

# Implementing Absolute Addressing in a Motorola 68000 Processor (DRAFT)

Dylan Leigh (s3017239)

2009

This project involved the further development of a limited 68000 processor core, developed by Dylan Leigh for the subject Advanced Digital Design 1 (EEET2192) during semester 1 2008.

The CPU core originally supported only immediate and register addressing, and used one read state for each data read. This project aims to extend the system by looping the read states, allowing for absolute and more advanced addressing modes.

This document describes the CPU features and usage, including the operation of the CPU on the DE1 development board, as well as the process of implementing the new addressing mode. The development of the initial CPU design is not covered in detail.

# Contents

<b>I</b>	<b>Features and Usage</b>	<b>4</b>
<b>1</b>	<b>Features</b>	<b>4</b>
1.1	Current Capabilities . . . . .	4
1.2	Development Tools . . . . .	4
<b>2</b>	<b>CPU Instruction Set</b>	<b>5</b>
2.1	Move . . . . .	5
2.2	Branch . . . . .	5
2.3	Add . . . . .	5
2.4	And . . . . .	5
2.5	No-op . . . . .	5
<b>3</b>	<b>Interface with DE1 Board</b>	<b>6</b>
3.1	Overview . . . . .	6
3.1.1	Structural Design Notes . . . . .	6
3.1.2	Pin Assignments . . . . .	6
3.2	Using the Interface . . . . .	6
3.2.1	Outputs . . . . .	7
<b>II</b>	<b>Development</b>	<b>8</b>
<b>4</b>	<b>Design</b>	<b>8</b>
4.1	Early Component Design . . . . .	8
4.2	Final Component Design . . . . .	8
4.3	Initial Fetch-Decode-Execute Cycle States . . . . .	9
4.4	Instruction Decoder . . . . .	9
4.5	MMU and Buses . . . . .	9
4.6	Design for Absolute Addressing . . . . .	10
4.6.1	Final Fetch-Decode-Execute Cycle States . . . . .	10
<b>5</b>	<b>Implementation Notes</b>	<b>11</b>
5.1	Debugging . . . . .	11
<b>6</b>	<b>Testing</b>	<b>12</b>
6.1	Sample Programs . . . . .	12
6.1.1	Move/Add/Branch Test . . . . .	12
6.1.2	Move Flag and Branch Test . . . . .	12
6.1.3	Move/And/Branch Test . . . . .	12
6.1.4	Absolute Addressing Test . . . . .	12
6.2	Simulation Waveforms . . . . .	13
6.2.1	Move/Add/Branch Test . . . . .	13
6.2.2	Move Flag and Branch Test . . . . .	14
6.2.3	Move/And/Branch Test . . . . .	15
6.2.4	Absolute Addressing Test . . . . .	16

<b>7 Future Development</b>	<b>17</b>
7.1 Hardware RAM . . . . .	17
7.2 Instruction Set . . . . .	17
7.3 Extended addressing modes . . . . .	17
7.4 Parameterizing memory access . . . . .	17
7.5 System Mode instructions . . . . .	17
<b>III Appendices</b>	<b>18</b>
<b>A Appendix A: CPU Source Code</b>	<b>18</b>
A.1 m68k_cpu_core.vhd . . . . .	18
<b>B Appendix B: DE1 Interface Source Code</b>	<b>32</b>
B.1 m68k_de1.vhd . . . . .	32
B.2 hex7seg.vhd . . . . .	37
B.3 clockdiv.vhd . . . . .	38
<b>C Appendix C: Test MMU Files</b>	<b>40</b>
C.1 m68k_fakemmu_1.vhd . . . . .	40
C.2 m68k_fakemmu_2.vhd . . . . .	43
C.3 m68k_fakemmu_3.vhd . . . . .	45
C.4 m68k_fakemmu_4.vhd . . . . .	48
<b>D Appendix D: Makefile</b>	<b>51</b>
<b>E Appendix E: Timing and Performance</b>	<b>52</b>
E.1 Timing . . . . .	52
E.2 Logic Elements Used . . . . .	52
E.3 CPU Performance . . . . .	52

## Part I

# Features and Usage

## 1 Features

### 1.1 Current Capabilities

The CPU described here implements 5 instructions from the Motorola 68000 instruction set:

- Move (MOVE),
- Branch Always (BRA),
- Add (ADD),
- Logical And (AND),
- No Operation (NOP).

The implementation supports data register, address register, immediate and absolute addressing for these instructions. The CPU is completely orthogonal; all operations can be used with all addressing modes<sup>1</sup>.

8 Data registers and 8 Address registers can be read from and written to. Although no currently implemented instructions read them the 5 user mode flags - Carry (C), Overflow (O), Negative (N), Zero (Z) and Sign Extend (X) - are implemented and are set by relevant operations.

This implementation has an added CPU panic feature. On any invalid, illegal or unimplemented instruction or addressing mode, the CPU will set the “panic” line high and halt until the CPU has been manually reset. In simulation an assertion will be raised describing the error.

A DE1 interface entity<sup>2</sup> which allows demonstration and examination of the CPU on the DE1 boards is also provided. This board uses the Cyclone II EP2C20F484C7 FPGA device.

### 1.2 Development Tools

The open source **GHDL** compiler (<http://ghdl.free.fr/>) was used for most of the EEET2192 development and the initial testing. Once all the operations had been tested successfully in simulation the system was synthesized using Quartus and tested on real hardware. The new version with absolute addressing has been developed using GHDL only and has not been tested in hardware.

The GHDL development flow is similar to using GCC or similar command line compilers. A VHDL file is analyzed to produce an object file based on an entity. This can then be linked with other object files - including, optionally, object files compiled from other sources, such as C or C++ code - to produce a native executable binary. The binary is then run to simulate the system. Assertions and reports print data to the console, and the program can optionally write a VCD or GHW waveform file.

**GTKWave** (<http://gtkwave.sourceforge.net/>) was used to view the output waveforms from the simulations, and to produce the waveform diagrams used in this document.

GNU or BSD **Make** (<http://www.gnu.org/software/make/>) can be used to automate this build and execution process. A makefile is provided in the Appendix<sup>3</sup> which builds all code and runs the three sample 68000 programs - simply executing “make” at the command line will suffice. The entire build process takes less than a second to build the CPU with GHDL on a typical desktop system, whereas analysis and synthesis under Quartus takes approximately 25 seconds, and the full build process can take minutes.

---

<sup>1</sup>Read only locations and modes (such as immediate addressing) cannot be used for the destination.

<sup>2</sup>Refer to Section 3 on page 6.

<sup>3</sup>Section D on page 51.

## 2 CPU Instruction Set

For more information on each instruction refer to *The M68000 Programmer's Reference Manual, 5th ed*, 1979-1986 Motorola Inc., ISBN 0-13-541475-X. A complete discussion of each operation is beyond the scope of this report.

This section describes what each instruction does in this implementation; the full 68000 implements more instructions and addressing modes.

Note that due to limitations of the “Fake MMU” simulated RAM entities used for testing, the “RAM” cannot be written to, although the CPU design itself supports writes to RAM.

### 2.1 Move

- Identified by “0001”, “0010” or “0011” at the start of the instruction, depending on data size.
- Copies 8, 16 or 32 bits of data from a register, memory location or a constant to a register.
- Sets Zero and Negative flags if the data copied is zero or negative.

### 2.2 Branch

- Identified by “0110” at the start of the instruction.
- Performs an 8 or 16 bit signed addition on the program counter. For 8 bit data the offset is encoded in the instruction itself; 16 bit data is read in a similar manner to immediate addressing.
- Changes no flags.

### 2.3 Add

- Identified by “1101” at the start of the instruction.
- Performs an addition of 8+8, 16+16 or 32+32 bits of data. One of the operands (which the data is saved in) must be a data register, but the other can be a memory location, immediate data or an address or data register.
- Sets Zero and Negative flags if the data copied is zero or negative.
- Note that this operation should set the Carry and Overflow flags, however this requires a proper ALU component to be implemented which could not be completed due to time constraints. See the “Planned Features not Fully Implemented” section<sup>4</sup>.

### 2.4 And

- Identified by “1100” at the start of the instruction.
- Performs a logical and of 8, 16 or 32 bits of data. One of the operands (which the data is saved in) must be a data register, but the other can be a memory location, immediate data or an address or data register.
- Sets Zero and Negative flags if the data copied is zero or negative.

### 2.5 No-op

- Identified by “0100111001110001” as the instruction.
- Performs no changes (other than changing the PC during the fetch cycle).
- Changes no flags.

---

<sup>4</sup>Section ?? on page ??.

## 3 Interface with DE1 Board

The source code for this interface is in the Appendix, section [B.1](#) on page [32](#).

### 3.1 Overview

The DE1 interface component allows examination and testing of the CPU implementation on the DE1 board and includes the following facilities:

- “MMU” with 4 sample programs (see [Testing<sup>5</sup>](#)).
- Display of flags, lower byte of address and data buses and CPU/MMU outputs including MMU requests and the panic line.
- Manually operated clock and free running clock with selectable speeds.

#### 3.1.1 Structural Design Notes

The interface file includes the code for links between the CPU outputs and the board lights, and the CPU itself is a component of the interface. The interface also contains the behavior of the MMU itself, a nested set of case statements which put different values on the data bus depending on the address bus and the program selected.

The hex display logic was previously used in the EEET2192 laboratories and is included verbatim. The source can be found in the [Appendix<sup>6</sup>](#).

The adjustable divider used to change the speed of the clock was modified from code used in the EEET2192 laboratories, which was itself a modified version of the behavioral counter code from the lab 2 specification. Source for this file is also located in the [Appendix<sup>7</sup>](#).

#### 3.1.2 Pin Assignments

As this component is not intended for use with other boards, to simplify assignments the names of board pins were used directly as port names. The standard DE1 pin assignments were used.

## 3.2 Using the Interface

Initialization:

- Set switch 0 off to disable the free running clock.
- Choose the test program with switches 9 and 8:
  - 00: Test move, add and branch.
  - 01: Test move flag setting and branch.
  - 10: Test move, and and branch.
  - 11: Test absolute addressing.
- Reset the CPU with key 3.

At this point the CPU can be traced using the manual clock, key 0. To use the free running clock:

- Choose speed with switches 3 to 1.
- Activate the clock with switch 0.
- The clock speed can be changed and the clock can be stopped and started at any time.

---

<sup>5</sup>Section [6](#) on page [12](#).

<sup>6</sup>Section [B.2](#) on page [37](#).

<sup>7</sup>Section [B.3](#) on page [38](#).

### 3.2.1 Outputs

The following outputs are displayed on the DE1 board LEDs:

- Hex digits 2 and 3 show the lower byte of the address bus. This is usually the location of the last memory access, or the PC minus 2.
- Hex digits 0 and 1 show the lower byte of the data bus. Note that the data bus is cleared after a request has been fulfilled, so data may only show up here for one or two clock cycles.
- Red LED 9 hows the panic line. If this is lit the CPU has halted and must be restarted with the reset (key 3).
- Red LED 4 shows the clock signal.
- Red LED 1 indicates the bus read-write line is high. This may appear on initialization or briefly on reset, but should not be lit for any sample programs.
- Red LED 0 indicates a bus request (i.e. the CPU has requested the MMU put the data at the address bus location on the data bus).
- Green LED 7 is lit by the MMU when the bus request has been fulfilled.
- Green LEDs 6 and 5 show the bus data size. For all sample programs word or byte data is used; as immediate byte data is packed into a word all requests from the sample programs will be word length - "10".
- Green LEDs 4 to 0 show the CPU flags:
  - 4 is X (sign extend) (never lit by implemented operations)
  - 3 is N (negative) (lit by some operations)
  - 2 is Z (zero) (lit by some operations)
  - 1 is V (overflow) (never lit by implemented operations)
  - 0 is C (carry) (never lit by implemented operations)

