const char * original )
104 {
105     char * str      = new char [ strlen ( original ) + 1 ] ;
106
107     strcpy ( str , original ) ;
108     return str ;
109 }
110
111 /* Returns true if the string contains any non-whitespace characters */
112 bool String_Contains_Non_Whitespace ( const char * str )
113 {
114     int     i ;
115
116     for ( i = strlen ( str ) - 1 ; i >= 0 ; i -- )
117     {
118         if ( ! isspace ( str [ i ] ) )
119         {
120             return true ;
121         }
122     }
123     return false ;
124 }
125
126 /* Removes whitespace characters from the beginning and end of the string */
127 void Remove_Whitespace_From_Ends ( char * str )
128 {
129     unsigned        i = strlen ( str ) - 1 ;
130
131     /* First remove whitespace from the end of the string. */
132     for ( ; i >= 0 ; i -- )
133     {
134         if ( isspace ( str [ i ] ) )
135         {
136             str [ i ] = '\0' ;
137         } else {
138             break ;
139         }
140     }
141     /* Now, remove whitespace from the beginning */
142     for ( i = 0 ; i < strlen ( str ) ; i ++ )
143     {
144         if ( ! isspace ( str [ i ] ) )
145         {
146             break ;
147         }
148     }
149     /* str[i] is the first non-whitespace character.
150      * Move str[i] to str[0], copying the null terminator as well. */
151     if (( i > 0 )
152     && ( i < strlen ( str )) )
153     {
154         memmove ( & str [ 0 ] , & str [ i ] , strlen ( & str [ i ] ) + 1 ) ;
```

```
155     }
156 }
157
```

G.23. Source code of utils.h

```
 1
 2 #ifndef UTILS_H
 3 #define UTILS_H
 4
 5 #include <stdio.h>
 6 #include <regex.h>
 7
 8 extern int g_verbose_setting ;
 9
10 #define VERBOSE_OFF         0
11 #define VERBOSE_LOW         1
12 #define VERBOSE_MEDIUM      2
13 #define VERBOSE_HIGH        3
14 #define VERBOSE_VERY_HIGH   4
15
16 #define MESSAGE     if ( g_verbose_setting >= VERBOSE_LOW ) printf
17 #define MESSAGE2    if ( g_verbose_setting >= VERBOSE_MEDIUM ) printf
18 #define MESSAGE3    if ( g_verbose_setting >= VERBOSE_HIGH ) printf
19 #define MESSAGE4    if ( g_verbose_setting >= VERBOSE_VERY_HIGH ) printf
20
21
22 /* This procedure deletes the newline character and everything
23  * after it from the supplied string. If no newline character
24  * is present, the string is unchanged. */
25 void Remove_Trailing_Newlines ( char * str ) ;
26
27 /* Extracts a particular regular expression match substring from
28  * match_str. An assertion fails if the regex function didn't find any
29  * substring to match the specified pattern.. i.e. if the match
30  * parameter is not valid.
31  *
32  * Note: a deep copy of the substring is made that will need to be freed
33  * later. */
34 char * Get_Regex_Match ( const char * match_str , regmatch_t * match ) ;
35
36 /* Compute the minimum number of bits required to represent
37  * the integer parameter. */
38 int Get_Number_Of_Bits_Needed_For ( int parameter ) ;
39
40 void Print_Binary ( FILE * fd , int number , int width ) ;
41
42 bool String_Contains_Non_Whitespace ( const char * str ) ;
43
44 void Remove_Whitespace_From_Ends ( char * str ) ;
45
46 const char * Binary_To_String ( int number , int width ) ;
47
48 const char * Copy_String ( const char * original ) ;
49
50 #endif
51
```

H. The Opcode Database

H.1. Source code of opcode_map

```
 1
 2 #BIT PATTERN         OP      DEA    SUBROUTINE      OPTIMS  COMMENTS
 3 0000 000 0SS EEEEEE  ORI     aip    alu_i_family    |Ee     ea <- [ea] | immediate
 4 0000 000 000 111100  ORICCR                         |       CCR <- CCR | immediate [byte]
 5 0000 000 001 111100  ORISR                          |       SR <- SR | immediate [word]
 6 0000 0SS 011 EEEEEE  CHK2                           Ee      68020 only
 7 0000 RRR 100 EEEEEE  BTST    a                      Ee      Dynamic Bit on D(RRR) in [ea]
 8 0000 RRR 1TT EEEEEE  Btt     aip                    Ee      Dynamic Bit on D(RRR) in [ea]
```

```
 9 0000 RRR 10Z 001rrr  MOVEP                                  D(RRR) <- [A(rrr) + d]
10 0000 RRR 11Z 001rrr  MOVEPR                                 A(rrr) + d  <- D(RRR)
11 0000 001 0SS EEEEEE  ANDI    aip    alu_i_family    &Ee     ea <- [ea] & immediate
12 0000 001 000 111100  ANDICCR                        &       CCR <- CCR & immediate [byte]
13 0000 001 001 111100  ANDISR                         &       SR <- SR & immediate [word]
14 0000 010 0SS EEEEEE  SUBI    aip    alu_i_family    -Ee     ea <- [ea] - immediate
15 0000 011 0SS EEEEEE  ADDI    aip    alu_i_family    +Ee     ea <- [ea] + immediate
16 0000 011 011 00DRRR  RTM                                    68020 only
17 0000 011 011 EEEEEE  CALLM   adri                   Ee      68020 only
18 0000 11Z 011 EEEEEE  CAS     adip                   Ee      68020 only
19 0000 100 000 EEEEEE  BTSTS   ai                     Ee      Static bit on <imm byte> bit in [ea]
20 0000 100 0TT EEEEEE  BttS    aip                    Ee      Static bit on <imm byte> bit in [ea]
21 0000 101 0SS EEEEEE  EORI    aip    alu_i_family    ^Ee     ea <- [ea] ^ immediate
22 0000 101 000 111100  EORICCR                        ^       CCR <- CCR ^ immediate [byte]
23 0000 101 001 111100  EORISR                         ^       SR <- SR ^ immediate [word]
24 0000 110 0SS EEEEEE  CMPI    ai     alu_i_cmp       cEe     [ea] - immediate CMP
25 0000 111 0SS EEEEEE  MOVES   adip                   Ee      68010 only
26
27 00ss MMM MMM eeeeee  MOVE    aip    move_family     EMem    ea <- [sea]
28 0010 RRR 001 EEEEEE  MOVEAL          move_family     EMem    A(RRR) <- [ea]
29 0011 RRR 001 EEEEEE  MOVEAW          move_family     EMem    A(RRR) <- [ea] [word mode w/ sign ex]
30
31 0100 000 0SS EEEEEE  NEGX    aip                    Ee      ea <- 0 - [ea] - X
32 0100 000 011 EEEEEE  MOVEsr  aip                    Ee      ea <- SR [word mode]
33 0100 RRR 1a0 EEEEEE  CHK     a                      Ee      TRAP if D(RRR) < 0 or D(RRR) > [ea]
34 0100 RRR 111 EEEEEE  LEA     adri   lea             Ee      A(RRR) <- ea
35 0100 001 0SS EEEEEE  CLR     aip    clr             Ee      ea <- 0
36 0100 001 011 EEEEEE  MOVEccr aip                    Ee      68010: ea <- CCR [word mode]
37 0100 010 0SS EEEEEE  NEG     aip                    Ee      ea <- 0 - [ea]
38 0100 010 011 EEEEEE  MOVEccr a                      Ee      CCR <- [ea] [word mode]
39 0100 011 0SS EEEEEE  NOT     aip                    Ee      ea <- ~[ea]
40 0100 011 011 EEEEEE  MOVEsr  a                      Ee      SR <- [ea] [word mode]
41 0100 100 000 EEEEEE  NBCD    aip                    Ee      decimal NEGX
42 0100 100 000 001RRR  LINKL                                  68020: link long
43 0100 100 001 000RRR  SWAP                                   D(RRR)(15..0) <-> D(RRR)(31..16)
44 0100 100 001 001III  BKPT                                   68010: hw breakpoint
45 0100 100 001 EEEEEE  PEA     adri   pea             Ee      Push Effective Address
46 0100 100 010 000RRR  EXT                                    Extend byte to word, D(RRR)
47 0100 100 011 000RRR  EXT                                    Extend word to long, D(RRR)
48 0100 100 111 000RRR  EXTB                                   68020: extend byte to long, D(RRR)
49 0100 100 01Z EEEEEE  MOVEM   ad+ip                  Ee      EW1 details regs to transfer to memory
50 0100 101 0SS EEEEEE  TST     ai     tst             Ee      [ea] CMP
51 0100 101 011 EEEEEE  TAS     aip                    Ee      [ea] CMP, ea <- [ea] | 0x80. MP sync.
52 0100 101 011 111100  ILLEGAL                                illegal instruction
53 0100 110 00X EEEEEE  MULU                           Ee      68020: long multiply/divide. AMs unknown.
54 0100 110 01Z EEEEEE  MOVEM   ad-i                   Ee      EW1 details regs to transfer from memory
55
56 0100 111 001 010RRR  LINK            link                   word link to A(RRR)
57 0100 111 001 011RRR  UNLK            unlk                   unlink
58 0100 111 001 00IIII  TRAP                                   trap w/ immediate vector
59 0100 111 001 100RRR  MOVEUSP                                USP <- A(RRR)
60 0100 111 001 101RRR  MOVEUSP                                A(RRR) <- USP
61 0100 111 001 110000  RESET
62 0100 111 001 110001  NOP             nop
63 0100 111 001 110010  STOP                                   Like Z80 HALT, but with SR <- EW1
64 0100 111 001 110011  RTE                                    Return from Exception
65 0100 111 001 110100  RTD                                    68010: RTS with displacement.
66 0100 111 001 110101  RTS             rts
67 0100 111 001 110110  TRAPV                                  if CCR[V], then vector (overflow)
68 0100 111 001 110111  RTR                                    Return from sub with CCR restore
69 0100 111 001 11101X  MOVEC                                  68010: move general reg to control reg
70 0100 111 010 EEEEEE  JSR     adri   jsr             Ee
71 0100 111 011 EEEEEE  JMP     adri   jmp             Ee
72
73 0101 III 0SS EEEEEE  ADDQ    ip     alu_q_family    +Ee     ea <- [ea] + III
74 0101 CCC C11 EEEEEE  Scc     aip    scc             Ee      if CONDITION then ea <- ~0 else ea <- 0
75 0101 CCC C11 001RRR  DBcc    aip    decbranch               if !CONDITION then D(RRR)--, PC <- PC + d
76 0101 CCC C11 11101X  TRAPcc                                 68020: trap on condition. (word/long operand)
77 0101 CCC C11 111100  TRAPcc                                 68020: trap on condition. (no operand)
78 0101 III 1SS EEEEEE  SUBQ    ip     alu_q_family    -Ee     ea <- [ea] - III
79 0110 ccc cII IIIIII  Bcc             branch                 if d=0, d=EW1. if d=255, d=EW1EW2. PC<-PC+d
80 0110 000 1II IIIIII  BSR                                    same as above wrt d. branch subroutine.
81 0111 RRR 0II IIIIII  MOVEQ           moveq                  D(RRR) <- d
82
```