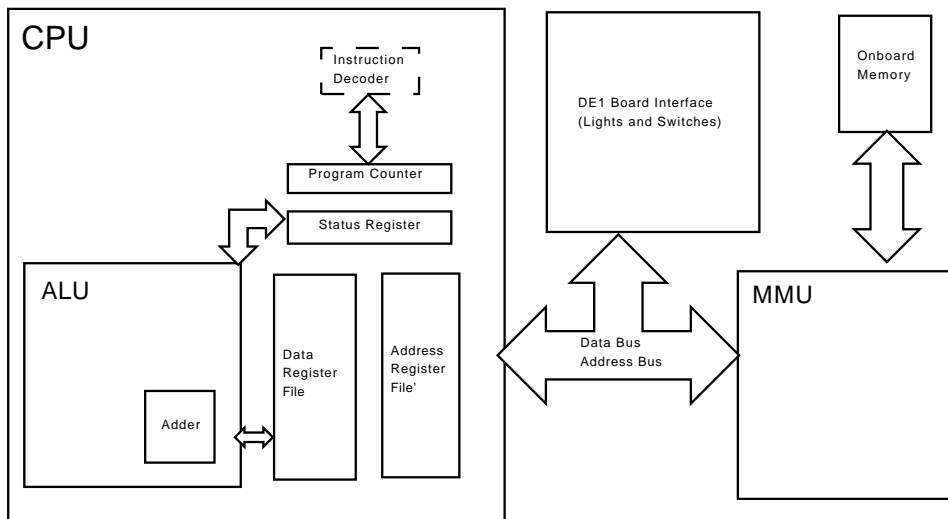
## Part II

# Development

## 4 Design

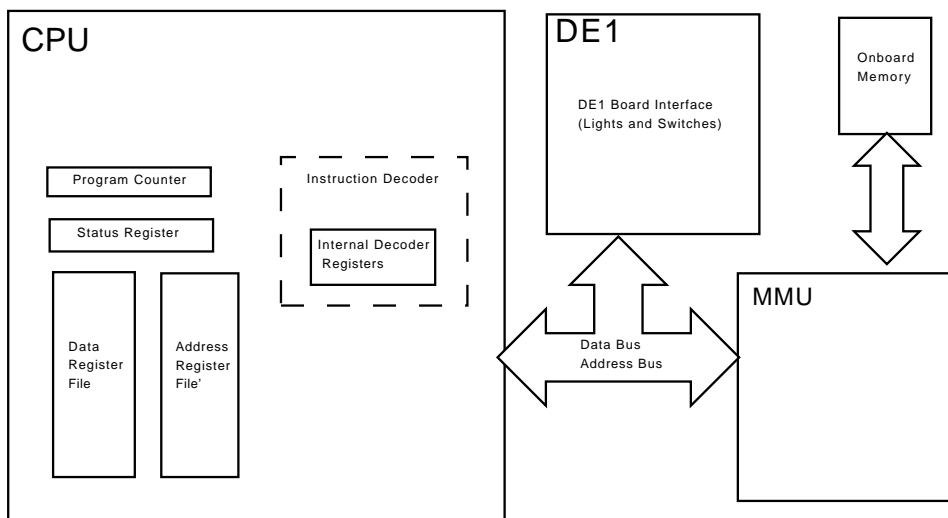
### 4.1 Early Component Design

This is the initial design from the EEET2192 project.



### 4.2 Final Component Design

This represents the final design from the EEET2192 project and this project. Not all ports or registers are shown explicitly on this diagram - for example, the buses include several control lines used to signal requests and acknowledgments.



Instead of using a separate ALU entity, all arithmetic operations are performed using VHDL operators in the CPU entity, simplifying the implementation. The synthesizer made use of the dedicated logic elements on the FPGA chip for these instructions.

The onboard memory access was not implemented and instead four “Fake MMU” entities<sup>8</sup> simulate memory reads for four different sample programs. These can be selected from the DE1 interface. In simulation these entities will raise a warning if there is an attempt to read from a location that should not be required by the program.

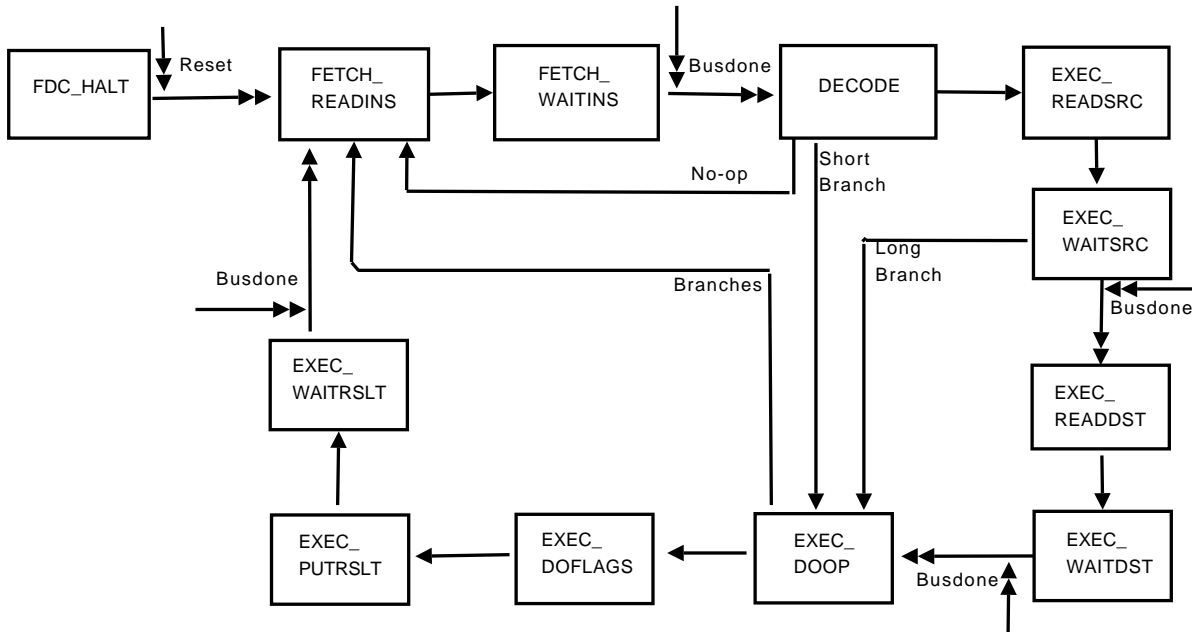
<sup>8</sup>Source code for these entities is in section C on page 40.



### 4.3 Initial Fetch-Decode-Execute Cycle States

This diagram shows the CPU's fetch-decode-execute cycle from the EEET2192 project, when the only addressing modes implemented were immediate and register.

The labelled double arrows with incoming double arrow indicate a signal which must be received for that state to advance. The plain labelled arrows indicate the state flow for a particular operation (which does not follow the common state flow at that point).



Note that moving to the FDC\_HALT state is not shown. Any illegal operations (including trying to run unimplemented operations or operations with unimplemented addressing modes) result in the CPU panic line being set high and the CPU changing to the FDC\_HALT state until it is reset manually.

The final CPU state cycle design is provided in section 4.6.1 on the following page.

### 4.4 Instruction Decoder

The instruction decoder is wholly contained within the CPU code. Due to the tight integration with the CPU no attempt was made to implement it as a separate component.

The decoder determines the current operation, and saves it to a state register. Any data sizes and addressing modes are determined and saved in separate CPU-internal registers, in a format common to most 68000 operations. These registers are read in later execute cycle states to allow common data fetching and saving procedures to be performed using operation-agnostic code.

### 4.5 MMU and Buses

Initially the implementation was intended to use real hardware RAM, which would be initialized with a program and which could be written to (see the “Early Component Design“ above<sup>9</sup>).

Due to lack of time a real MMU component could not be completed, and the implementation instead has read only “memory”, which is emulated by a fake MMU which uses a case statement on the address bus to put different values on the data bus. 4 sample programs were written and assembled which test various CPU instructions and addressing modes. These may be selected in the DE1 board interface<sup>10</sup> or run in simulation<sup>11</sup>.

<sup>9</sup>Section 4.1 on the previous page.

<sup>10</sup>Section 3 on page 6.

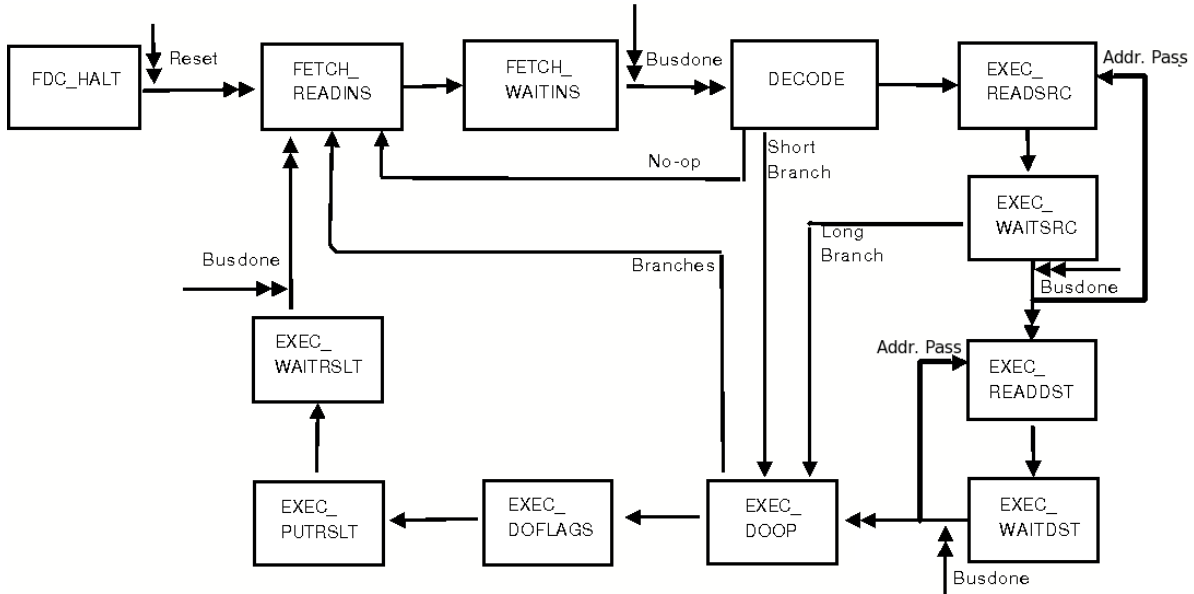
<sup>11</sup>Section 6 on page 12.

## 4.6 Design for Absolute Addressing

For absolute addressing memory reads, it is necessary for the CPU to make two reads from memory - one to get the address, and another to get the data from that address.

The original CPU state cycle design allowed for only one read and wait. The new design “loops” these states, using a new register to keep track of how many passes have been completed. This design supports up to three passes, which will be sufficient to allow all addressing modes on the 68000 to be implemented.

### 4.6.1 Final Fetch-Decode-Execute Cycle States

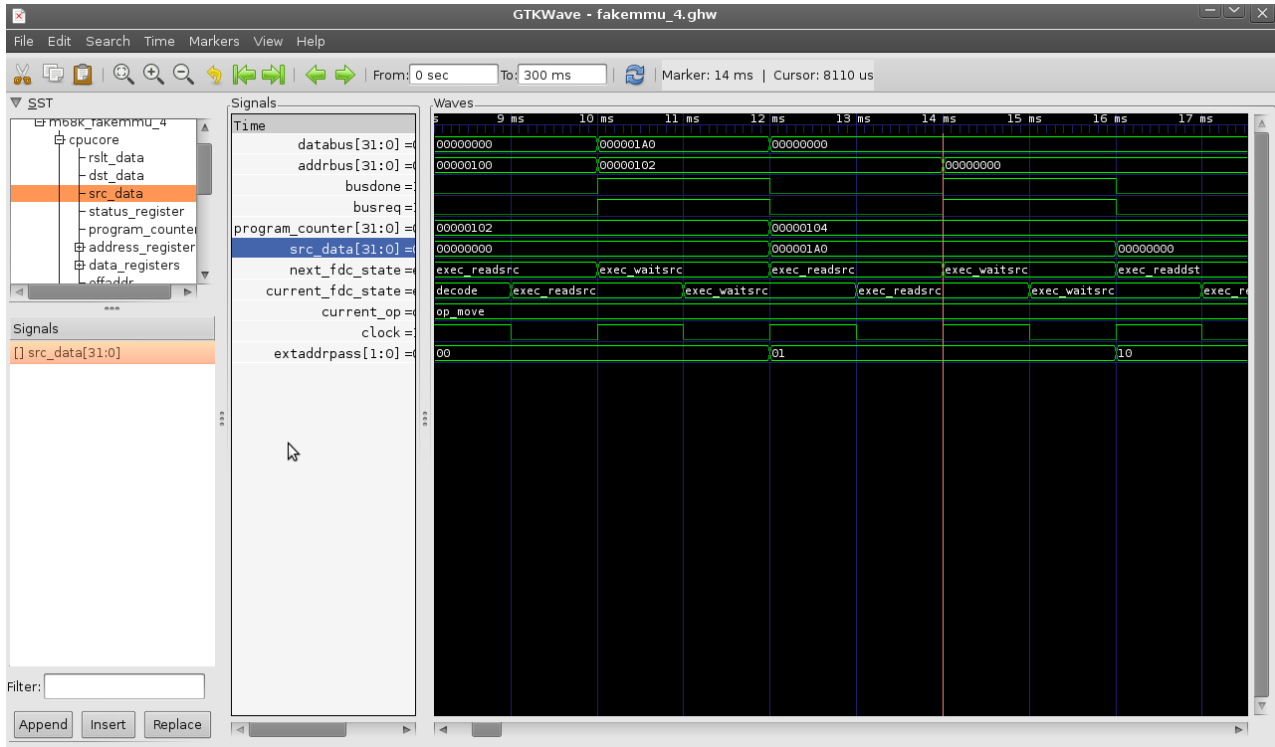


## 5 Implementation Notes

Implementation was straightforward, with a new test program developed<sup>12</sup> to test the new addressing mode. The new CPU core has not yet been tested in hardware.

### 5.1 Debugging

The GTKWave software was used extensively for debugging and testing. An example is shown below:



One error found late during development is shown above. Sample program 4<sup>13</sup> was being run to test absolute addressing mode, and Fake MMU 4 was warning that at 14ms, a memory request was made for an unhandled location (i.e. one which the program should not have accessed during normal operation).

As can be seen in the screenshot, the CPU was requesting a read of location \$0, instead of the data at \$140. Tracing through the CPU code it was found that the address for the second memory request was being taken directly from the data bus, instead of the SRC\_DATA internal register. The \$140 had already been received, acknowledged and placed in the SRC\_DATA register. Transferring SRC\_DATA to the address bus during the second EXEC\_READSRC pass fixed the error.

<sup>12</sup>See section 6.1.4 on the following page.

<sup>13</sup>Assembly source code for the program is in section 6.1.4 on the next page; the corresponding MMU source code is in section C.4 on page 48.

## 6 Testing

Most parts of the system were developed using the open source GHDL compiler (see Design Tools<sup>14</sup>), which builds test-benches as native binary applications. These applications will simulate the system and output relevant assertions and reports. They can also generate waveform files.

Four sample assembly programs were written which use various features of the implemented CPU. These were encapsulated within “fake” MMU programs which set the data bus to the machine code values of the program when the CPU requested them. These programs also reset the system when starting and provided a clock signal to operate the CPU.

The VHDL source for the MMU programs is in the Appendix<sup>15</sup>; the assembly programs themselves are provided below.

### 6.1 Sample Programs

Annotated simulation waveforms for these programs can be found in section 6.2 on the next page.

#### 6.1.1 Move/Add/Branch Test

```
org $100
move.w #02, d1
move.w #03, d2
add.w d2, d1
add.w #5, d0 ; note d0 will continue increasing while program loops
bra $100
end
```

#### 6.1.2 Move Flag and Branch Test

```
org $100
move.w #0, d1 ; sets zero flag
move.w #BEEF, d2 ; sets negative flag
bra $100
end
```

#### 6.1.3 Move/And/Branch Test

```
org $100
move.w #$AAAA, d1
move.w #$5555, d2
and.b d2,d1 ; sets zero flag
and.w $#FFFF, d0
bra $100
end
```

#### 6.1.4 Absolute Addressing Test

```
org $100
move.w $10A, d1
move.w $1BB, d2
bra $100
org $10A
dw $1111
org $1BB
dw $2222
end
```

---

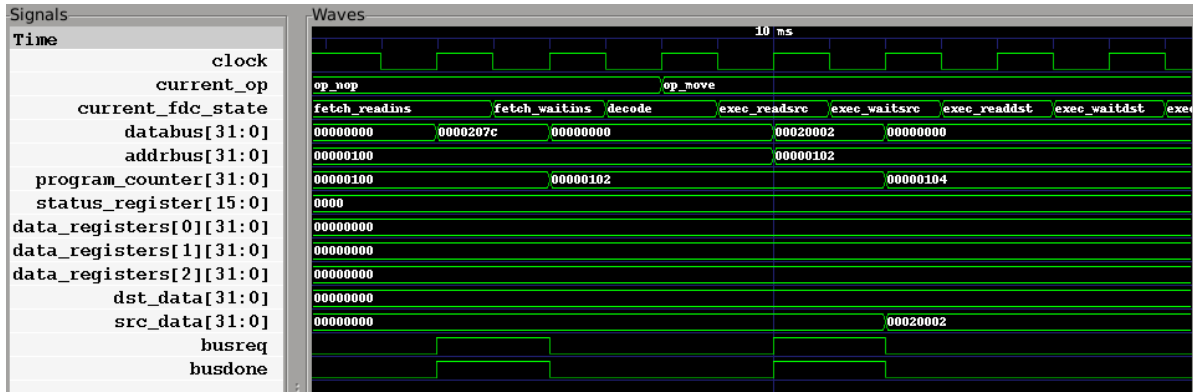
<sup>14</sup>Section 1.2 on page 4.

<sup>15</sup>Section C on page 40.

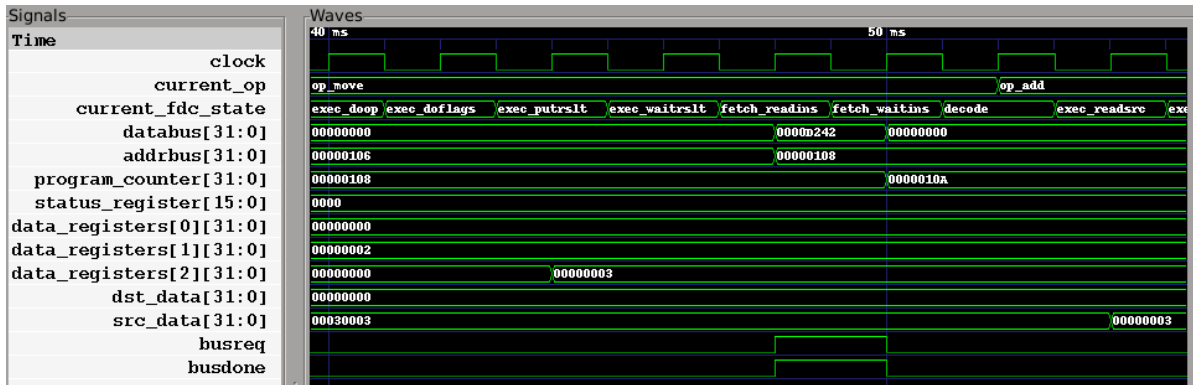
## 6.2 Simulation Waveforms

These images were produced using GTKWave (see Tools, section 1.2 on page 4). A full set of waveforms would be too large to include in this report. A selected number which show state and register transitions have been included. The source code for the assembly programs running on the CPU during these tests is in section 6.1 on the previous page.

### 6.2.1 Move/Add/Branch Test



This is just after reset (note the current\_op is no-op until the first instruction is decoded). The instruction is fetched and the program counter incremented by 2, and then immediate data is fetched from the PC location (and the PC is incremented again). Note that the destination is a data register so there are no memory requests and no activity during the destination read states.



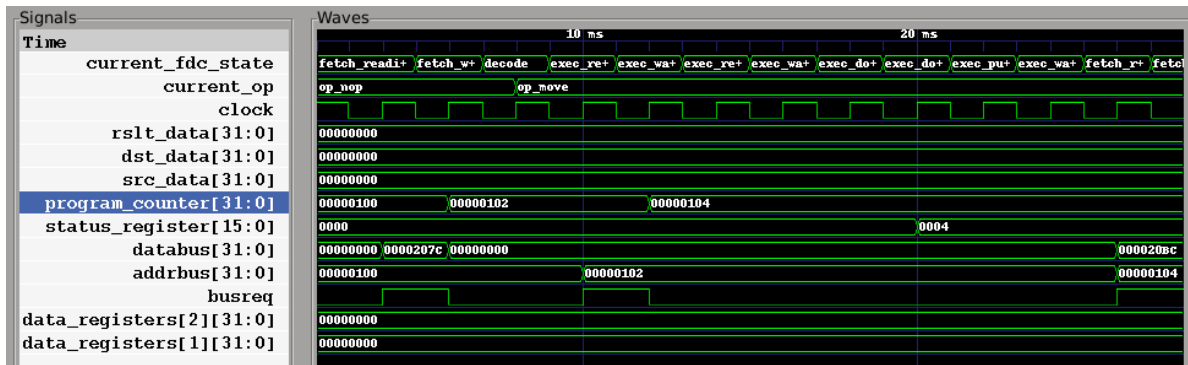
This screenshot shows the end of the move instruction. The data is written to the destination (in this case, data register 2) during the EXEC\_PUTRSLT cycle. The next instruction is an add and we can see it decoded here. The operands are registers so we do not see any memory accesses until the source data is placed into the CPU internal src\_data register.



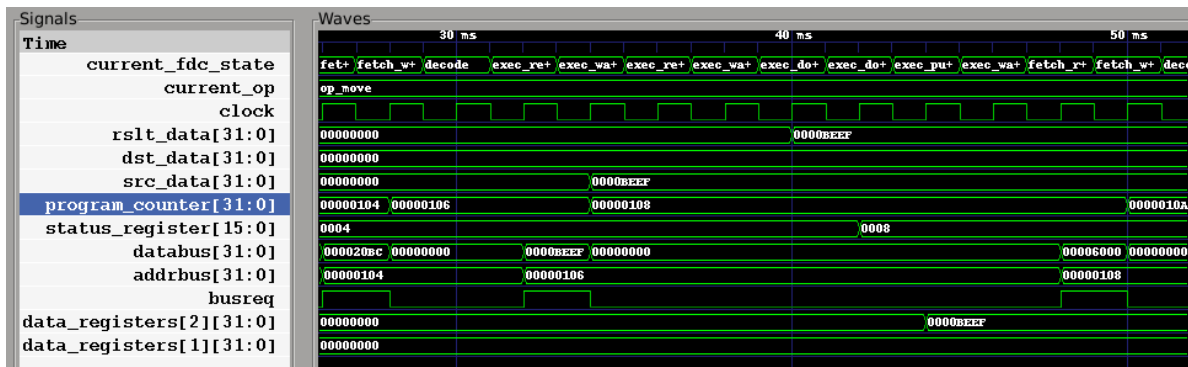
This screenshot shows the execution of the add instruction. The source and destination - both data registers - are read into the src\_data and dst\_data internal registers. During the EXEC\_DOOP cycle these are added and

the result is placed in the internal rslt\_data register. This is saved to the destination during the EXEC\_PUTRSLT cycle.

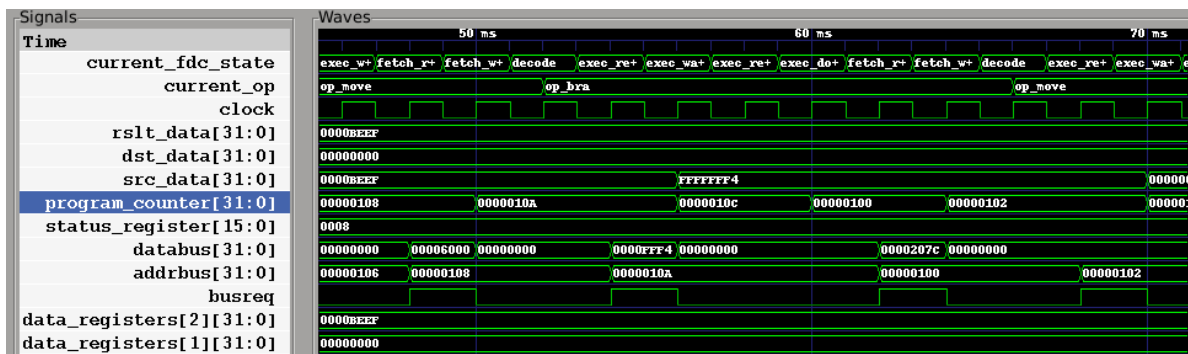
### 6.2.2 Move Flag and Branch Test



This screenshot shows the first instruction, which moves the immediate value 0 to d1. As the value is zero, the initial value of the data bus, we cannot see it being put on the data bus (although the PC is incremented when the immediate data is read). As d1 is also already zero, we do not see this change. However, during the `EXEC_DOFLAGS` cycle we can see bit 3 of the status register is set, indicating the last instruction generated a zero result.