```
 83 1000 RRR 0SS EEEEEE  OR      a      alu_no_family   |Ee     D(RRR) <- D(RRR) | [ea]
 84 1000 RRR 1SS EEEEEE  OR      adip   alu_no_family   |Ee     ea <- D(RRR) | [ea]
 85 1000 RRR X11 EEEEEE  DIV                           Ee      DIVIDE
 86 1000 RRR 100 00XRRR  SBCD                                  decimal subtract
 87 1000 RRR 101 00XRRR  PACK                                  68020
 88 1000 RRR 110 00XRRR  UNPK                                  68020
 89 1001 RRR 0SS EEEEEE  SUB             alu_no_family   -Ee     D(RRR) <- D(RRR) - [ea]
 90 1001 RRR 1SS EEEEEE  SUB     adip    alu_no_family   -Ee     ea <- D(RRR) - [ea]
 91 1001 RRR 011 EEEEEE  SUBA            alu_a_family    -Ee     ea <- A(RRR) - [ea]   [word, sign ex]
 92 1001 RRR 111 EEEEEE  SUBA            alu_a_family    -Ee     ea <- A(RRR) - [ea]   [long]
 93 1001 RRR 1SS 000rrr  SUBX                            -       D(RRR) <- D(RRR) - D(rrr) - X
 94 1001 RRR 1SS 001rrr  SUBX                            -       [-A(RRR)]<-[A(RRR)]-[-A(rrr)] - X (predec)
 95 1011 RRR 0SS EEEEEE  CMP             alu_no_cmp      cEe     D(RRR) - [ea] CMP
 96 1011 RRR 011 EEEEEE  CMPA            alu_a_cmp       cEe     A(RRR) - [ea] CMP [word sign ex]
 97 1011 RRR 111 EEEEEE  CMPA            alu_a_cmp       cEe     A(RRR) - [ea] CMP [long]
 98 1011 RRR 1SS EEEEEE  EOR     aip     alu_no_family   ^Ee     ea <- [ea] ^ D(RRR)
 99 1011 RRR 1SS 001rrr  CMPM                            c       [-A(RRR)]-[-A(rrr)] CMP, memory compare
100
101 1100 RRR 0SS EEEEEE  AND     a      alu_no_family   &Ee     D(RRR) <- D(RRR) & [ea]
102 1100 RRR 1SS EEEEEE  AND     adip   alu_no_family   &Ee     ea <- D(RRR) & [ea]
103 1100 RRR X11 EEEEEE  MULT                          Ee
104 1100 RRR 100 00XRRR  ABCD                                  decimal add
105 1100 RRR 101 000rrr  EXCH                                  D(RRR) <-> D(rrr)
106 1100 RRR 101 001rrr  EXCH                                  A(RRR) <-> A(rrr)
107 1100 RRR 110 001rrr  EXCH                                  D(RRR) <-> A(rrr)
108 1101 RRR 0SS EEEEEE  ADD             alu_no_family   +Ee     D(RRR) <- D(RRR) + [ea]
109 1101 RRR 1SS EEEEEE  ADD     adip    alu_no_family   +Ee     ea <- D(RRR) + [ea]
110 1101 RRR 011 EEEEEE  ADDA            alu_a_family    +Ee     ea <- A(RRR) + [ea]   [word, sign ex]
111 1101 RRR 111 EEEEEE  ADDA            alu_a_family    +Ee     ea <- A(RRR) + [ea]   [long]
112 1101 RRR 1SS 000rrr  ADDX                            +       D(RRR) <- D(RRR) - D(rrr) - X
113 1101 RRR 1SS 001rrr  ADDX                            +       [-A(RRR)]<-[A(RRR)]-[-A(rrr)] - X (predec)
114 1110 III 0SS 000RRR  ASR
115 1110 III 0SS 001RRR  LSR
116 1110 III 0SS 010RRR  ROXR
117 1110 III 0SS 011RRR  ROR
118 1110 III 1SS 000RRR  ASL
119 1110 III 1SS 001RRR  LSL
120 1110 III 1SS 010RRR  ROXL
121 1110 III 1SS 011RRR  ROL
122 1110 RRR 0SS 100rrr  ASR                                   Shift by D(RRR)
123 1110 RRR 0SS 101rrr  LSR                                   Shift by D(RRR)
124 1110 RRR 0SS 110rrr  ROXR                                  Rotate by D(RRR)
125 1110 RRR 0SS 111rrr  ROR                                   Rotate by D(RRR)
126 1110 RRR 1SS 100rrr  ASL                                   Shift by D(RRR)
127 1110 RRR 1SS 101rrr  LSL                                   Shift by D(RRR)
128 1110 RRR 1SS 110rrr  ROXL                                  Rotate by D(RRR)
129 1110 RRR 1SS 111rrr  ROL                                   Rotate by D(RRR)
130
131 1110 000 011 EEEEEE  ASR     adip                  Ee      Shift memory
132 1110 001 011 EEEEEE  LSR     adip                  Ee      Shift memory
133 1110 010 011 EEEEEE  ROXR    adip                  Ee      Rotate memory
134 1110 011 011 EEEEEE  ROR     adip                  Ee      Rotate memory
135 1110 000 111 EEEEEE  ASL     adip                  Ee      Shift memory
136 1110 001 111 EEEEEE  LSL     adip                  Ee      Shift memory
137 1110 010 111 EEEEEE  ROXL    adip                  Ee      Rotate memory
138 1110 011 111 EEEEEE  ROL     adip                  Ee      Rotate memory
139 1110 1XX X11 EEEEEE  BF      aimp                  Ee      68020 ext BF* not totally correct EA.
140
141 1111 XXX XXX XXXXXX  cp                                    Unsupported Coprocessor Instruction
142
```

I. State Machine Sequences

I.1. Source code of alu_a_family.sm

```
1 LABEL alu_a_family
2
3     if instruction_register ( 8 ) = '1'
4     then
5         operation_size_control <= SET_TO_DWORD ;
6     else
```

```
 7         operation_size_control <= SET_TO_WORD ;
 8     end if ;
 9
10     CLOCK
11
12     -- ADDA or SUBA
13
14     -- always ARF(B) <- ARF(B) <op> EA
15     -- Note: ALU_A_FAMILY uses reverse subtraction, so the
16     -- reversal of the operands is ok.
17
18     case ea_mode is
19     when "000" =>
20                 -- ARF(B) <- ARF(B) <op> DRF(A)
21                 alu_mode <= ALU_A_FAMILY ;
22                 alu_source_a <= ALU_A_DATA_X ;
23                 alu_source_b <= ALU_B_ADDRESS_Y ;
24                 reg_update_address_x <= '1' ;
25                 alu_reverse_operands <= '1' ;
26                 register_file_source_x <= RF_X_11_TO_9_FIELD ;
27                 RETURN
28     when "001" =>
29                 -- ARF(B) <- ARF(B) <op> ARF(A)
30                 alu_mode <= ALU_A_FAMILY ;
31                 alu_source_a <= ALU_A_ADDRESS_X ;
32                 alu_source_b <= ALU_B_ADDRESS_Y ;
33                 alu_reverse_operands <= '1' ;
34                 reg_update_address_x <= '1' ;
35                 register_file_source_x <= RF_X_11_TO_9_FIELD ;
36                 RETURN
37     when others =>
38                 -- ARF(B) <- ARF(B) <op> EA
39                 CALL decode_ea_and_dereference
40     end case ;
41
42     -- And now, do the work.
43     CLOCK
44
45     -- If we have reached here, EA must be a memory address.
46     -- Do the computation and store in ARF(B)
47
48     alu_mode <= ALU_A_FAMILY ;
49     alu_source_a <= ALU_A_OPERAND_VALUE ;
50     alu_source_b <= ALU_B_ADDRESS_Y ;
51     alu_reverse_operands <= '1' ;
52     register_file_source_x <= RF_X_11_TO_9_FIELD ;
53     reg_update_address_x <= '1' ;
54     RETURN
55     CLOCK
56
```

I.2. Source code of `alu_a_family_cmp.sm`

```
 1 LABEL alu_a_cmp
 2
 3     if instruction_register ( 8 ) = '1'
 4     then
 5         operation_size_control <= SET_TO_DWORD ;
 6     else
 7         operation_size_control <= SET_TO_WORD ;
 8     end if ;
 9
10     CLOCK
11
12     -- CMPA
13
14     -- always do ARF(B) <op> EA
15     -- Note: ALU_A_FAMILY uses reverse subtraction, so the
16     -- reversal of the operands is ok.
17
18     case ea_mode is
19     when "000" =>
20                 -- ARF(B) <op> DRF(A)
```

```
21                  alu_mode <= ALU_A_FAMILY ;
22                  alu_source_a <= ALU_A_DATA_X ;
23                  alu_source_b <= ALU_B_ADDRESS_Y ;
24                  alu_reverse_operands <= '1' ;
25                  RETURN
26      when "001" =>
27                  -- ARF(B) <op> ARF(A)
28                  alu_mode <= ALU_A_FAMILY ;
29                  alu_source_a <= ALU_A_ADDRESS_X ;
30                  alu_source_b <= ALU_B_ADDRESS_Y ;
31                  alu_reverse_operands <= '1' ;
32                  RETURN
33      when others =>
34                  -- ARF(B) <op> EA
35                  CALL decode_ea_and_dereference
36      end case ;
37
38      CLOCK
39
40      -- If we have reached here, EA must be a memory address.
41      -- Do the computation.
42
43      alu_mode <= ALU_A_FAMILY ;
44      alu_source_a <= ALU_A_OPERAND_VALUE ;
45      alu_source_b <= ALU_B_ADDRESS_Y ;
46      alu_reverse_operands <= '1' ;
47      RETURN
48      CLOCK
49
```

I.3. Source code of alu_i_cmp.sm

```
1  LABEL alu_i_cmp
2
3       -- For the CMPI instruction
4
5       operation_size_control <= SET_TO_IR ;
6
7       CALL fetch_immediate_data
8       CLOCK
9
10      -- This operation is:
11      -- EA - <immediate>
12      -- The immediate value precedes the effective address.
13
14      case ea_mode is
15      when "000" =>
16                  -- DRF(A) <op> IDR
17                  alu_mode <= ALU_I_FAMILY ;
18                  alu_source_a <= ALU_A_DATA_X ;
19                  alu_source_b <= ALU_B_IDR ;
20                  RETURN
21      -- when "001" => -- not allowed!
22      when others =>
23                  -- EA <- EA <op> IDR
24                  CALL decode_ea_and_dereference
25      end case ;
26
27      -- And now, do the work.
28      CLOCK
29
30      -- If we have reached here, EA must be a memory address.
31      -- Do the computation..
32
33      alu_mode <= ALU_I_FAMILY ;
34      alu_source_a <= ALU_A_OPERAND_VALUE ;
35      alu_source_b <= ALU_B_IDR ;
36      RETURN
37      CLOCK
38
```

I.4. Source code of alu_i_family.sm

```
1 LABEL alu_i_family
2
3     operation_size_control <= SET_TO_IR ;
4
5     -- ADDI, ANDI, EORI, ORI, SUBI
6     -- but not CMPI
7
8     CALL fetch_immediate_data
9     CLOCK
10
11    -- This operation is:
12    -- EA <- EA - <immediate>
13    -- The immediate value precedes the effective address.
14
15    case ea_mode is
16    when "000" =>
17                -- DRF(A) <- DRF(A) <op> IDR
18                alu_mode <= ALU_I_FAMILY ;
19                alu_source_a <= ALU_A_DATA_X ;
20                alu_source_b <= ALU_B_IDR ;
21                reg_update_data_x <= '1' ;
22                RETURN
23    -- when "001" => -- not allowed!
24    when others =>
25                -- EA <- EA <op> IDR
26                CALL decode_ea_and_dereference
27    end case ;
28
29    -- And now, do the work.
30    CLOCK
31
32    -- If we have reached here, EA must be a memory address.
33    -- Do the computation..
34
35    alu_mode <= ALU_I_FAMILY ;
36    alu_source_a <= ALU_A_OPERAND_VALUE ;
37    alu_source_b <= ALU_B_IDR ;
38    operand_value_source <= ALU_TO_OV ;
39    JUMP store_operand_value
40    CLOCK
41
```

I.5. Source code of alu_no_cmp.sm

```
1 LABEL alu_no_cmp
2
3     operation_size_control <= SET_TO_IR ;
4
5     -- CMP instruction
6
7     -- Operation is DRF(B) <op> EA
8
9     case ea_mode is
10    when "000" =>
11                  -- DRF(B) <op> DRF(A)
12                  alu_mode <= ALU_NO_FAMILY ;
13                  alu_source_a <= ALU_A_DATA_X ;
14                  alu_source_b <= ALU_B_DATA_Y ;
15                  alu_reverse_operands <= '1' ;
16                  RETURN
17    when "001" =>
18                  -- DRF(B) <op> ARF(A)
19                  alu_mode <= ALU_NO_FAMILY ;
20                  alu_source_a <= ALU_A_ADDRESS_X ;
21                  alu_source_b <= ALU_B_DATA_Y ;
22                  alu_reverse_operands <= '1' ;
23                  RETURN
24    when others =>  -- DRF(B) <op> EA
25                  CALL decode_ea_and_dereference
26    end case ;
27
```

```
28      -- And now, do the work.
29      CLOCK
30
31      -- If we have reached here, EA must be a memory address.
32      -- Do the computation.
33
34      -- DRF(B) <op> EA
35      alu_mode <= ALU_NO_FAMILY ;
36      alu_source_a <= ALU_A_OPERAND_VALUE ;
37      alu_source_b <= ALU_B_DATA_Y ;
38      alu_reverse_operands <= '1' ;
39      RETURN
40      CLOCK
41
```

I.6. Source code of alu_no_family.sm

```
 1 LABEL alu_no_family
 2
 3      operation_size_control <= SET_TO_IR ;
 4
 5      -- One of ADD, SUB, OR, EOR, AND
 6      -- but not CMP
 7
 8      -- The operation may be in either direction:
 9      -- If IR(8) = 0, then operation is DRF(B) <- DRF(B) <op> EA
10      -- If IR(8) = 1, then operation is EA <- EA <op> DRF(B)
11      -- The EA mode is never Register Direct if IR(8) = 1.
12
13      case ea_mode is
14      when "000" =>
15                      -- IR(8) = 0 DRF(B) <- DRF(B) <op> DRF(A)
16                      -- Set A=B so as to program DRF(B)..
17                      register_file_source_x <= RF_X_11_TO_9_FIELD ;
18                      alu_mode <= ALU_NO_FAMILY ;
19                      alu_source_a <= ALU_A_DATA_X ;
20                      alu_source_b <= ALU_B_DATA_Y ;
21                      alu_reverse_operands <= '1' ;
22                      reg_update_data_x <= '1' ;
23                      RETURN
24      when "001" =>
25                      -- IR(8) = 0 DRF(B) <- DRF(B) <op> ARF(A)
26                      -- Set A=B so as to program DRF(B)..
27                      register_file_source_x <= RF_X_11_TO_9_FIELD ;
28                      alu_mode <= ALU_NO_FAMILY ;
29                      alu_source_a <= ALU_A_ADDRESS_X ;
30                      alu_source_b <= ALU_B_DATA_Y ;
31                      alu_reverse_operands <= '1' ;
32                      reg_update_data_x <= '1' ;
33                      RETURN
34      when others =>  -- IR(8) = 0 DRF(B) <- DRF(B) <op> EA
35                      -- IR(8) = 1 EA <- EA <op> DRF(B)
36                      CALL decode_ea_and_dereference
37      end case ;
38
39      -- And now, do the work.
40      CLOCK
41
42      -- If we have reached here, EA must be a memory address.
43      -- Do the computation.
44
45      alu_mode <= ALU_NO_FAMILY ;
46
47      if ( instruction_register ( 8 ) = '0' )
48      then
49          -- IR(8) = 0 DRF(B) <- DRF(B) <op> EA
50          -- store in a register, specifically DRF(B).
51          register_file_source_x <= RF_X_11_TO_9_FIELD ;
52          reg_update_data_x <= '1' ;
53          alu_source_a <= ALU_A_OPERAND_VALUE ;
54          alu_source_b <= ALU_B_DATA_Y ;
55          alu_reverse_operands <= '1' ;
56          RETURN
```

```
57      else
58          -- IR(8) = 1 EA <- EA <op> DRF(B)
59          -- store in memory. First do the computation.
60          operand_value_source <= ALU_TO_OV ;
61          alu_source_a <= ALU_A_OPERAND_VALUE ;
62          alu_source_b <= ALU_B_DATA_Y ;
63
64          -- Then store the data.
65          JUMP store_operand_value
66      end if ;
67
68      CLOCK
69
```

I.7. Source code of alu_q_family.sm

```
 1 LABEL alu_q_family
 2
 3      operation_size_control <= SET_TO_IR ;
 4
 5      -- One of ADDQ or SUBQ.
 6
 7      -- What is the instruction acting upon?
 8      case ea_mode is
 9      when "000" => -- Acting on a Data register
10              alu_mode <= ALU_Q_FAMILY ;
11              alu_source_a <= ALU_A_DATA_X ;
12              alu_source_b <= ALU_B_PGI ;
13              pgi_source <= PGI_QUICK_IMMEDIATE ;
14
15              -- so the output must go back into the register file.
16              reg_update_data_x <= '1' ;
17              RETURN
18      when "001" => -- Acting on an Address register
19              alu_mode <= ALU_Q_FAMILY ;
20              alu_source_a <= ALU_A_ADDRESS_X ;
21              alu_source_b <= ALU_B_PGI ;
22              pgi_source <= PGI_QUICK_IMMEDIATE ;
23
24              -- so the output must go back into the register file.
25              reg_update_address_x <= '1' ;
26              RETURN
27      when others => -- Acting on a memory address
28              CALL decode_ea_and_dereference
29      end case ;
30
31      -- Now the ALU control lines are programmed. One clock
32      -- cycle later, and (if the source was a register) the work
33      -- is done. The microsubroutine returns.
34      CLOCK
35
36      -- If the execution reaches this point, we are working on
37      -- a memory location. The EA has been dereferenced, so both
38      -- OA and OV have the right values. Apply the operation to OV
39      -- and store the result in OA.
40
41      alu_mode <= ALU_Q_FAMILY ;
42      alu_source_a <= ALU_A_OPERAND_VALUE ;
43      alu_source_b <= ALU_B_PGI ;
44      pgi_source <= PGI_QUICK_IMMEDIATE ;
45      operand_value_source <= ALU_TO_OV ;
46
47      -- Now store it.
48      JUMP store_operand_value
49      CLOCK
50
```

I.8. Source code of branch.sm