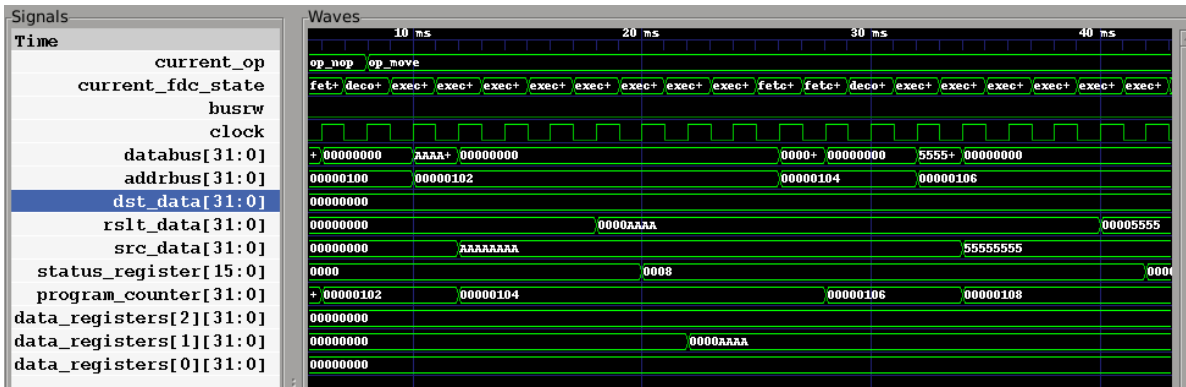


The second move, `$BEEF` to d2, is visible here. We can see the instruction being read, decoded, the immediate value being read and placed in the `src_data` internal register, and then in the `rslt_data` register. The status register changes from 4 to 8, indicating that now the negative flag is set - as a 16 bit value `$BEEF` is a negative number.

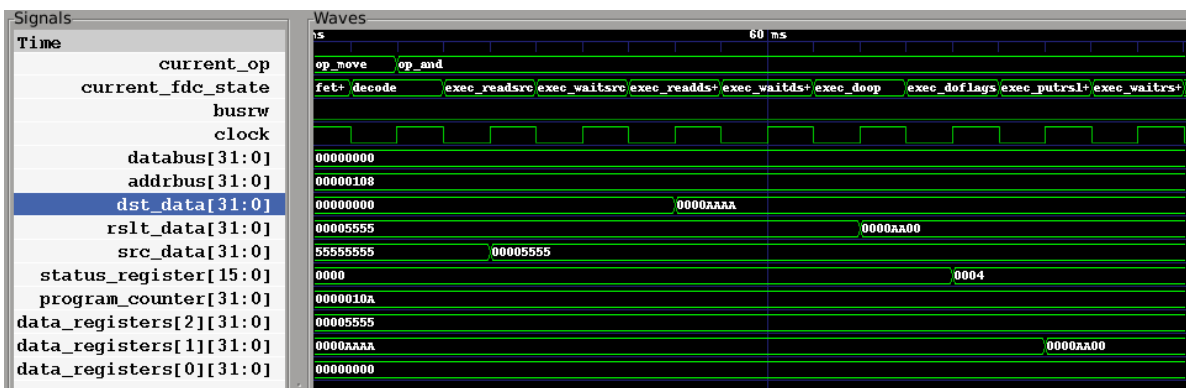


The full set of cycles for a long branch instruction can be seen here. After the instruction is decoded the next word is read (`$FFF4` or -12 decimal). This is sign extended to 32 bits (the `FFFFFFF4` in `src_data`) and added to the program counter in the `EXEC_DOOP` state, returning it to the start of the program (`$100`). We can see that in a branch the next instruction starts immediately - state after `EXEC_DOOP` is `FETCH_READINS`, normally it would be `EXEC_PUTRSLT`.

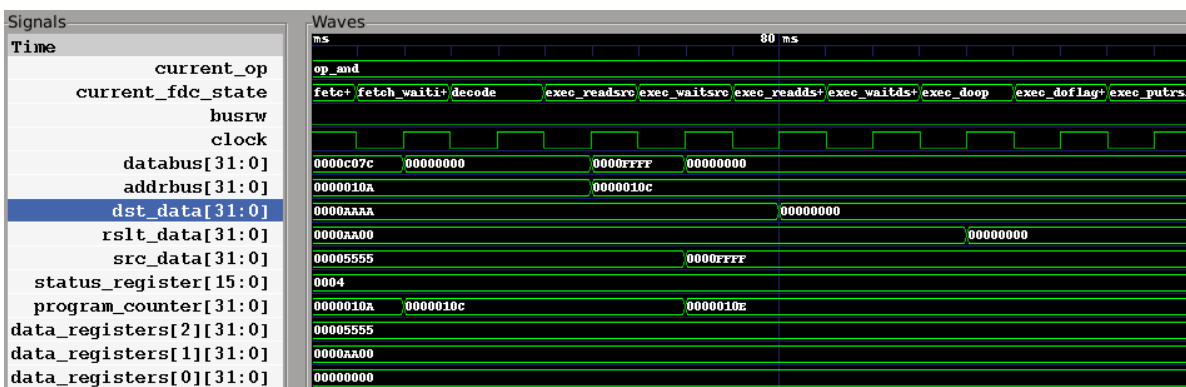
### 6.2.3 Move/And/Branch Test



This screenshot shows the first two instructions, which move \$AAAA and \$FFFF into d1 and d2.

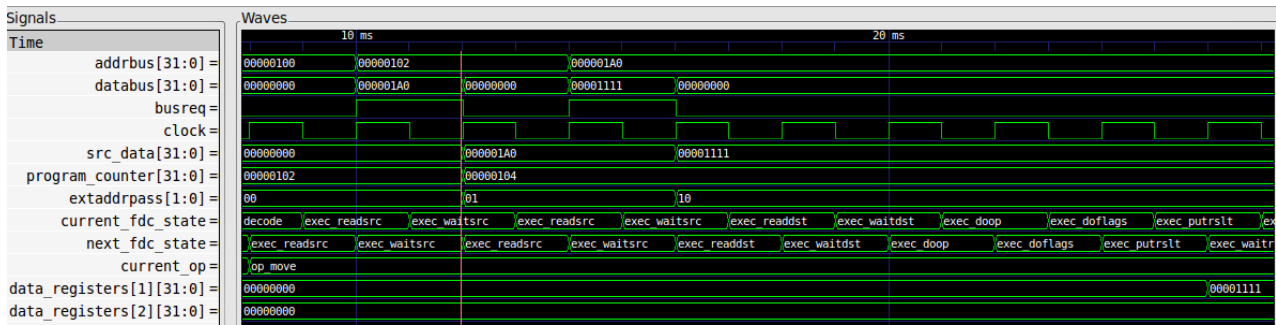


The next instruction is a *byte* size and operation on d1 and d2 - and 1010 with 0101. The lower byte is set to 0 and the zero flag is set.

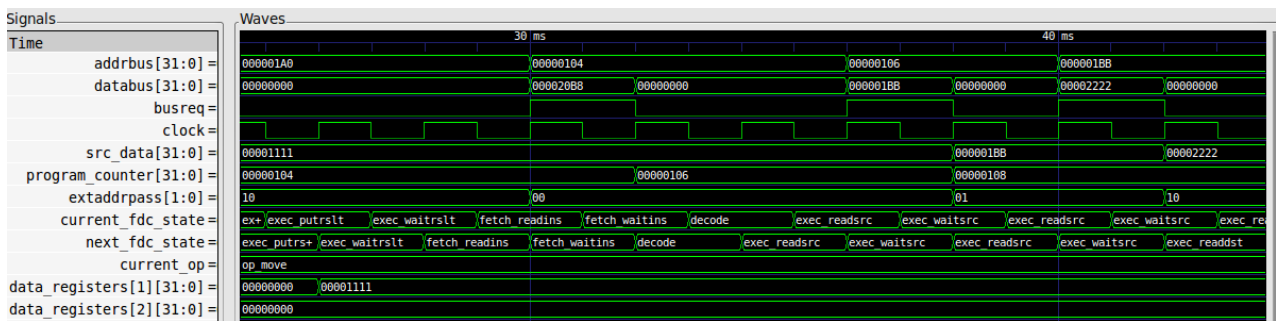


The next instruction is an and with immediate data (\$FFFF) and d0 (0), again producing zero.

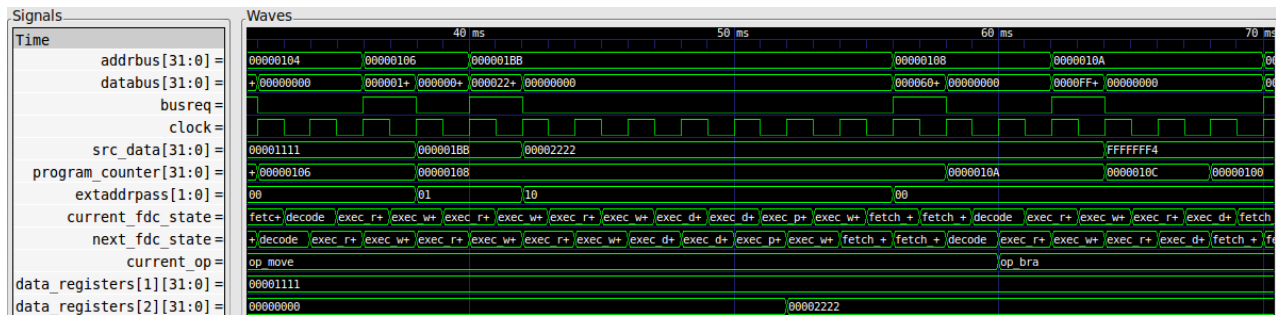
### 6.2.4 Absolute Addressing Test



This screenshot shows the source read states (EXEC\_READSRC and EXEC\_WAITSRC) looping to fetch the address of the data from \$102 (pass 1 - EXTADDRPASS changes from 00 to 01) and then again to fetch the data from that address (at \$1A0 - EXTADDRPASS changes from 01 to 10).



The data from \$1A0 is put into d1 during the EXEC\_PUTRSLT state. When the CPU cycle restarts, EXTADDRPASS is reset to 00 (during the FETCH\_READINS state). To the left of this screenshot we can see the start of the next move instruction where EXTADDRPASS is again used in a pair of EXEC\_READSRC and EXEC\_WAITSRC states. The address \$1BB is read from location \$106 (within the program code) and the data \$2222 is read from address \$1BB.



At the end of the second move instruction \$2222 is loaded into data register 2. At the end of this diagram the branch instruction is executing and the PC returns to the start of the program (location \$100).



## 7 Future Development

### 7.1 Hardware RAM

### 7.2 Instruction Set

### 7.3 Extended addressing modes

### 7.4 Parameterizing memory access

It may be advantageous to replace the memory access states (see the state diagram <sup>16</sup>) with a single pair of access-wait states which are parametrized depending on the addressing mode. This would allow the same code to be used for immediate and absolute addressing for source and destination, as well as more advanced modes such as register indirect.

### 7.5 System Mode instructions

---

<sup>16</sup>Section 4.3 on page 9.