```
1 LABEL branch
2
```

```
3     -- This machine handles BRA and B<cc>
4     -- (branch always, branch on condition code) but doesn't handle BSR
5     -- (branch to subroutine)
6
7     -- First, fetch extension word/dword if it is present.
8     -- Use fetch_immediate_data
9     -- If the low byte of IR is 0, then there is one extension word
10    -- containing a 16 bit branch offset. If the low byte of IR is 255,
11    -- then there is an extension dword containing a 32 bit branch offset.
12    -- Otherwise there are no extension words.
13    if ( instruction_register ( 7 downto 0 ) = "0000000" )
14    then
15        -- There is one extension word.
16        -- We also set a flag so that after the fetch, PC will be
17        -- restored to the value it has at present.
18        -- The immediate value precedes the effective address.
19
20        restore_pc_after_immediate_fetch_set <= '1' ;
21        operation_size_control <= SET_TO_WORD ;
22        CALL fetch_immediate_data
23    elsif ( instruction_register ( 7 downto 0 ) = "11111111" )
24    then
25        -- There's a long extension word.
26        -- We also set a flag so that after the fetch, PC will be
27        -- restored to the value it has at present.
28
29        restore_pc_after_immediate_fetch_set <= '1' ;
30        operation_size_control <= SET_TO_DWORD ;
31        CALL fetch_immediate_data
32    else
33        -- No extension words. We can branch now if condition_true = '1'.
34        if ( condition_true = '1' )
35        then
36            alu_mode <= ALU_ADD ;
37            alu_source_a <= ALU_A_PC ;
38            alu_source_b <= ALU_B_LOW_BYTE_OF_IR ;
39            pc_source <= ALU_TO_PC ;
40        end if ;
41        RETURN
42    end if ;
43    CLOCK
44
45    -- There were one or two extension words.
46    if ( condition_true = '1' )
47    then
48        alu_mode <= ALU_ADD ;
49        alu_source_a <= ALU_A_PC ;
50        alu_source_b <= ALU_B_IDR ;
51        pc_source <= ALU_TO_PC ;
52    else
53        -- Jump over extension words.
54        alu_mode <= ALU_ADD ;
55        alu_source_a <= ALU_A_PC ;
56        pc_source <= ALU_TO_PC ;
57        if ( instruction_register ( 7 downto 0 ) = "00000000" )
58        then
59            -- 1 word
60            alu_source_b <= ALU_B_PGI ;
61            pgi_source <= PGI_TWO ;
62        else
63            -- 1 dword
64            alu_source_b <= ALU_B_PGI ;
65            pgi_source <= PGI_FOUR ;
66        end if ;
67    end if ;
68
69    RETURN
70    CLOCK
71
```

I.9. Source code of `clr.sm`

```
1 LABEL clr
```

```
 2
 3     -- CLR: write zero to an effective address
 4
 5     operation_size_control <= SET_TO_IR ;
 6
 7     case ea_mode is
 8     when "000" =>
 9                     -- DRF(A) <- 0
10                     alu_mode <= ALU_CLR_FAMILY ;
11                     alu_source_a <= ALU_A_PGI ;
12                     alu_source_b <= ALU_B_PGI ;
13                     pgi_source <= PGI_ZERO ;
14                     reg_update_data_x <= '1' ;
15                     RETURN
16     when "001" =>
17                     -- ARF(A) <- 0
18                     alu_mode <= ALU_CLR_FAMILY ;
19                     alu_source_a <= ALU_A_PGI ;
20                     alu_source_b <= ALU_B_PGI ;
21                     pgi_source <= PGI_ZERO ;
22                     reg_update_address_x <= '1' ;
23                     RETURN
24     when others =>  -- EA <- 0 -- No need to dereference EA.
25                     CALL decode_ea
26     end case ;
27
28     CLOCK
29
30     alu_mode <= ALU_CLR_FAMILY ;
31     alu_source_a <= ALU_A_PGI ;
32     alu_source_b <= ALU_B_PGI ;
33     pgi_source <= PGI_ZERO ;
34     operand_value_source <= ALU_TO_OV ;
35     JUMP store_operand_value
36
37     CLOCK
38
```

I.10. Source code of `decbranch.sm`

```
 1 LABEL decbranch
 2
 3     -- This machine is for DB<cc>
 4     -- (decrement and branch on condition)
 5
 6     -- If condition is true, then do nothing.
 7     -- (Although we must still skip the extension word containing
 8     -- the displacement)
 9
10     operation_size_control <= SET_TO_WORD ;
11     restore_pc_after_immediate_fetch_set <= '1' ;
12     alu_mode <= ALU_ADD ;
13     alu_source_a <= ALU_A_PC ;
14     alu_source_b <= ALU_B_PGI ;
15     pgi_source <= PGI_TWO ;
16
17     if ( condition_true = '1' )
18     then
19         -- Jump over extension word.
20         pc_source <= ALU_TO_PC ;
21         RETURN
22     else
23         -- Fetch the displacement
24         CALL fetch_immediate_data
25     end if ;
26     CLOCK
27
28     -- Do the following things
29     -- Dn <- Dn - 1 (decrement Dn)
30     -- If Dn /= -1, then PC <- PC + d
31
32     -- This should be a word mode operation. Should it affect CC's?
33     alu_mode <= ALU_SUBTRACT ;
```

```
34      alu_source_a <= ALU_A_DATA_X ;
35      alu_source_b <= ALU_B_PGI ;
36      pgi_source <= PGI_ONE ;
37      reg_update_data_x <= '1' ;
38
39      CLOCK
40      -- is Dn == -1?
41
42      alu_mode <= ALU_ADD ;
43      alu_source_a <= ALU_A_DATA_X ;
44      alu_source_b <= ALU_B_PGI ;
45      pgi_source <= PGI_ZERO ;
46
47      if ( alu_output_is_minus_one = '1' )
48      then
49          -- Stop looping if the data register is exactly -1.
50          JUMP dbcc_stop_looping
51      end if ;
52
53      CLOCK
54
55      -- dbcc_continue_looping
56
57      -- PC <- PC + d
58      alu_mode <= ALU_ADD ;
59      alu_source_a <= ALU_A_PC ;
60      alu_source_b <= ALU_B_IDR ;
61      pc_source <= ALU_TO_PC ;
62      RETURN
63      CLOCK
64
65  LABEL dbcc_stop_looping
66      -- Jump over the extension word.
67      alu_mode <= ALU_ADD ;
68      alu_source_a <= ALU_A_PC ;
69      alu_source_b <= ALU_B_PGI ;
70      pgi_source <= PGI_TWO ;
71      pc_source <= ALU_TO_PC ;
72      RETURN
73      CLOCK
74
```

I.11. Source code of `decode_ea.sm`

```
 1  LABEL decode_ea
 2
 3  -- Decode an effective address field in an instruction, storing the
 4  -- effective address itself in the OPERAND_ADDRESS register.
 5  -- Prereqs:
 6  --      The instruction must have an effective address field.
 7  --      The effective address mode is not 0 or 1.
 8  --          i.e. the mode is not register direct.
 9  -- Postconditions:
10  --      EA loaded into operand_address.
11
12      -- Check mode field. This is normally instruction_register ( 5 downto 3 )
13      -- but not always (see the MOVE instruction).
14      case apply_ea_mode ( ea_mode ) ( 2 downto 0 ) is
15
16      when "010" =>
17          -- Address Register Indirect mode:
18          -- OA <- AR ( ea_reg )
19
20          -- since, by default, register_file_source_x = RF_X_EA_REG,
21          -- the address register chosen by ea_reg will already be on
22          -- the output of the register file.
23
24          alu_mode <= ALU_ADD ;
25          alu_source_a <= ALU_A_ADDRESS_X ;
26          alu_source_b <= ALU_B_PGI ;
27          pgi_source <= PGI_ZERO ;
28          operand_address_source <= ALU_TO_OA ;
29          RETURN
```

```
30
31      when "011" =>
32          -- Address Register Indirect (with Postinc)
33          -- [1] OA <- AR ( ea_reg )
34          -- [2] AR ( ea_reg ) <- AR ( ea_reg ) + size
35
36          -- We do the first stage now.
37          alu_mode <= ALU_ADD ;
38          alu_source_a <= ALU_A_ADDRESS_X ;
39          alu_source_b <= ALU_B_PGI ;
40          pgi_source <= PGI_ZERO ;
41          operand_address_source <= ALU_TO_OA ;
42
43          -- and then do the second part in a helper state
44          JUMP decode_ea_postinc_helper
45
46      when "100" =>
47          -- Address Register Indirect (with Predec)
48          -- [1] AR ( ea_reg ) <- AR ( ea_reg ) - size
49          -- [2] OA <- AR ( ea_reg )
50
51          -- First, do the subtraction
52          alu_mode <= ALU_SUBTRACT ;
53          alu_source_a <= ALU_A_ADDRESS_X ;
54          alu_source_b <= ALU_B_PGI ;
55          pgi_source <= PGI_POSTINC_PREDEC ;
56          reg_update_address_x <= '1' ;
57          reg_update_override_size <= '1' ;
58
59          -- Do the second part in a helper state
60          JUMP decode_ea_predec_helper
61
62      when "101" =>
63          -- Address Register Indirect with Displacement
64          -- The first extension word must be fetched. It goes into OA.
65          CALL fetch_extension_word
66
67      when "110" =>
68          -- This mode isn't supported.
69          CALL fault
70
71      when "111" =>
72          -- Check register field.
73          case apply_ea_reg ( ea_reg ) ( 2 downto 0 ) is
74          when "000" =>   -- Absolute address (Word mode)
75              JUMP fetch_extension_word
76          when "001" =>   -- Absolute address (Dword mode)
77              JUMP fetch_extension_dword
78          when "010"|"011" =>
79              -- PC relative w/ Displacement
80              -- PC memory indirect w/ index
81              -- The extension word must be fetched into OA
82              CALL fetch_extension_word
83          when others =>   -- Immediate
84              -- For word or long immediates, we copy OA <- PC.
85              -- For byte immediates, use OA <- PC+1 to get the
86              -- correct address of the data.
87              alu_mode <= ALU_ADD ;
88              alu_source_a <= ALU_A_PC ;
89              if ( operation_size = BYTE )
90              then
91                  alu_source_b <= ALU_B_PGI ;
92                  pgi_source <= PGI_ONE ;
93              else
94                  alu_source_b <= ALU_B_PGI ;
95                  pgi_source <= PGI_ZERO ;
96              end if ;
97              operand_address_source <= ALU_TO_OA ;
98          end case ;
99      when others =>
100         CALL fault
101     end case ;
102     CLOCK
103
```

```
104     -- By this point, many of the addressing modes will have caused
105     -- the decode_ea subroutine to return. The ones requiring some
106     -- extension word, however, will now have fetched that word.
107     -- Here we deal with it.
108     case apply_ea_mode ( ea_mode ) ( 2 downto 0 ) is
109     when "101" =>
110         -- Address Register Indirect with Displacement
111         -- The displacement is already in OA.
112         -- OA <- AR ( ea_reg ) + OA.
113
114         alu_mode <= ALU_ADD ;
115         alu_source_a <= ALU_A_ADDRESS_X ;
116         alu_source_b <= ALU_B_OA ;
117         operand_address_source <= ALU_TO_OA ;
118         RETURN
119     -- when "110" => TBD.
120     when "111" =>
121         -- Check register field.
122         case apply_ea_reg ( ea_reg ) ( 2 downto 0 ) is
123         when "010" =>
124             -- PC relative w/ Displacement
125             -- The displacement is already in OA.
126
127             -- OA <- PC + OA
128             alu_mode <= ALU_ADD ;
129             alu_source_a <= ALU_A_PC ;
130             alu_source_b <= ALU_B_OA ;
131             operand_address_source <= ALU_TO_OA ;
132
133             -- Unforunately, the value loaded into OA is actually
134             -- the address plus 2, since the PC relative address
135             -- is relative to the end of the opcode not the end of
136             -- the instruction word! Correct for this.
137             JUMP correct_pc_relative
138         -- when "011" =>  TBD
139         when others =>   -- Immediate
140             -- If the effective address is an immediate, the PC will need
141             -- incrementing over the immediate so that it isn't executed
142             -- as code.
143
144             if ( ea_move_destination_control = '0' )
145             then
146                 case operation_size is
147                 when BYTE|WORD =>
148                         -- Add two to the PC. PC <- PC + 2
149                         pgi_source <= PGI_TWO ;
150                 when others => -- DWORD
151                         -- Add four to the PC. PC <- PC + 4
152                         pgi_source <= PGI_FOUR ;
153                 end case ;
154                 alu_mode <= ALU_ADD ;
155                 alu_source_a <= ALU_A_PC ;
156                 alu_source_b <= ALU_B_PGI ;
157                 pc_source <= ALU_TO_PC ;
158
159                 RETURN
160             else
161                 CALL fault
162             end if ;
163         end case ;
164     when others =>
165         CALL fault
166     end case ;
167     CLOCK
168
169 -- decode_ea helper functions. These simplify the state machine
170 -- somewhat.
171 LABEL decode_ea_postinc_helper
172     alu_mode <= ALU_ADD ;
173     alu_source_a <= ALU_A_ADDRESS_X ;
174     alu_source_b <= ALU_B_PGI ;
175     pgi_source <= PGI_POSTINC_PREDEC ;
176     reg_update_address_x <= '1' ;
177     reg_update_override_size <= '1' ;
```

```
178     RETURN
179     CLOCK
180
181 LABEL decode_ea_predec_helper
182     alu_mode <= ALU_ADD ;
183     alu_source_a <= ALU_A_ADDRESS_X ;
184     alu_source_b <= ALU_B_PGI ;
185     pgi_source <= PGI_ZERO ;
186     operand_address_source <= ALU_TO_OA ;
187     RETURN
188     CLOCK
189
190 LABEL correct_pc_relative
191     -- OA <- OA - 2
192
193     alu_mode <= ALU_SUBTRACT ;
194     alu_source_a <= ALU_A_PGI ;
195     alu_source_b <= ALU_B_OA ;
196     alu_reverse_operands <= '1' ;
197     pgi_source <= PGI_TWO ;
198     operand_address_source <= ALU_TO_OA ;
199     RETURN
200     CLOCK
201
```

I.12. Source code of decode_ea_and_dereference.sm

```
 1 LABEL decode_ea_and_dereference
 2
 3 -- Decode an effective address and load data at it.
 4 -- Prereqs:
 5 --      operation_size must be set correctly.
 6 -- Postconditions:
 7 --      EA loaded into operand_address.
 8 --      operand_value <- [operand_address]
 9     case ea_mode is
10     when "000" => -- Source is a Data register
11         alu_mode <= ALU_ADD ;
12         alu_source_a <= ALU_A_DATA_X ;
13         alu_source_b <= ALU_B_PGI ;
14         pgi_source <= PGI_ZERO ;
15         operand_value_source <= ALU_TO_OV ;
16         RETURN
17     when "001" => -- Source is an Address register
18         alu_mode <= ALU_ADD ;
19         alu_source_a <= ALU_A_ADDRESS_X ;
20         alu_source_b <= ALU_B_PGI ;
21         pgi_source <= PGI_ZERO ;
22         operand_value_source <= ALU_TO_OV ;
23         RETURN
24     when others => -- Acting on a memory address
25         CALL decode_ea
26     end case ;
27     CLOCK
28
29 LABEL load_operand_value
30     -- OV <- [OA]
31
32     -- Prepare to fetch 1st byte of operand value.
33     mar_source <= OA_TO_MAR ;
34
35     -- Increment OA to obtain the 2nd byte address
36     alu_mode <= ALU_ADD ;
37     alu_source_a <= ALU_A_PGI ;
38     alu_source_b <= ALU_B_OA ;
39     pgi_source <= PGI_ONE ;
40     operand_address_source <= ALU_TO_OA ;
41     CLOCK
42
43     -- Store the byte that was fetched.
44     case operation_size is
45     when BYTE =>
46             operand_value_source <= MDR_TO_OV_0 ;
```

```
47      when WORD =>
48              operand_value_source <= MDR_TO_OV_1 ;
49      when others => -- DWORD
50              operand_value_source <= MDR_TO_OV_3 ;
51      end case ;
52
53      -- Prepare to fetch the 2nd byte
54      mar_source <= OA_TO_MAR ;
55
56      if ( operation_size = BYTE )
57      then
58          -- Decrement OA to restore original value
59          alu_mode <= ALU_SUBTRACT ;
60
61          RETURN
62      else
63          -- Increment OA to get the 3rd byte address
64          alu_mode <= ALU_ADD ;
65      end if ;
66
67      alu_reverse_operands <= '1' ;
68      alu_source_a <= ALU_A_PGI ;
69      alu_source_b <= ALU_B_OA ;
70      pgi_source <= PGI_ONE ;
71      operand_address_source <= ALU_TO_OA ;
72      CLOCK
73
74      -- Store the byte that was fetched.
75      case operation_size is
76      when WORD =>
77              operand_value_source <= MDR_TO_OV_0 ;
78      when others => -- DWORD
79              operand_value_source <= MDR_TO_OV_2 ;
80      end case ;
81
82      -- Prepare to fetch the 3rd byte
83      mar_source <= OA_TO_MAR ;
84
85      if ( operation_size = WORD )
86      then
87          -- Decrement OA by 2 to restore the original value
88          pgi_source <= PGI_TWO ;
89          alu_mode <= ALU_SUBTRACT ;
90
91          RETURN
92      else
93          -- Increment OA to get the 4th byte address
94          pgi_source <= PGI_ONE ;
95          alu_mode <= ALU_ADD ;
96      end if ;
97      alu_reverse_operands <= '1' ;
98      alu_source_a <= ALU_A_PGI ;
99      alu_source_b <= ALU_B_OA ;
100     operand_address_source <= ALU_TO_OA ;
101     CLOCK
102
103     -- Store the byte that was fetched.
104     operand_value_source <= MDR_TO_OV_1 ;
105
106     -- Prepare the fetch the 4th byte
107     mar_source <= OA_TO_MAR ;
108
109     -- Decrement OA to restore the original value.
110     pgi_source <= PGI_THREE ;
111     alu_mode <= ALU_SUBTRACT ;
112     alu_reverse_operands <= '1' ;
113     alu_source_a <= ALU_A_PGI ;
114     alu_source_b <= ALU_B_OA ;
115     operand_address_source <= ALU_TO_OA ;
116
117     CLOCK
118     -- Store the byte that was fetched.
119     operand_value_source <= MDR_TO_OV_0 ;
120
```

```
121     RETURN
122     CLOCK
123
```

I.13. Source code of decode_ea_and_store.sm

```
 1 LABEL decode_ea_and_store
 2
 3 -- Decode an effective address and store data at it.
 4 -- Prereqs:
 5 --      operation_size must be set correctly.
 6 --      The instruction must have an effective address field.
 7 -- Postconditions:
 8 --      EA loaded into operand_address.
 9 --      operand_value stored at operand_address, with data size
10 --      operation_size.
11     case ea_mode is
12     when "000" =>   -- Data register direct
13                     alu_mode <= ALU_ADD ;
14                     alu_source_a <= ALU_A_OPERAND_VALUE ;
15                     alu_source_b <= ALU_B_PGI ;
16                     pgi_source <= PGI_ZERO ;
17                     reg_update_data_x <= '1' ;
18                     RETURN
19
20     when "001" =>   -- Address register direct
21                     alu_mode <= ALU_ADD ;
22                     alu_source_a <= ALU_A_OPERAND_VALUE ;
23                     alu_source_b <= ALU_B_PGI ;
24                     pgi_source <= PGI_ZERO ;
25                     reg_update_address_x <= '1' ;
26                     RETURN
27
28     when others =>  -- Operating on a memory address.
29                     CALL decode_ea
30     end case ;
31     CLOCK
32
33 -- Store OV at OA.
34 -- Prereqs:
35 --      OA, OV programmed.
36 --      operation_size must be set correctly.
37 -- Postconditions:
38 --      OA <- OV
39 LABEL store_operand_value
40     -- You can't do an immediate store. Don't even bother
41     -- to check for this. (If you could we might need to increment PC here)
42
43     -- Memory[OA] <- OV
44
45     mar_source <= OA_TO_MAR ;
46     case operation_size is
47     when BYTE =>
48             -- There is only one byte to store
49             mdr_source <= OV_0_TO_MDR ;
50             RETURN
51     when WORD =>
52             -- Store the high byte of the word
53             mdr_source <= OV_1_TO_MDR ;
54             alu_mode <= ALU_ADD ;
55             alu_source_a <= ALU_A_PGI ;
56             alu_source_b <= ALU_B_OA ;
57             pgi_source <= PGI_ONE ;
58             operand_address_source <= ALU_TO_OA ;
59     when others => --DWORD
60             -- Store the high byte of the dword
61             mdr_source <= OV_3_TO_MDR ;
62             alu_mode <= ALU_ADD ;
63             alu_source_a <= ALU_A_PGI ;
64             alu_source_b <= ALU_B_OA ;
65             pgi_source <= PGI_ONE ;
66             operand_address_source <= ALU_TO_OA ;
67     end case ;
```