## Part III

## Appendices

## A Appendix A: CPU Source Code

## A.1 m68k\_cpu\_core.vhd

```

1  -- CPU core for 68k
2  -- Dylan Leigh s3017239
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity m68k_cpu_core is
9      port (
10         reset: in std_logic; -- active high
11         clock: in std_logic;
12
13         panic: out std_logic; -- set high when CPU panics and halts
14             -- nothing changes until manually reset
15
16         -- bus controls
17         busrw: out std_logic; -- bus read/write: zero is write
18         busreq: out std_logic; -- set to 1 to make a request from the MMU
19         busdone: in std_logic; -- is set to 1 when request fulfilled by the MMU
20         busdatasize: out std_logic_vector(1 downto 0);
21             -- 68k convention: 01: byte (8), 10: word(16), 11: long (32)
22
23         -- address bus
24         addrbus: out std_logic_vector(31 downto 0);
25         -- data bus
26         databus: in std_logic_vector(31 downto 0);
27         -- memory writes TODO databus: inout std_logic_vector(31 downto 0)
28
29         -- for debugging
30         flags: out std_logic_vector(4 downto 0)
31     );
32 end m68k_cpu_core;
33
34 architecture behaviour of m68k_cpu_core is
35     type FDC_State_Type is (FETCH_READINS, FETCH_WAITINS,
36                             DECODE,
37                             -- note source or destination may be immediate data,
38                             -- which still has to be read from memory (and
39                             -- waited on)
40                             EXEC_READSRC, EXEC_WAITSRC, -- EXEC_PUTSRCALU,
41                             EXEC_READDST, EXEC_WAITDST, -- EXEC_PUTDSTALU,
42                             EXEC_DOOP, EXEC_DOFLAGS,
43                             EXEC_PUTRSLT, EXEC_WAITRSLT,
44                             FDC_HALT);
45     signal current_fdc_state, next_fdc_state : FDC_State_Type;
46
47     -- Note we are using "instruction" to refer to the whole thing
48     -- including operands and "opcode" to refer to the opcode part.
49     -- "Operation" is what the opcode represents
50     -- internal use only

```

```

type CPUOp is (OP_MOVE, OP_ADD, OP_AND, OP_BRA, OP_NOP);
53 signal current_op: CPUOp;
signal current_inst: std_logic_vector(15 downto 0);
55
signal opdatasize: std_logic_vector(1 downto 0);
57
-- effective address codes, either copied directly from the instruction or
59 -- determined from it during decode
-- internal use only
61 signal src_addrcode: std_logic_vector(5 downto 0);
signal dst_addrcode: std_logic_vector(5 downto 0);
63 -- internal use only -- set for the second memory access
-- pass, for extended addressing TODO
65 signal extaddrpass: std_logic_vector(1 downto 0);
-- internal use only -- effective address location
67 signal effaddr: std_logic_vector(31 downto 0);

69 -- programmer registers
-- TODO: these should be replaced later with clocked registers
71 type RegisterSet is array (0 to 7) of std_logic_vector(31 downto 0);
signal data_registers: RegisterSet;
73 signal address_registers: RegisterSet;
signal program_counter: std_logic_vector(31 downto 0);
75 signal status_register: std_logic_vector(15 downto 0);

77 -- internal registers
signal src_data: std_logic_vector(31 downto 0);
79 signal dst_data: std_logic_vector(31 downto 0);
signal rslt_data: std_logic_vector(31 downto 0);
81
begin
83 -- TODO component registers

85 flags <= status_register(4 downto 0);

87 fdc_activity: process (reset, clock) -- busdone for wait states TODO
begin
89   if reset = '1' then
-- XXX: SET INTER-CYCLE SIGNALS AND ALL OUTPUTS TO KNOWN
91     -- INITIAL VALUES HERE
program_counter <= x"00000100";
93     status_register <= x"0000";
next_fdc_state <= FETCH_READINS;

95
-- Starting from fetch state the instruction decoding signals
97 -- should not affect anything but are included here anyway
current_op <= OP_NOP;
99     current_inst <= x"0000";
opdatasize <= "10"; -- word
101     src_addrcode <= "000000"; -- d0
dst_addrcode <= "000000"; -- d0
103     extaddrpass <= "00";
src_data <= x"00000000";
105     dst_data <= x"00000000";
rslt_data <= x"00000000";

107
for i in 0 to 7
109   loop
data_registers(i) <= x"00000000";
111   end loop;

```

```

113     addrbus <= x"00000100";
114     -- TODO no mem writes databus <= x"00000000";
115     busreq <= '0';
116     busrw <= '0';
117     busdatasize <= "00";
118     panic <= '0';
119
120 else -- set next state based on current state
121     -- start huge current state statement of doom
122     if rising_edge(clock) then
123         case current_fdc_state is
124
125             when FETCH_READINS =>
126                 addrbus <= program_counter;
127                 busrw <= '0'; --read
128                 busdatasize <= "10"; --word size
129                 busreq <= '1'; --request
130                 next_fdc_state <= FETCH_WAITINS;
131                 extaddrpass <= "00";
132
133             when FETCH_WAITINS =>
134                 if busdone = '1' then
135                     program_counter <= std_logic_vector(
136                         unsigned(program_counter) + 2);
137                     busreq <= '0'; --end request
138                     current_inst <= databus(15 downto 0);
139                     next_fdc_state <= DECODE;
140                 end if; -- busdone
141
142             when DECODE =>
143                 -- start outer big decode case statement
144                 -- Note - this is more complex than just one case statement on
145                 -- one set of bits as some operations only have 2 unique bits ,
146                 -- and some have 8 unique bits
147                 case current_inst(15 downto 12) is
148                     -- TODO when "0000" => -- cmpi (immediate)
149                     -- current_op <= OP_CMPI;
150                     -- TODO decode effective addresses
151                     -- next_fdc_state <= EXEC_READSRC;
152
153                     when "0001" => -- move.b
154                         current_op <= OP_MOVE;
155                         opdatasize <= "01"; -- byte
156                         -- decode effective addresses
157                         src_addrcode <= current_inst(5 downto 0);
158                         dst_addrcode <= current_inst(11 downto 6);
159                         next_fdc_state <= EXEC_READSRC;
160
161                     when "0010" => -- move.w
162                         current_op <= OP_MOVE;
163                         opdatasize <= "10"; -- word
164                         -- decode effective addresses
165                         src_addrcode <= current_inst(5 downto 0);
166                         dst_addrcode <= current_inst(11 downto 6);
167                         next_fdc_state <= EXEC_READSRC;
168
169                     when "0011" => -- move.l
170                         current_op <= OP_MOVE;
171                         opdatasize <= "11"; -- long
172                         -- decode effective addresses
173                         src_addrcode <= current_inst(5 downto 0);

```

```

175         dst_addrcode <= current_inst(11 downto 6);
176         next_fdc_state <= EXEC_READSRC;
177
178     when "0100" => -- nop
179         if current_inst(11 downto 0) = "111001110001"
180         then -- no-op
181             current_op <= OP_NOP;
182             next_fdc_state <= FETCH_READINS;
183         else -- PANIC due to unimplemented opcode
184             next_fdc_state <= FDC_HALT;
185             panic <= '1';
186             report "CPU Panic - unimplemented opcode"
187                 severity FAILURE;
188         end if;
189
190     when "0110" => -- branch
191         if current_inst(11 downto 8) = "0000"
192         then -- branch
193             current_op <= OP_BRA;
194             -- decode destination
195             if current_inst(7 downto 0) = x"00"
196             then -- 16 bit branch - read another word
197                 src_addrcode <= "111100"; -- immediate
198                 next_fdc_state <= EXEC_READSRC;
199             else -- 8 bit branch - handle here
200                 if current_inst(7) = '1'
201                 then -- sign extend
202                     src_data <= x"FFFFFF" & current_inst(7 downto
203                         0);
204                 else
205                     src_data <= x"000000" & current_inst(7 downto
206                         0);
207                 end if; -- sign extend
208                 next_fdc_state <= EXEC_DOOP;
209             end if; -- 8 bit destination
210
211         else -- PANIC due to unimplemented opcode
212             next_fdc_state <= FDC_HALT;
213             panic <= '1';
214             report "CPU Panic - unimplemented opcode"
215                 severity FAILURE;
216         end if;
217
218     -- TODO when "1011" => -- cmp (not immediate)
219     -- current_op <= OP_CMP;
220     -- TODO decode effective addresses
221     -- next_fdc_state <= EXEC_READSRC;
222
223     when "1101" => -- add.b/w/l
224         current_op <= OP_ADD;
225         -- determines source dest order of operands
226         if (current_inst(8) = '1')
227         then
228             next_fdc_state <= FDC_HALT;
229             panic <= '1';
230             report "CPU Panic - unimplemented addressing mode"
231                 severity FAILURE;
232         else
233             opdatasize <= current_inst(7 downto 6);
234             src_addrcode <= current_inst(5 downto 0);
235             -- dest is a data register

```

```

233         dst_addrcode <= "000" & current_inst(11 downto 9);
234     end if;
235     next_fdc_state <= EXEC_READSRC;

237     when "1100" => -- and.b/w/l
238         current_op <= OP_AND;
239         -- decode effective addresses
240         -- detemines source dest order of operands
241         if (current_inst(8) = '1')
242         then
243             next_fdc_state <= FDC_HALT;
244             panic <= '1';
245             report "CPU Panic - unimplemented addressing mode"
246                 severity FAILURE;
247         else
248             opdatasize <= current_inst(7 downto 6);
249             src_addrcode <= current_inst(5 downto 0);
250             -- dest is a data register
251             dst_addrcode <= "000" & current_inst(11 downto 9);
252         end if;
253         next_fdc_state <= EXEC_READSRC;

255     when others => -- PANIC due to unimplemented opcode
256         next_fdc_state <= FDC_HALT;
257         panic <= '1';
258         report "CPU Panic - unimplemented opcode"
259             severity FAILURE;
260     end case; -- current inst is?
261     -- end outer big decode case statement

263     when EXEC_READSRC => -- TODO direct memory
264         case src_addrcode(5 downto 3) is
265             when "000" => -- data register
266                 -- this should be replaced with an integer/unsigned
267                 -- expression on the array index
268                 case src_addrcode(2 downto 0) is
269                     when "000" => src_data <= data_registers(0);
270                     when "001" => src_data <= data_registers(1);
271                     when "010" => src_data <= data_registers(2);
272                     when "011" => src_data <= data_registers(3);
273                     when "100" => src_data <= data_registers(4);
274                     when "101" => src_data <= data_registers(5);
275                     when "110" => src_data <= data_registers(6);
276                     when "111" => src_data <= data_registers(7);
277                 end case; -- src_addrcode register section
278                 next_fdc_state <= EXEC_WAITSRC;

279             when "001" => -- address register
280                 -- this should be replaced with an integer/unsigned
281                 -- expression on the array index
282                 case src_addrcode(2 downto 0) is
283                     when "000" => src_data <= address_registers(0);
284                     when "001" => src_data <= address_registers(1);
285                     when "010" => src_data <= address_registers(2);
286                     when "011" => src_data <= address_registers(3);
287                     when "100" => src_data <= address_registers(4);
288                     when "101" => src_data <= address_registers(5);
289                     when "110" => src_data <= address_registers(6);
290                     when "111" => src_data <= address_registers(7);
291                 end case; -- src_addrcode register section
292                 next_fdc_state <= EXEC_WAITSRC;

```

```

295     when "111" => -- could be immediate, absolute, offset
                case src_addrcode(2 downto 0) is
297
300         when "000" => -- word addr absolute
                case extaddrpass is
301         when "00" =>
302             addrbus <= program_counter;
303             busrw <= '0'; --read
304             -- note we don't use opdatasize as that is
305             -- the size of the final value we want not
306             -- the address it is at.
307             busdatasize <= "10";
308             busreq <= '1'; --request
309             next_fdc_state <= EXEC_WAITSRC;
310
311         when "01" => -- pass 2
                -- read from location we just got
                addrbus <= src_data;
313
314             busrw <= '0'; --read
315             if opdatasize = "11"
316             then -- long
317                 busdatasize <= "11";
318             else --word size, byte is in lower half
319                 busdatasize <= "10";
320             end if; --datasize
321             busreq <= '1'; --request
322             next_fdc_state <= EXEC_WAITSRC;
323
324         when others =>
325             next_fdc_state <= FDC_HALT;
326             panic <= '1';
327             report "CPU Panic - bad extaddrpass"
328                 severity FAILURE;
329     end case; -- extaddrpass
330
331 when "001" => -- long addr absolute
                case extaddrpass is
332         when "00" =>
333             addrbus <= program_counter;
334             busrw <= '0'; --read
335             -- note we don't use opdatasize as that is
336             -- the size of the final value we want not
337             -- the address it is at.
338             busdatasize <= "11";
339             busreq <= '1'; --request
340             next_fdc_state <= EXEC_WAITSRC;
341
342         when "01" => -- pass 2
                -- read from location we just got
                addrbus <= src_data;
343
344             busrw <= '0'; --read
345             if opdatasize = "11"
346             then -- long
347                 busdatasize <= "11";
348             else --word size, byte is in lower half
349                 busdatasize <= "10";
350             end if; --datasize
351             busreq <= '1'; --request
352
353