```
68
69      CLOCK
70
71      mar_source <= OA_TO_MAR ;
72      case operation_size is
73      when WORD =>
74              -- Store the low byte of the word
75              mdr_source <= OV_0_TO_MDR ;
76              -- Subtract 1 from OA so that the original value
77              -- is restored.
78              alu_mode <= ALU_SUBTRACT ;
79              alu_source_a <= ALU_A_PGI ;
80              alu_source_b <= ALU_B_OA ;
81              alu_reverse_operands <= '1' ;
82              pgi_source <= PGI_ONE ;
83              operand_address_source <= ALU_TO_OA ;
84              RETURN
85      when others => --DWORD
86              -- Store the next byte of the dword (2)
87              mdr_source <= OV_2_TO_MDR ;
88              alu_mode <= ALU_ADD ;
89              alu_source_a <= ALU_A_PGI ;
90              alu_source_b <= ALU_B_OA ;
91              pgi_source <= PGI_ONE ;
92              operand_address_source <= ALU_TO_OA ;
93      end case ;
94
95      CLOCK
96
97      -- Store the next byte of the dword (1)
98      mar_source <= OA_TO_MAR ;
99      mdr_source <= OV_1_TO_MDR ;
100     alu_mode <= ALU_ADD ;
101     alu_source_a <= ALU_A_PGI ;
102     alu_source_b <= ALU_B_OA ;
103     pgi_source <= PGI_ONE ;
104     operand_address_source <= ALU_TO_OA ;
105
106     CLOCK
107
108     -- Store the low byte of the dword (0)
109     mar_source <= OA_TO_MAR ;
110     mdr_source <= OV_0_TO_MDR ;
111     -- Subtract 3 from OA so that the original value
112     -- is restored.
113     alu_mode <= ALU_SUBTRACT ;
114     alu_source_a <= ALU_A_PGI ;
115     alu_source_b <= ALU_B_OA ;
116     alu_reverse_operands <= '1' ;
117     pgi_source <= PGI_THREE ;
118     operand_address_source <= ALU_TO_OA ;
119
120     RETURN
121     CLOCK
122
```

I.14. Source code of `fetch_extension_dword.sm`

```
1 LABEL fetch_extension_dword
2
3         -- This loads OA <- [PC]. 32 bits are loaded.
4         -- PC <- PC + 4 is also done.
5
6         mar_source <= PC_TO_MAR ;
7         alu_mode <= ALU_ADD ;
8         alu_source_a <= ALU_A_PC ;
9         alu_source_b <= ALU_B_PGI ;
10        pgi_source <= PGI_ONE ;
11        pc_source <= ALU_TO_PC ;
12        CLOCK
13
14        operand_address_source <= MDR_TO_OA_3 ;
15
```

```
16          mar_source <= PC_TO_MAR ;
17          alu_mode <= ALU_ADD ;
18          alu_source_a <= ALU_A_PC ;
19          alu_source_b <= ALU_B_PGI ;
20          pgi_source <= PGI_ONE ;
21          pc_source <= ALU_TO_PC ;
22          CLOCK
23
24          operand_address_source <= MDR_TO_OA_2 ;
25
26          mar_source <= PC_TO_MAR ;
27          alu_mode <= ALU_ADD ;
28          alu_source_a <= ALU_A_PC ;
29          alu_source_b <= ALU_B_PGI ;
30          pgi_source <= PGI_ONE ;
31          pc_source <= ALU_TO_PC ;
32          CLOCK
33
34          operand_address_source <= MDR_TO_OA_1 ;
35
36          mar_source <= PC_TO_MAR ;
37          alu_mode <= ALU_ADD ;
38          alu_source_a <= ALU_A_PC ;
39          alu_source_b <= ALU_B_PGI ;
40          pgi_source <= PGI_ONE ;
41          pc_source <= ALU_TO_PC ;
42
43          CLOCK
44
45          operand_address_source <= MDR_TO_OA_0 ;
46
47          RETURN
48          CLOCK
49
```

I.15. Source code of `fetch_extension_word.sm`

```
 1 LABEL fetch_extension_word
 2
 3          -- This loads OA <- [PC]. 16 bits are loaded. The
 4          -- top 16 bits of OA are sign extended.
 5          -- PC <- PC + 2 is also done.
 6
 7          mar_source <= PC_TO_MAR ;
 8          alu_mode <= ALU_ADD ;
 9          alu_source_a <= ALU_A_PC ;
10          alu_source_b <= ALU_B_PGI ;
11          pgi_source <= PGI_ONE ;
12          pc_source <= ALU_TO_PC ;
13          CLOCK
14
15          operand_address_source <= MDR_TO_OA_1_SE ;
16
17          mar_source <= PC_TO_MAR ;
18          alu_mode <= ALU_ADD ;
19          alu_source_a <= ALU_A_PC ;
20          alu_source_b <= ALU_B_PGI ;
21          pgi_source <= PGI_ONE ;
22          pc_source <= ALU_TO_PC ;
23
24          CLOCK
25
26          operand_address_source <= MDR_TO_OA_0 ;
27
28          RETURN
29          CLOCK
30
```

I.16. Source code of `fetch_immediate_data.sm`

```
1 LABEL fetch_immediate_data
```

```
 2
 3        -- This loads IDR <- [PC], with size according to operation_size.
 4        -- The value in IDR is sign-extended to 32 bits. This means that short
 5        -- immediates can be added directly to PC or OA.
 6
 7        -- PC is advanced appropriately: by 2 for BYTE/WORD immediates
 8        -- and by 4 for DWORD immediates
 9
10        -- Prepare to fetch 1st byte
11        mar_source <= PC_TO_MAR ;
12
13        -- Set PC for fetch of 2nd byte
14        alu_mode <= ALU_ADD ;
15        alu_source_a <= ALU_A_PC ;
16        alu_source_b <= ALU_B_PGI ;
17        pgi_source <= PGI_ONE ;
18        pc_source <= ALU_TO_PC ;
19        CLOCK
20
21        -- Store 1st byte
22        if ( operation_size = BYTE )
23        or ( operation_size = WORD )
24        then
25            -- When the most significant byte of the word is
26            -- loaded, sign extend it to fill the remaining 16 bits of IDR.
27            immediate_data_source <= MDR_TO_IDR_1_SE ;
28        else
29            immediate_data_source <= MDR_TO_IDR_3 ;
30        end if ;
31
32        -- Prepare to fetch 2nd byte
33        mar_source <= PC_TO_MAR ;
34
35        -- Set PC for fetch of 3rd byte
36        alu_mode <= ALU_ADD ;
37        alu_source_a <= ALU_A_PC ;
38        alu_source_b <= ALU_B_PGI ;
39        pgi_source <= PGI_ONE ;
40        pc_source <= ALU_TO_PC ;
41        CLOCK
42
43        -- Store 2nd byte
44        if ( operation_size = BYTE )
45        or ( operation_size = WORD )
46        then
47            if ( operation_size = BYTE )
48            then
49                -- Bytes are sign extended to fill the rest of IDR.
50                immediate_data_source <= MDR_TO_IDR_0_SE ;
51            else
52                immediate_data_source <= MDR_TO_IDR_0 ;
53            end if ;
54
55            -- We may need to put the PC value back to the 1st
56            -- byte.
57
58            if ( restore_pc_after_immediate_fetch = '1' )
59            then
60                -- Do PC <- PC - 2 so that PC returns to the
61                -- value it had when this subroutine was entered
62                alu_source_a <= ALU_A_PC ;
63                alu_source_b <= ALU_B_PGI ;
64                pgi_source <= PGI_TWO ;
65                alu_mode <= ALU_SUBTRACT ;
66                pc_source <= ALU_TO_PC ;
67            end if ;
68
69            RETURN
70        else
71            immediate_data_source <= MDR_TO_IDR_2 ;
72
73            -- Prepare to fetch 3rd byte
74            mar_source <= PC_TO_MAR ;
75
```

```
76              -- Set PC for fetch of 4th byte
77              alu_mode <= ALU_ADD ;
78              alu_source_a <= ALU_A_PC ;
79              alu_source_b <= ALU_B_PGI ;
80              pgi_source <= PGI_ONE ;
81              pc_source <= ALU_TO_PC ;
82          end if ;
83
84          CLOCK
85
86          -- Store 3rd byte
87          immediate_data_source <= MDR_TO_IDR_1 ;
88
89          -- Prepare to fetch 4th byte
90          mar_source <= PC_TO_MAR ;
91
92          if ( restore_pc_after_immediate_fetch = '1' )
93          then
94              -- Set PC back to 1st byte
95              alu_mode <= ALU_SUBTRACT ;
96              alu_source_a <= ALU_A_PC ;
97              alu_source_b <= ALU_B_PGI ;
98              pgi_source <= PGI_THREE ;
99              pc_source <= ALU_TO_PC ;
100         else
101             -- Set PC to point to byte following immediate.
102             alu_mode <= ALU_ADD ;
103             alu_source_a <= ALU_A_PC ;
104             alu_source_b <= ALU_B_PGI ;
105             pgi_source <= PGI_ONE ;
106             pc_source <= ALU_TO_PC ;
107         end if ;
108         CLOCK
109
110         -- Store 4th byte
111         immediate_data_source <= MDR_TO_IDR_0 ;
112
113         RETURN
114         CLOCK
115
```

I.17. Source code of `jmp.sm`

```
1 LABEL jmp
2
3     -- Work out the effective address
4     -- No need to handle register direct modes - they are
5     -- not supported.
6     CALL decode_ea
7     CLOCK
8
9     -- PC <- OA  - jump to the appropriate place.
10    alu_mode <= ALU_ADD ;
11    alu_source_a <= ALU_A_PGI ;
12    alu_source_b <= ALU_B_OA ;
13    pgi_source <= PGI_ZERO ;
14    pc_source <= ALU_TO_PC ;
15
16    RETURN
17    CLOCK
18
```

I.18. Source code of `jsr.sm`

```
1 LABEL jsr
2
3     -- Machine for JSR (Jump to Subroutine)
4     -- The effect is:
5     -- SP <- SP - 4
6     -- M[SP] <- PC for next instruction
7     -- PC <- EA
```

```
 8
 9      -- Decrement SP: SP <- SP - 4
10      -- We would normally have to ensure that register B was
11      -- set to the stack pointer before doing this, so that its
12      -- value was fetched correctly. Fortunately, the default
13      -- setting for register B (IR field 11 to 9) is always '111'
14      -- for this opcode - i.e. already the stack pointer.
15
16      operation_size_control <= SET_TO_DWORD ;
17      register_file_source_x <= RF_X_FORCE_TO_SP ;
18      register_file_source_y <= RF_Y_FORCE_TO_SP ;
19  CLOCK
20      register_file_source_x <= RF_X_FORCE_TO_SP ;
21      register_file_source_y <= RF_Y_FORCE_TO_SP ;
22      alu_mode <= ALU_SUBTRACT ;
23      alu_reverse_operands <= '1' ;
24      alu_source_a <= ALU_A_PGI ;
25      alu_source_b <= ALU_B_ADDRESS_Y ;
26      pgi_source <= PGI_FOUR ;
27      reg_update_address_x <= '1' ;
28
29      CLOCK
30
31      -- Decode the effective address of the subroutine.
32      -- We have to leave an extra clock cycle so that the setting
33      -- of register_file_source_x reverts to normal.
34      CALL decode_ea
35      CLOCK
36
37      -- OV <- PC (store the return PC in OV)
38      -- Can't do this before decode_ea in case there are extension words.
39      alu_mode <= ALU_ADD ;
40      alu_source_a <= ALU_A_PC ;
41      alu_source_b <= ALU_B_PGI ;
42      pgi_source <= PGI_ZERO ;
43      operand_value_source <= ALU_TO_OV ;
44
45      CLOCK
46      -- PC <- OA (store the new PC, from OA)
47
48      alu_mode <= ALU_ADD ;
49      alu_source_a <= ALU_A_PGI ;
50      alu_source_b <= ALU_B_OA ;
51      pgi_source <= PGI_ZERO ;
52      pc_source <= ALU_TO_PC ;
53
54      CLOCK
55      -- OA <- SP (store the new stack pointer in OA)
56      alu_mode <= ALU_ADD ;
57      alu_source_a <= ALU_A_PGI ;
58      alu_source_b <= ALU_B_ADDRESS_Y ;
59      pgi_source <= PGI_ZERO ;
60      operand_address_source <= ALU_TO_OA ;
61
62      -- Now, M[OA] <- OV (store return PC at SP)
63      JUMP store_operand_value
64      CLOCK
65
```

I.19. Source code of `lea.sm`

```
 1 LABEL lea
 2
 3      -- Machine for LEA (Load Effective Address)
 4      -- The effect is:
 5      -- Address Reg <- EA
 6
 7      operation_size_control <= SET_TO_DWORD ;
 8
 9      -- Decode the effective address.
10      -- Note: no need to handle register direct modes.
11      -- They are not supported by LEA.
12      CALL decode_ea
```

```
13      CLOCK
14
15      -- Now, Address Reg <- EA
16      register_file_source_x <= RF_X_11_TO_9_FIELD ;
17      alu_mode <= ALU_ADD ;
18      alu_source_a <= ALU_A_PGI ;
19      alu_source_b <= ALU_B_OA ;
20      pgi_source <= PGI_ZERO ;
21      reg_update_address_x <= '1' ;
22
23      RETURN
24      CLOCK
25
```

I.20. Source code of `link.sm`

```
 1 LABEL link
 2      -- This machine is for the LINK instruction.
 3      -- SP <- SP - 4
 4      -- M[SP] <- ARF_A
 5      -- ARF_A <- SP
 6      -- SP <- SP + IDR
 7
 8      -- Is this a long link or a word link?
 9      if ( instruction_register ( 10 ) = '1' )
10      then
11          -- word link.
12          operation_size_control <= SET_TO_WORD ;
13      else
14          -- long link.
15          operation_size_control <= SET_TO_DWORD ;
16      end if ;
17
18      CALL fetch_immediate_data
19      CLOCK
20      operation_size_control <= SET_TO_DWORD ;
21
22      -- Ensure that register file output A is the stack pointer.
23      register_file_source_x <= RF_X_FORCE_TO_SP ;
24
25      -- Set register file output B to be the address register
26      -- in use for the link.
27      register_file_source_y <= RF_Y_2_TO_0_FIELD ;
28
29      CLOCK
30      -- SP <- SP - 4
31      -- at the same time, do OA <- SP - 4
32      register_file_source_x <= RF_X_FORCE_TO_SP ;
33      register_file_source_y <= RF_Y_2_TO_0_FIELD ;
34      alu_mode <= ALU_SUBTRACT ;
35      alu_source_a <= ALU_A_ADDRESS_X ;
36      alu_source_b <= ALU_B_PGI ;
37      pgi_source <= PGI_FOUR ;
38      reg_update_address_x <= '1' ;
39      operand_address_source <= ALU_TO_OA ;
40      CLOCK
41      -- M[SP] <- An.
42      --  This is done by first copying An to OV.
43      register_file_source_x <= RF_X_FORCE_TO_SP ;
44      register_file_source_y <= RF_Y_2_TO_0_FIELD ;
45      alu_mode <= ALU_ADD ;
46      alu_source_a <= ALU_A_PGI ;
47      alu_source_b <= ALU_B_ADDRESS_Y ;
48      pgi_source <= PGI_ZERO ;
49      operand_value_source <= ALU_TO_OV ;
50      --  Now store OV at OA. Thus, M[SP] <- An
51      CALL store_operand_value
52      CLOCK
53      -- An <- SP
54      register_file_source_x <= RF_X_EA_REG ;
55      register_file_source_y <= RF_Y_FORCE_TO_SP ;
56      alu_mode <= ALU_ADD ;
57      alu_source_a <= ALU_A_PGI ;
```

```
58      alu_source_b <= ALU_B_ADDRESS_Y ;
59      pgi_source <= PGI_ZERO ;
60      reg_update_address_x <= '1' ;
61      CLOCK
62      register_file_source_x <= RF_X_FORCE_TO_SP ;
63      register_file_source_y <= RF_Y_FORCE_TO_SP ;
64      CLOCK
65      -- SP <- SP + d
66      register_file_source_x <= RF_X_FORCE_TO_SP ;
67      register_file_source_y <= RF_Y_FORCE_TO_SP ;
68      alu_mode <= ALU_ADD ;
69      alu_source_a <= ALU_A_ADDRESS_X ;
70      alu_source_b <= ALU_B_IDR ;
71      -- Use IDR for displacement.
72      reg_update_address_x <= '1' ;
73
74      RETURN
75      CLOCK
76
```

I.21. Source code of move_family.sm

```
 1 LABEL move_family
 2
 3      -- Full MOVE: Can move data at any EA to any other EA.
 4
 5      -- What is the size of the operation?
 6      case instruction_register ( 13 downto 12 ) is
 7      when "01" =>    -- Byte mode MOVE
 8                      operation_size_control <= SET_TO_BYTE ;
 9      when "11" =>    -- Word mode MOVE
10                      operation_size_control <= SET_TO_WORD ;
11      when others =>  -- Long mode MOVE
12                      operation_size_control <= SET_TO_DWORD ;
13      end case ;
14
15      -- Decode the source address, storing the data at it in OV.
16
17      CALL decode_ea_and_dereference
18      CLOCK
19
20      -- Then decode the destination address, and store OV at OA.
21      ea_move_destination_control_set <= '1' ;
22
23      -- Update CCRs
24      alu_mode <= ALU_ADD_UPDATE_CCRS ;
25      alu_source_a <= ALU_A_OPERAND_VALUE ;
26      alu_source_b <= ALU_B_PGI ;
27      pgi_source <= PGI_ZERO ;
28
29      JUMP decode_ea_and_store
30      CLOCK
31
```

I.22. Source code of moveq.sm

```
 1 LABEL moveq
 2
 3      -- MOVEQ: Move an immediate (part of the opcode, low byte)
 4      -- to a data register.
 5
 6      -- operation: DRF(B) <- LOW_BYTE_OF_IR
 7      -- Always dword-sized
 8
 9      operation_size_control <= SET_TO_DWORD ;
10      alu_mode <= ALU_ADD ;
11      alu_source_a <= ALU_A_PGI ;
12      pgi_source <= PGI_ZERO ;
13      alu_source_b <= ALU_B_LOW_BYTE_OF_IR ;
14
15      -- Set A=B so that DRF(B) is updated.
```

```
16      register_file_source_x <= RF_X_11_TO_9_FIELD ;
17      reg_update_data_x <= '1' ;
18      RETURN
19      CLOCK
20
```

I.23. Source code of `nop.sm`

```
 1 LABEL nop
 2      -- No-op
 3      RETURN
 4      CLOCK
 5
```

I.24. Source code of `pea.sm`

```
 1 LABEL pea
 2
 3      -- Machine for PEA (Push Effective Address)
 4      -- The effect is:
 5      -- SP <- SP - 4
 6      -- M[SP] <- EA
 7
 8      -- Prepare to decrement the stack pointer
 9      register_file_source_x <= RF_X_FORCE_TO_SP ;
10      operation_size_control <= SET_TO_DWORD ;
11
12      CLOCK
13
14      -- SP <- SP - 4
15      register_file_source_x <= RF_X_FORCE_TO_SP ;
16      alu_mode <= ALU_SUBTRACT ;
17      alu_source_a <= ALU_A_ADDRESS_X ;
18      alu_source_b <= ALU_B_PGI ;
19      pgi_source <= PGI_FOUR ;
20      reg_update_address_x <= '1' ;
21
22      CLOCK
23      register_file_source_x <= RF_X_EA_REG ;
24
25      -- Decode the effective address.
26      -- Note: no need to handle register direct modes.
27      -- They are not supported by PEA.
28      CALL decode_ea
29
30      CLOCK
31
32      -- Move OV <- OA
33      alu_mode <= ALU_ADD ;
34      alu_source_a <= ALU_A_PGI ;
35      alu_source_b <= ALU_B_OA ;
36      pgi_source <= PGI_ZERO ;
37      operand_value_source <= ALU_TO_OV ;
38      register_file_source_y <= RF_Y_FORCE_TO_SP ;
39
40      CLOCK
41
42      -- Move OA <- SP
43      register_file_source_y <= RF_Y_FORCE_TO_SP ;
44      alu_mode <= ALU_ADD ;
45      alu_source_a <= ALU_A_PGI ;
46      alu_source_b <= ALU_B_ADDRESS_Y ;
47      pgi_source <= PGI_ZERO ;
48      operand_address_source <= ALU_TO_OA ;
49
50      -- Store OV at OA: M[OV] <- OA.
51      JUMP store_operand_value
52      CLOCK
53
```

I.25. Source code of `rts.sm`