```

```

355         next_fdc_state <= EXEC_WAITSRC;

357         when others =>
359             next_fdc_state <= FDC_HALT;
361             panic <= '1';
363             report "CPU Panic - bad extaddrpass"
365                 severity FAILURE;
367         end case; -- extaddrpass

369     when "100" => -- immediate
371         addrbus <= program_counter;
373         busrw <= '0'; --read
375         if opdatasize = "11"
377             then -- long
379                 busdatasize <= "11";
381             else --word size , byte is in lower half
383                 busdatasize <= "10";
385             end if; --datasize
387         busreq <= '1'; --request
389         next_fdc_state <= EXEC_WAITSRC;
391     -- end when immediate

393     when others =>
395         next_fdc_state <= FDC_HALT;
397         panic <= '1';
399         report "CPU Panic - invalid addresssing mode"
401             severity FAILURE;
403     end case; -- addr mode 2 downto 0

405     when others =>
407         next_fdc_state <= FDC_HALT;
409         panic <= '1';
411         report "CPU Panic - unimplemented addressing mode"
413             severity FAILURE;
415     end case; -- src_addrmode 5 downto 3

417 when EXEC_WAITSRC => -- TODO direct memory
419     case src_addrmode(5 downto 3) is
421     when "000" => -- data register
423         -- TODO this is necessary for the clocked registers
425         later
427         next_fdc_state <= EXEC_READDST;
429         -- TODO seperate ALU next_fdc_state <= EXEC_PUTSRCALU;
431     when "001" => -- address register clock
433         -- TODO this is necessary for the clocked registers
435         later
437         next_fdc_state <= EXEC_READDST;
439         -- TODO seperate ALU next_fdc_state <= EXEC_PUTSRCALU;

441     when "111" => -- immediate or direct
443         if busdone = '1'
445             then
447                 busreq <= '0'; --end request
449                 case src_addrmode(2 downto 0) is
451                 when "000" => -- word absolute address
453                     case extaddrpass is
455                     when "00" => -- FIXME test
457                         program_counter <= std_logic_vector(
459                             unsigned(program_counter) + 2);
461                         src_data <= x"0000" & databus(15 downto
463                             0);

```



```

413         -- set to 01 here - we've made 1 pass
         extaddrpass <= "01";
415
         next_fdc_state <= EXEC_READSRC;
417
         when "01" => -- FIXME test
         if current_op = OP_BRA
         then
421             -- sign extend
             src_data <= x"FFFF" & databus(15
                 downto 0);
423         else
             src_data <= databus; -- regardless of
                 data size
425                                     -- both of these
                                     are 32 bits
         end if; -- current op is branch
         extaddrpass <= "10"; -- 2nd pass done
         next_fdc_state <= EXEC_READDST;
429         -- TODO sep ALU next_fdc_state <=
             EXEC_PUTSRCALU;

         when others =>
             next_fdc_state <= FDC_HALT;
433             panic <= '1';
             report "CPU Panic - bad extaddrpass"
                 severity FAILURE;
435         end case; -- extaddrpass

437
         when "001" => -- long absolute address TODO
         program_counter <= std_logic_vector(
             unsigned(program_counter) + 4);
439         -- TODO
         next_fdc_state <= FDC_HALT;
441         panic <= '1';
         report "CPU Panic - unimplemented addressing
             mode"
             severity FAILURE;
443
445
         when "100" => -- immediate data
         if opdatasize = "11"
         then -- long
449             program_counter <= std_logic_vector(
                 unsigned(program_counter) + 4);
451         else -- word size, byte is in lower half
             program_counter <= std_logic_vector(
                 unsigned(program_counter) + 2);
453         end if; -- datasize

455
         if current_op = OP_BRA
         then
457             -- sign extend
             src_data <= x"FFFF" & databus(15 downto 0);
459         else
             src_data <= databus; -- regardless of data
                 size
461                                     -- both of these are 32
                                     bits
         end if; -- current op is branch
         next_fdc_state <= EXEC_READDST;
463
465

```

```

467         -- TODO sep ALU next_fdc_state <=
468         EXEC_PUTSRCALU;
469
470     when others =>
471         next_fdc_state <= FDC_HALT;
472         panic <= '1';
473         report "CPU Panic - unimplemented addressing
474             mode"
475             severity FAILURE;
476     end case; -- immediate or direct
477 end if; -- busdone
478
479 when others =>
480     next_fdc_state <= FDC_HALT;
481     panic <= '1';
482     report "CPU Panic - unimplemented addressing mode"
483         severity FAILURE;
484 end case; -- src_addrcode
485
486 when EXEC_READDST => -- TODO direct memory
487     case current_op is
488     when OP_BRA =>
489         -- Instruction has only one operand
490         -- Note that for branh, we already skip this state
491         next_fdc_state <= EXEC_DLOOP;
492     when others =>
493         case dst_addrcode(5 downto 3) is
494         when "000" => -- data register
495             -- this should be replaced with an integer/
496             unsigned
497             -- expression on the array index
498             case dst_addrcode(2 downto 0) is
499             when "000" => dst_data <= data_registers(0);
500             when "001" => dst_data <= data_registers(1);
501             when "010" => dst_data <= data_registers(2);
502             when "011" => dst_data <= data_registers(3);
503             when "100" => dst_data <= data_registers(4);
504             when "101" => dst_data <= data_registers(5);
505             when "110" => dst_data <= data_registers(6);
506             when "111" => dst_data <= data_registers(7);
507         end case; -- dst_addrcode register section
508         next_fdc_state <= EXEC_WAITDST;
509
510     when "001" => -- address register
511         -- this should be replaced with an integer/
512         unsigned
513         -- expression on the array index
514         case dst_addrcode(2 downto 0) is
515         when "000" => dst_data <= address_registers(0);
516         when "001" => dst_data <= address_registers(1);
517         when "010" => dst_data <= address_registers(2);
518         when "011" => dst_data <= address_registers(3);
519         when "100" => dst_data <= address_registers(4);
520         when "101" => dst_data <= address_registers(5);
521         when "110" => dst_data <= address_registers(6);
522         when "111" => dst_data <= address_registers(7);
523         end case; -- dst_addrcode register section
524         next_fdc_state <= EXEC_WAITDST;
525
526     when "111" => -- could be immediate or direct
527         if src_addrcode(2 downto 0) = "100"

```

```

523         then -- immediate data invalid as destination
524             next_fdc_state <= FDC_HALT;
525             panic <= '1';
526             report "CPU Panic - immediate addressing used
527                 as dest."
528                 severity FAILURE;
529         else
530             next_fdc_state <= FDC_HALT;
531             panic <= '1';
532             report "CPU Panic - unimplemented addressing
533                 mode"
534                 severity FAILURE;
535         end if; -- when 111

536     when others =>
537         next_fdc_state <= FDC_HALT;
538         panic <= '1';
539         report "CPU Panic - unimplemented addressing mode"
540             severity FAILURE;
541     end case; -- src_addrcode
542 end case; -- need destination data read

543 when EXEC_WAITDST => -- TODO direct memory
544     case dst_addrcode(5 downto 3) is
545     when "000" => -- data register
546         -- TODO this is necessary for the clocked registers
547         later
548         next_fdc_state <= EXEC_DOOP;
549         -- TODO separate ALU next_fdc_state <= EXEC_PUTDSTALU;
550     when "001" => -- address register clock
551         -- TODO this is necessary for the clocked registers
552         later
553         next_fdc_state <= EXEC_DOOP;
554         -- TODO separate ALU next_fdc_state <= EXEC_PUTDSTALU;
555     when others =>
556         next_fdc_state <= FDC_HALT;
557         panic <= '1';
558         report "CPU Panic - unimplemented addressing mode"
559             severity FAILURE;
560     end case; --src_addrcode

561 when EXEC_DOOP =>
562     case current_op is
563     when OP_MOVE =>
564         case opdatasize is
565         when "11" => -- long
566             rslt_data <= src_data;
567         when "10" => -- word
568             rslt_data <= dst_data(31 downto 16) &
569                 src_data(15 downto 0);
570         when "01" => -- byte
571             rslt_data <= dst_data(31 downto 8) &
572                 src_data(7 downto 0);
573         when others =>
574             next_fdc_state <= FDC_HALT;
575             panic <= '1';
576             report "CPU Panic - invalid operation data size"
577                 severity FAILURE;
578         end case;
579     next_fdc_state <= EXEC_DOFLAGS;

```

```

579
580     when OP_BRA =>
581         program_counter <= std_logic_vector(
582             unsigned(program_counter)
583             + unsigned(src_data));
584         -- Go direct to to next instruction , no flags change
585         next_fdc_state <= FETCH_READINS;
586
587     when OP_ADD =>
588         case opdatasize is
589             when "11" => -- long
590                 rslt_data <= std_logic_vector(unsigned(src_data)
591                     + unsigned(dst_data));
592
593             when "10" => -- word
594                 rslt_data <= dst_data(31 downto 16) &
595                     std_logic_vector(
596                         unsigned(src_data(15 downto 0))
597                         + unsigned(dst_data(15 downto 0)));
598
599             when "01" => -- byte
600                 rslt_data <= dst_data(31 downto 8) &
601                     std_logic_vector(
602                         unsigned(src_data(7 downto 0))
603                         + unsigned(dst_data(7 downto 0)));
604
605             when others =>
606                 next_fdc_state <= FDC_HALT;
607                 panic <= '1';
608                 report "CPU Panic - invalid operation data size"
609                     severity FAILURE;
610         end case;
611         -- TODO: this needs to be done in compnent ALU which
612         -- also determines carry/overflow flags
613         next_fdc_state <= EXEC_DOFLAGS;
614
615     when OP_AND =>
616         case opdatasize is
617             when "11" => -- long
618                 rslt_data <= src_data and dst_data;
619             when "10" => -- word
620                 rslt_data <= dst_data(31 downto 16) &
621                     (src_data(15 downto 0) and
622                     dst_data(15 downto 0));
623             when "01" => -- byte
624                 rslt_data <= dst_data(31 downto 8) &
625                     (src_data(7 downto 0) and
626                     dst_data(7 downto 0));
627             when others =>
628                 next_fdc_state <= FDC_HALT;
629                 panic <= '1';
630                 report "CPU Panic - invalid operation data size"
631                     severity FAILURE;
632         end case;
633         next_fdc_state <= EXEC_DOFLAGS;
634
635     when others =>
636         next_fdc_state <= FDC_HALT;
637         panic <= '1';
638         report "CPU Panic - unimplemented operation"
639             severity FAILURE;
640     end case;
641
642     when EXEC_DOFLAGS =>

```

```

641     case current_op is -- x/carry/overflow
        when OP_MOVE | OP_AND =>
            -- determine flags
643             -- X not affected by move/and
            status_register(0) <= '0'; --carryt
645             status_register(1) <= '0'; --overflow
        when OP_ADD =>
647             -- TODO: arithmetic ops set flags from ALU in EXEC_DOOP
        when others =>
649             next_fdc_state <= FDC_HALT;
            panic <= '1';
651             report "No X/C/O flags coded for op, possible state
                    error"
                    severity WARNING;
653     end case; -- current_op for x/carry/overflow

655     case current_op is -- negative/zero - nearly all ops
        when OP_MOVE | OP_AND | OP_ADD =>
657             case opdatasize is
                when "01" => -- byte
659                 if rslt_data(7 downto 0) = x"00"
                    then --zero
661                     status_register(2) <= '1'; --zero
                        status_register(3) <= '0'; --negative
663                 else
                    status_register(2) <= '0'; --zero
665                     if rslt_data(7) = '1'
                        then
667                         status_register(3) <= '1'; --negative
                    else
669                         status_register(3) <= '0'; --negative
                    end if; -- negative
                end if; -- zero
                when "10" => -- word
673                 if rslt_data(15 downto 0) = x"0000"
                    then --zero
675                     status_register(2) <= '1'; --zero
                        status_register(3) <= '0'; --negative
677                 else
                    status_register(2) <= '0'; --zero
679                     if rslt_data(15) = '1'
                        then
681                         status_register(3) <= '1'; --negative
                    else
683                         status_register(3) <= '0'; --negative
                    end if; -- negative
                end if; -- zero
                when "11" => -- long
687                 if rslt_data(31 downto 0) = x"00000000"
                    then --zero
689                     status_register(2) <= '1'; --zero
                        status_register(3) <= '0'; --negative
691                 else
                    status_register(2) <= '0'; --zero
693                     if rslt_data(31) = '1'
                        then
695                         status_register(3) <= '1'; --negative
                    else
697                         status_register(3) <= '0'; --negative
                    end if; -- negative
                end if; -- zero
699