```
LABEL rts

    -- Machine for RTS (return from subroutine)
    -- The effect is:
    -- PC <- M[SP]
    -- SP <- SP + 4

    -- Prepare to do OA <- SP
    register_file_source_y <= RF_Y_FORCE_TO_SP ;
    operation_size_control <= SET_TO_DWORD ;
    CLOCK

    -- Move OA <- SP
    register_file_source_y <= RF_Y_FORCE_TO_SP ;
    alu_mode <= ALU_ADD ;
    alu_source_a <= ALU_A_PGI ;
    alu_source_b <= ALU_B_ADDRESS_Y ;
    pgi_source <= PGI_ZERO ;
    operand_address_source <= ALU_TO_OA ;

    -- Dereference OA to OV.
    CALL load_operand_value
    CLOCK

    -- PC <- OV
    alu_mode <= ALU_ADD ;
    alu_source_a <= ALU_A_OPERAND_VALUE ;
    alu_source_b <= ALU_B_PGI ;
    pgi_source <= PGI_ZERO ;
    pc_source <= ALU_TO_PC ;

    register_file_source_x <= RF_X_FORCE_TO_SP ;

    CLOCK
    -- SP <- SP + 4
    register_file_source_x <= RF_X_FORCE_TO_SP ;
    alu_mode <= ALU_ADD ;
    alu_source_a <= ALU_A_ADDRESS_X ;
    alu_source_b <= ALU_B_PGI ;
    pgi_source <= PGI_FOUR ;
    reg_update_address_x <= '1' ;

    RETURN
    CLOCK

```

I.26. Source code of `scc.sm`

```
LABEL scc

    -- Machine for S<cc> "Set on condition code"
    -- If the condition code is true then the EA is filled with
    -- ones, otherwise it is filled with zeroes. Size: byte.

    -- Zero the OV register.
    operation_size_control <= SET_TO_BYTE ;
    alu_mode <= ALU_ADD ;
    alu_source_a <= ALU_A_PGI ;
    alu_source_b <= ALU_B_PGI ;
    pgi_source <= PGI_ZERO ;
    operand_value_source <= ALU_TO_OV ;
    CLOCK

    -- So the ALU output will be either -1 (all 1s) or
    -- 0 according to the truth of the condition. Where should the
    -- output be sent?

    case ea_mode is
    when "000" =>   reg_update_data_x <= '1' ;
                    alu_mode <= ALU_SUBTRACT ;
                    alu_source_a <= ALU_A_OPERAND_VALUE ;
```

```
24
25                      if ( condition_true = '1' )
26                      then
27                          alu_source_b <= ALU_B_PGI ;
28                          pgi_source <= PGI_ONE ;
29                      else
30                          alu_source_b <= ALU_B_PGI ;
31                          pgi_source <= PGI_ZERO ;
32                      end if ;
33                      RETURN
34
35      -- Mode 001 is not allowed.
36      when others =>
37                      CALL decode_ea
38
39      end case ;
40      CLOCK
41
42      -- It's a memory address
43      alu_mode <= ALU_SUBTRACT ;
44      alu_source_a <= ALU_A_OPERAND_VALUE ;
45
46      if ( condition_true = '1' )
47      then
48          alu_source_b <= ALU_B_PGI ;
49          pgi_source <= PGI_ONE ;
50      else
51          alu_source_b <= ALU_B_PGI ;
52          pgi_source <= PGI_ZERO ;
53      end if ;
54      operand_value_source <= ALU_TO_OV ;
55
56      JUMP store_operand_value
57      CLOCK
58
```

I.27. Source code of `start.sm`

```
 1 LABEL start
 2
 3 -- Initialise the processor by clearing the PC register.
 4 -- PC <- 0
 5          alu_mode <= ALU_ADD ;
 6          alu_source_a <= ALU_A_PGI ;
 7          alu_source_b <= ALU_B_PGI ;
 8          pgi_source <= PGI_ZERO ;
 9          pc_source <= ALU_TO_PC ;
10
11 -- Now: instruction fetch. Essentially, the following operations
12 -- are done in parallel.
13 -- IR <- [PC].
14 -- PC <- PC + 2
15          JUMP fetch_ir_high
16          CLOCK
17
18 LABEL pause_state
19          if ( run_single_instruction = '1' )
20          and ( button_clock_event = '0' )
21          then
22              -- We are in instruction stepping mode
23              -- and waiting for the step button to be pressed.
24              JUMP pause_state
25          else
26              button_clock_event_clear_2 <= '1' ;
27          end if ;
28          CLOCK
29
30          operation_size_control <= SET_TO_IR ;
31
32          -- Instruction decode!
33          IDECODE
34          CLOCK
35
```

```
36 LABEL fetch_ir_high
37         -- Prepare to fetch 1st byte of next instruction
38         mar_source <= PC_TO_MAR ;
39
40         -- Increment PC to point to 2nd byte
41         alu_mode <= ALU_ADD ;
42         alu_source_a <= ALU_A_PC ;
43         alu_source_b <= ALU_B_PGI ;
44         pgi_source <= PGI_ONE ;
45         pc_source <= ALU_TO_PC ;
46         CLOCK
47
48 LABEL fetch_ir_low
49         -- Store the 1st byte.
50         ir_source <= MDR_TO_IR_1 ;
51
52         -- Prepare to fetch the 2nd byte.
53         mar_source <= PC_TO_MAR ;
54
55         -- Increment PC to point to byte following instruction
56         alu_mode <= ALU_ADD ;
57         alu_source_a <= ALU_A_PC ;
58         alu_source_b <= ALU_B_PGI ;
59         pgi_source <= PGI_ONE ;
60         pc_source <= ALU_TO_PC ;
61
62         -- Extra per-instruction initialisation
63         ea_move_destination_control_clear <= '1' ;
64         restore_pc_after_immediate_fetch_clear <= '1' ;
65
66         CLOCK
67
68         -- Store the 2nd byte
69         ir_source <= MDR_TO_IR_0 ;
70         JUMP pause_state
71         CLOCK
72
73
74 -- The fault state indicates a bug in the microcode or an illegal opcode.
75
76 LABEL fault
77     JUMP fault
78     CLOCK
79
80
```

I.28. Source code of `tst.sm`

```
 1 LABEL tst
 2
 3     -- Machine for TST - which compares an effective address with
 4     -- zero, setting the CCs appropriately.
 5
 6     operation_size_control <= SET_TO_IR ;
 7
 8     -- Operation is EA <op> ZERO
 9     -- ALU_CLR_FAMILY is used as the ALU family because
10     -- it will program the CCRs. It may add or subtract - but it
11     -- doesn't matter which, since operand B is zero.
12
13     case ea_mode is
14     when "000" =>
15                     -- DRF(A) <op> ZERO
16                     alu_mode <= ALU_CLR_FAMILY ;
17                     alu_source_a <= ALU_A_DATA_X ;
18                     alu_source_b <= ALU_B_PGI ;
19                     pgi_source <= PGI_ZERO ;
20                     RETURN
21     -- when "001" =>    not supported
22     when others =>
23                     CALL decode_ea_and_dereference
24     end case ;
25
```

```
26      CLOCK
27
28      -- EA <op> ZERO
29      alu_mode <= ALU_CLR_FAMILY ;
30      alu_source_a <= ALU_A_OPERAND_VALUE ;
31      alu_source_b <= ALU_B_PGI ;
32      pgi_source <= PGI_ZERO ;
33      RETURN
34      CLOCK
35
```

I.29. Source code of `unlk.sm`

```
 1 LABEL unlk
 2      -- This machine is for the UNLK instruction.
 3      -- SP <- ARF_A
 4      -- ARF_A <- M[SP]
 5      -- SP <- SP + 4
 6
 7      operation_size_control <= SET_TO_DWORD ;
 8
 9      CLOCK
10      -- First, do SP <- ARF_A
11      -- Also, OA <- ARF_A
12      register_file_source_x <= RF_X_FORCE_TO_SP ;
13      alu_mode <= ALU_ADD ;
14      alu_source_a <= ALU_A_ADDRESS_X ;
15      alu_source_b <= ALU_B_PGI ;
16      pgi_source <= PGI_ZERO ;
17      reg_update_address_x <= '1' ;
18      operand_address_source <= ALU_TO_OA ;
19
20      -- Now, do ARF_A <- M[SP]. First do OV <- M[OA]
21      CALL load_operand_value
22      CLOCK
23      -- Then do ARF_A <- OV
24
25      register_file_source_x <= RF_X_EA_REG ;
26      register_file_source_y <= RF_Y_FORCE_TO_SP ;
27      alu_mode <= ALU_ADD ;
28      alu_source_a <= ALU_A_OPERAND_VALUE ;
29      alu_source_b <= ALU_B_PGI ;
30      pgi_source <= PGI_ZERO ;
31      reg_update_address_x <= '1' ;
32
33      CLOCK
34      -- Now SP <- SP + 4.
35      register_file_source_x <= RF_X_FORCE_TO_SP ;
36      register_file_source_y <= RF_Y_FORCE_TO_SP ;
37      alu_mode <= ALU_ADD ;
38      alu_source_a <= ALU_A_PGI ;
39      alu_source_b <= ALU_B_ADDRESS_Y ;
40      pgi_source <= PGI_FOUR ;
41      reg_update_address_x <= '1' ;
42
43      RETURN
44      CLOCK
45
```