```

```

701         when others =>
              next_fdc_state <= FDC_HALT;
              panic <= '1';
703         report "CPU Panic – invalid operation data size"
              severity FAILURE;
705     end case; — datasize for n/z flags
        next_fdc_state <= EXEC_PUTRSLT;
707
        when others =>
709             next_fdc_state <= FDC_HALT;
              panic <= '1';
711             report "No N/Z flags coded for op, possible state error"
              severity WARNING;
713     end case; — current_op for n/z

715 when EXEC_PUTRSLT => — TODO direct memory
        case dst_addrcode(5 downto 3) is
717     when "000" => — data register
              — this should be replaced with an integer/unsigned
              — expression on the array index
              case dst_addrcode(2 downto 0) is
721         when "000" => data_registers(0) <= rslt_data;
              when "001" => data_registers(1) <= rslt_data;
723         when "010" => data_registers(2) <= rslt_data;
              when "011" => data_registers(3) <= rslt_data;
725         when "100" => data_registers(4) <= rslt_data;
              when "101" => data_registers(5) <= rslt_data;
727         when "110" => data_registers(6) <= rslt_data;
              when "111" => data_registers(7) <= rslt_data;
729         end case; — dst_addrcode register section
        next_fdc_state <= EXEC_WAITRSLT;
731
        when "001" => — address register
              — this should be replaced with an integer/unsigned
              — expression on the array index
              case dst_addrcode(2 downto 0) is
733         when "000" => address_registers(0) <= rslt_data;
              when "001" => address_registers(1) <= rslt_data;
735         when "010" => address_registers(2) <= rslt_data;
              when "011" => address_registers(3) <= rslt_data;
737         when "100" => address_registers(4) <= rslt_data;
              when "101" => address_registers(5) <= rslt_data;
739         when "110" => address_registers(6) <= rslt_data;
              when "111" => address_registers(7) <= rslt_data;
741         end case; — dst_addrcode register section
743         next_fdc_state <= EXEC_WAITRSLT;
745
        when others =>
747             next_fdc_state <= FDC_HALT;
              panic <= '1';
749             report "CPU Panic – unimplemented addressing mode"
              severity FAILURE;
751     end case; — src_addrcode
753
        when EXEC_WAITRSLT => — TODO direct memory
755         case dst_addrcode(5 downto 3) is
              when "000" => — data register
              — TODO this is necessary for the clocked registers
              later
757                 next_fdc_state <= FETCH_READINS;

```

```
759         when "001" => -- address register clock
              -- TODO this is necessary for the clocked registers
              later
761             next_fdc_state <= FETCH_READINS;
              -- TODO seperate ALU next_fdc_state <= EXEC_PUTDSTALU;
763         when others =>
              next_fdc_state <= FDC_HALT;
765             panic <= '1';
              report "CPU Panic - unimplemented addressing mode"
767                 severity FAILURE;
         end case; --src_addrcode

769     when FDC_HALT =>
771         panic <= '1';
         next_fdc_state <= FDC_HALT;

773     when others => -- This should not occur in final version
775         next_fdc_state <= FDC_HALT;
         panic <= '1';
777         report "CPU Panic - unimplemented state" severity FAILURE;

779     end case; -- end huge current state statement of doom
     end if; -- rising edge clock
781 end if; --reset
     current_fdc_state <= next_fdc_state;
783 end process fdc_activity;
end behaviour;
```

## B Appendix B: DE1 Interface Source Code

### B.1 m68k\_de1.vhd

```

1  -- DE1 board <=> 68k cpu interface
2  -- Dylan Leigh s3017239
3
4  -- Controls:
5  --
6  -- SW 9-8: select program:
7  --     00: move/add/branch test
8  --     01: move flags test
9  --     10: move/and/branch test
10 --     11: all memory is a nop
11 --
12 -- SW 0-7: Binary clock speed selector
13 --     selects divider for auto clock
14 --
15 -- KEY3: Reset
16 -- KEY2: Hold for auto clock
17 -- KEY0: Clock (for manual clock)
18 --
19 -- Outputs:
20 -- HEX: Lower bytes of address and data busses
21 --
22 -- LEDR9: panic (cpu halts until reset)
23 -- LEDR2-8: UNUSED
24 -- LEDR1 - bus write (should not be lit inside any sample programs)
25 -- LEDR0 - Bus request (should be same as busdone)
26 --
27 -- LEDG7 - Bus done (should be same as bus request)
28 -- LEDG5-6: Bus data size (should be 10 in the sample programs)
29 -- LEDG0-4: SR Flags
30 --
31 library ieee;
32 use ieee.std_logic_1164.all;
33 use ieee.numeric_std.all;
34
35 entity m68k_de1 is
36     port (
37         -- All off the board
38         KEY: in std_logic_vector(3 downto 0);
39         SW: in std_logic_vector(9 downto 0);
40
41         -- have to use "0 downto 0" as default pinout uses vectors for all
42         CLOCK_50: in std_logic;
43
44         HEX0: out std_logic_vector(6 downto 0);
45         HEX1: out std_logic_vector(6 downto 0);
46         HEX2: out std_logic_vector(6 downto 0);
47         HEX3: out std_logic_vector(6 downto 0);
48
49         LEDG: out std_logic_vector(7 downto 0);
50         LEDR: out std_logic_vector(9 downto 0)
51     );
52 end m68k_de1;
53
54 architecture mixed of m68k_de1 is
55     -- logic for displaying 4 bits as a hex digit on the 7 segment display.
56     component hex7seg
57         port
58         (
59             inbits: in std_logic_vector(3 downto 0);
60             hexout: out std_logic_vector(0 to 6)
61         );
62     end component;

```



```

64  -- clock divider
65  component clockdiv is
66    Port (  rst : in std_logic;
67           clk : in std_logic;
68           clkout : out std_logic;
69           speed : in std_logic_vector (2 downto 0)
70         );
71  end component;
72
73  -- cpu
74  component m68k_cpu_core is
75    port (
76      reset: in std_logic; -- active high
77      clock: in std_logic;
78
79      panic: out std_logic; -- set high when CPU panics and halts
80            -- nothing changes until manually reset
81
82      -- bus controls
83      busrw: out std_logic; -- bus read/write: zero is write
84      busreq: out std_logic; -- set to 1 to make a request from the MMU
85      busdone: in std_logic; -- is set to 1 when request fulfilled by the MMU
86      busdatasize: out std_logic_vector(1 downto 0);
87      -- 68k convention: 01: byte (8), 10: word(16), 11: long (32)
88
89      -- address bus
90      addrbus: out std_logic_vector(31 downto 0);
91      -- data bus
92      databus: in std_logic_vector(31 downto 0);
93      -- memory writes TODO databus: inout std_logic_vector(31 downto 0)
94
95      -- outputs for debugging purposes
96      flags: out std_logic_vector(4 downto 0)
97    );
98  end component;
99
100  signal clock: std_logic;
101  signal busrw: std_logic;
102  signal busreq: std_logic;
103  signal busdone: std_logic;
104  signal busdatasize: std_logic_vector(1 downto 0);
105  signal addrbus: std_logic_vector(31 downto 0);
106  signal databus: std_logic_vector(31 downto 0);
107  signal flags: std_logic_vector(4 downto 0);
108
109  signal divclk: std_logic;
110
111  begin
112  cpu: m68k_cpu_core port map (
113    reset => not KEY(3), clock => clock, panic => LEDR(9),
114    busrw => busrw, busreq => busreq, busdone => busdone,
115    busdatasize => busdatasize,
116    addrbus => addrbus, databus => databus, flags => flags);
117
118  -- clock divider
119  div: clockdiv port map (rst => not KEY(3), clk => CLOCK_50,
120    clkout => divclk, speed => SW(3 downto 1));
121
122  -- board stuff
123  -- clock manually on key0 or continuously when key1 pressed
124  clock <= (SW(0) and divclk) or (not KEY(0));
125  -- LEDR(7) <= divclk; -- distracting
126  LEDR(4) <= clock;
127  LEDR(1) <= busrw;
128  LEDR(0) <= busreq;

```

```

130 LEDG(7) <= busdone;
131 LEDG(4 downto 0) <= flags;
132 LEDG(6 downto 5) <= busdatasize;
133 h0: hex7seg port map (databus(3 downto 0), HEX0);
134 h1: hex7seg port map (databus(7 downto 4), HEX1);
135 h2: hex7seg port map (addrbus(3 downto 0), HEX2);
136 h3: hex7seg port map (addrbus(7 downto 4), HEX3);
137
138 -- memory read responses
139 process (clock)
140 begin
141     if rising_edge(clock) then
142         if (busreq = '1') then
143             -- this program is all reads, word size
144             assert (busrw = '0')
145                 report "CPU requested a write"
146                 severity WARNING;
147             assert (busdatasize = "10")
148                 report "Bus data size request not 16 bits"
149                 severity WARNING;
150
151             case SW(9 downto 8) is -- select program
152                 when "00" => -- from fakemmu_1
153                     -- memory reads
154                     case addrbus is
155                         when x"00000100" => -- move.w #02, d1
156                             databus <= "00000000000000000000010000001111100";
157                                     -- ^ ^ ^ ^ immediate
158                                     -- | | +- data reg d1
159                                     -- | +- word size
160                                     -- +- move
161
162                         when x"00000102" =>
163                             databus <= x"00020002"; -- the #$20002
164
165                         when x"00000104" => -- move.w #03, d2
166                             databus <= "000000000000000000010000010111100";
167
168                         when x"00000106" =>
169                             databus <= x"00030003"; -- the #$30003
170
171                     -- add together
172                     when x"00000108" => -- add d2 to d1
173                         databus <= "000000000000000001101001001000010";
174                                     -- ^ ^ ^ ^ source data reg 2
175                                     -- | | +- word length
176                                     -- | | +- destination is data
177                                     -- reg
178                                     -- | +- which data register
179                                     -- +- add
180
181                     when x"0000010A" => -- add #5 to d0
182                         databus <= "00000000000000000110100001111100";
183                                     -- ^ ^ ^ ^ immediate
184                                     -- | | +- word length
185                                     -- | | +- destination is data
186                                     -- reg
187                                     -- | +- which data register
188                                     -- +- add
189
190                     when x"0000010C" => -- the #$50005
191                         databus <= x"00050005"; -- the #$50005
192
193                     when x"0000010E" => -- bra $100
194                         databus <= "00000000000000000110000011110000";
195                                     -- ^ ^ -- offset
196                                     -- +- branch
197                         report "Branching to start..." severity NOTE;

```

```

194         when others =>
195             report "Memory request from unhandled location"
196             severity WARNING;
197     end case;
198 when "01" => -- from fakemmu_2
199     case addrbus is
200     when x"0000100" => -- move.w #0, d1
201         databus <= "00000000000000001000001111100";
202             -- ^ ^ ^ ^ immediate
203             -- | | +- data reg d1
204             -- | +- word size
205             -- +- move
206
207     when x"0000102" =>
208         databus <= x"00000000"; -- the #0
209
210     when x"0000104" => -- move.w #$DEADBEEF, d2
211         databus <= "00000000000000001000010111100";
212     when x"0000106" =>
213         databus <= x"DEADBEEF"; -- the immediate data
214
215     -- more to come here when more opcodes implemented
216     -- add and save in d0
217
218     when x"0000108" => -- bra $100
219         databus <= "0000000000000000110000000000000";
220             -- ^ ^ ^ ^ -- offset all 0
221             -- +- branch
222
223     when x"000010A" => -- bra $100
224         databus <= x"0000FFF4"; -- -12 decimal in word
225         report "Branching to start..." severity NOTE;
226
227     when others =>
228         report "Memory request from unhandled location"
229         severity WARNING;
230     end case;
231 when "10" => -- fakemmu_3
232     case addrbus is
233     when x"0000100" => -- move.w 10 repeating, d1
234         databus <= "00000000000000001000001111100";
235             -- ^ ^ ^ ^ immediate
236             -- | | +- data reg d1
237             -- | +- word size
238             -- +- move
239
240     when x"0000102" =>
241         databus <= "101010101010101010101010101010";
242
243     when x"0000104" => -- move.w 01 repeating, d2
244         databus <= "00000000000000001000010111100";
245     when x"0000106" =>
246         databus <= "010101010101010101010101010101";
247
248     -- and together
249     when x"0000108" => -- and d2 to d1
250         databus <= "0000000000000001100001001000010";
251             -- ^ ^ ^ ^ source data reg 2
252             -- | | +- word length
253             -- | | +- destination is data
254             -- | +- reg
255             -- | +- which data register
256             -- +- and
257
258     when x"000010A" => -- and 1all to d0
259         databus <= "000000000000000110000001111100";
260             -- ^ ^ ^ ^ immediate
261             -- | | +- word length

```

```

258                                     -- | | + destination is data
                                         reg
260                                     -- | + which data register
                                         -- + and

262         when x"000010C" =>
             databus <= x"0000FFFF"; -- the #$FFFF
264
266         when x"000010E" => -- bra $100
             databus <= "0000000000000000110000011110000";
                                         -- ^         ^-- offset
268                                     -- + branch
             report "Branching to start..." severity NOTE;
270
272         when others =>
             report "Memory request from unhandled location"
             severity WARNING;
274     end case;
276     when "11" => -- always nop FIXME
             databus <= "0000000000000000100111001110001";
278     end case; --
             busdone <= '1';
280     else -- no busreq
             busdone <= '0';
             databus <= x"00000000";
282     end if; --busreq
284     end if; --clock rising edge
end process;
end mixed;

```

## B.2 hex7seg.vhd

```
1  -- vim: sw=4 ts=4 et
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.all;
5
6  ENTITY hex7seg IS
7      PORT
8      (
9          inbits: in std_logic_vector(3 downto 0);
10         hexout: out std_logic_vector(0 to 6)
11     );
12 END hex7seg;
13
14 ARCHITECTURE Behavior OF hex7seg IS
15 BEGIN
16     WITH inbits SELECT
17         hexout <= "1000000" WHEN "0000", -- 0
18                 "1111001" WHEN "0001", -- 1
19                 "0100100" WHEN "0010", -- 2
20                 "0110000" WHEN "0011", -- 3
21                 "0011001" WHEN "0100", -- 4
22                 "0010010" WHEN "0101", -- 5
23                 "0000010" WHEN "0110", -- 6
24                 "1111000" WHEN "0111", -- 7
25                 "0000000" WHEN "1000", -- 8
26                 "0011000" WHEN "1001", -- 9
27                 "0001000" WHEN "1010", -- a
28                 "0000011" WHEN "1011", -- b
29                 "1000110" WHEN "1100", -- c
30                 "0100001" WHEN "1101", -- d
31                 "0000110" WHEN "1110", -- e
32                 "0001110" WHEN "1111", -- f
33                 "1111111" WHEN OTHERS;
34 END Behavior;
```

### B.3 clockdiv.vhd

```

1  — Adjustable clock divider
2  — Takes in a signal (usually the board clock), divides it by a selectable
3  — amount based on 3 switches.
4  — vim: ts=4 sw=4 et:

6  library IEEE;
7  use IEEE.STD_LOGIC_1164.ALL;
8  use IEEE.STD_LOGIC_ARITH.ALL; — Two very useful
9  use IEEE.STD_LOGIC_UNSIGNED.ALL; — IEEE libraries
10
11 entity clockdiv is
12     Port (  rst : in std_logic;
13            clk : in std_logic;
14            clkout : out std_logic;
15            speed : in std_logic_vector (2 downto 0)
16           );
17 end clockdiv;
18
19
20 architecture behavioral of clockdiv is
21     signal temp: std_logic_vector(25 downto 0);
22 begin
23     process (clk, rst)
24     begin
25         if (rst = '1') then
26             temp <= "00000000000000000000000000";
27         elsif rising_edge(clk) then
28             temp <= temp + 1;
29         end if;
30         case speed is
31             when "000" =>
32                 if temp(25) = '1' then
33                     clkout <= '1';
34                 else
35                     clkout <= '0';
36                 end if;
37             when "001" =>
38                 if temp(24) = '1' then
39                     clkout <= '1';
40                 else
41                     clkout <= '0';
42                 end if;
43             when "010" =>
44                 if temp(23) = '1' then
45                     clkout <= '1';
46                 else
47                     clkout <= '0';
48                 end if;
49             when "011" =>
50                 if temp(22) = '1' then
51                     clkout <= '1';
52                 else
53                     clkout <= '0';
54                 end if;
55             when "100" =>
56                 if temp(21) = '1' then
57                     clkout <= '1';
58                 else
59                     clkout <= '0';
60                 end if;
61             when "101" =>
62                 if temp(19) = '1' then
63                     clkout <= '1';
64                 else

```

```
        clkout <= '0';
66     end if;
    when "110" =>
68     if temp(18) = '1' then
        clkout <= '1';
70     else
        clkout <= '0';
72     end if;
    when "111" =>
74     if temp(17) = '1' then
        clkout <= '1';
76     else
        clkout <= '0';
78     end if;
    end case;
80 end process;
end behavioral;
```

## C Appendix C: Test MMU Files

### C.1 m68k\_fakemmu\_1.vhd

```

1  -- fake/test "mmu" for m68k
2  -- returns various values set here depending on data requested.
3  -- test of move, add and branch
4  -- Dylan Leigh s3017239

6  library ieee;
7  use ieee.std_logic_1164.all;
8  use ieee.numeric_std.all;

10 entity m68k_fakemmu_1 is
11 end m68k_fakemmu_1;

12
13 architecture mixed of m68k_fakemmu_1 is
14     component m68k_cpu_core is
15         port (
16             reset: in std_logic; -- active high
17             clock: in std_logic;
18
19             panic: out std_logic; -- set high when CPU panics and halts
20                                     -- nothing changes until manually reset
21
22             -- bus controls
23             busrw: out std_logic; -- bus read/write: zero is write
24             busreq: out std_logic; -- set to 1 to make a request from the MMU
25             busdone: in std_logic; -- is set to 1 when request fulfilled by the MMU
26             busdatasize: out std_logic_vector(1 downto 0);
27                 -- 68k convention: 01: byte (8), 10: word(16), 11: long (32)
28
29             -- address bus
30             addrbus: out std_logic_vector(31 downto 0);
31             -- data bus
32             databus: inout std_logic_vector(31 downto 0);
33
34             -- for debugging
35             flags: out std_logic_vector(4 downto 0)
36         );
37     end component;

40     signal reset: std_logic;
41     signal clock: std_logic;
42     signal busrw: std_logic;
43     signal busreq: std_logic;
44     signal busdone: std_logic;
45     signal busdatasize: std_logic_vector(1 downto 0);
46     signal addrbus: std_logic_vector(31 downto 0);
47     signal databus: std_logic_vector(31 downto 0);
48
49 begin
50     cpucore: m68k_cpu_core
51         port map (
52             reset => reset, clock => clock, -- panic => null,
53             -- Note: we ignore panic as in simulation the CPU core will
54             -- raise an assertion itself on a panic.
55             busrw => busrw, busreq => busreq, busdone => busdone,
56             busdatasize => busdatasize, addrbus => addrbus, databus => databus
57         );
58
59     busread: process (busreq)
60     begin
61         if rising_edge(busreq)
62             then

```



```

64     assert (busrw = '0')
        report "Bus requested a read."
        severity WARNING;
66     assert (busdatasize = "10")
        report "Bus data size request not 16 bits."
        severity WARNING;
68
70     -- memory reads
        case addrbus is
72         when x"0000100" => -- move.w #02, d1
            databus <= "0000000000000000010000001111100";
74             -- ^ ^ ^ ^ immediate
75             -- | | +- data reg d1
76             -- | +- word size
77             -- +- move
78         when x"0000102" =>
            databus <= x"00020002"; -- the #$20002
80
81         when x"0000104" => -- move.w #03, d2
            databus <= "0000000000000000010000010111100";
82         when x"0000106" =>
            databus <= x"00030003"; -- the #$30003
84
85     -- add together
86     when x"0000108" => -- add d2 to d1
87         databus <= "00000000000000001101001001000010";
88             -- ^ ^ ^ ^ source data reg 2
89             -- | | +- word length
90             -- | | +- destination is data reg
91             -- | +- which data register
92             -- +- add
94
95     when x"000010A" => -- add #5 to d0
96         databus <= "00000000000000001101000001111100";
97             -- ^ ^ ^ ^ immediate
98             -- | | +- word length
99             -- | | +- destination is data reg
100            -- | +- which data register
101            -- +- add
102
103     when x"000010C" => -- the #$50005
104         databus <= x"00050005"; -- the #$50005
106
107     when x"000010E" => -- bra $100
108         databus <= "0000000000000000110000011110000";
109             -- ^ ^ -- offset
110             -- +- branch
111         report "Branching to start..." severity NOTE;
112
113     when others =>
114         report "Memory request from unhandled location"
115         severity WARNING;
116     end case;
117     busdone <= '1';
118
119     else -- no busreq
120         busdone <= '0';
121         databus <= x"00000000";
122     end if; -- busreq
123 end process busread;
124
125 init_run: process
126 begin
127     reset <= '1';
128     clock <= '0';
129     wait for 1 ms;

```

```
130     clock <= '1';
131     wait for 1 ms;
132
133     reset <= '0';
134     wait for 1 ms;
135
136     — from this point, the other stuff does the work
137
138     while (true)
139     loop
140         clock <= '0';
141         wait for 1 ms;
142         clock <= '1';
143         wait for 1 ms;
144     end loop;
145
146     wait for 1000 ms;
147     end process init_run;
148 end mixed;
```

## C.2 m68k\_fakemmu\_2.vhd

```

1  — fake/test "mmu" for m68k
2  — returns various values set here depending on data requested.
3  — this one for testing flags in move instruction and long branch
4  — Dylan Leigh s3017239

6  library ieee;
7  use ieee.std_logic_1164.all;
8  use ieee.numeric_std.all;

10 entity m68k_fakemmu_2 is
11 end m68k_fakemmu_2;

12
13 architecture mixed of m68k_fakemmu_2 is
14     component m68k_cpu_core is
15         port (
16             reset: in std_logic; — active high
17             clock: in std_logic;
18
19             panic: out std_logic; — set high when CPU panics and halts
20                                     — nothing changes until manually reset
21
22             — bus controls
23             busrw: out std_logic; — bus read/write: zero is write
24             busreq: out std_logic; — set to 1 to make a request from the MMU
25             busdone: in std_logic; — is set to 1 when request fulfilled by the MMU
26             busdatasize: out std_logic_vector(1 downto 0);
27                 — 68k convention: 01: byte (8), 10: word(16), 11: long (32)
28
29             — address bus
30             addrbus: out std_logic_vector(31 downto 0);
31             — data bus
32             databus: inout std_logic_vector(31 downto 0)
33         );
34     end component;

35
36     signal reset: std_logic;
37     signal clock: std_logic;
38     signal busrw: std_logic;
39     signal busreq: std_logic;
40     signal busdone: std_logic;
41     signal busdatasize: std_logic_vector(1 downto 0);
42     signal addrbus: std_logic_vector(31 downto 0);
43     signal databus: std_logic_vector(31 downto 0);
44
45 begin
46     cpucore: m68k_cpu_core
47         port map (
48             reset => reset, clock => clock, — panic => null,
49             — Note: we ignore panic as in simulation the CPU core will
50             — raise an assertion itself on a panic.
51             busrw => busrw, busreq => busreq, busdone => busdone,
52             busdatasize => busdatasize, addrbus => addrbus, databus => databus
53         );
54
55     busread: process (busreq)
56     begin
57         if rising_edge(busreq)
58         then
59             assert (busrw = '0')
60                 report "Bus requested a read."
61                 severity WARNING;
62             assert (busdatasize = "10")
63                 report "Bus data size request not 16 bits."
64                 severity WARNING;

```

```

66     -- fake memory reads
67     case addrbus is
68         when x"0000100" => -- move.w #0, d1
69             databus <= "000000000000000010000011111100";
70             -- ^ ^ ^ ^ immediate
71             -- | | +--- data reg d1
72             -- | +--- word size
73             -- +--- move
74         when x"0000102" =>
75             databus <= x"00000000"; -- the #0
76
77         when x"0000104" => -- move.w #$BEEF, d2
78             databus <= "000000000000000010000101111100";
79         when x"0000106" =>
80             databus <= x"0000BEEF"; -- the immediate data
81
82     -- more to come here when more opcodes implemented
83     -- add and save in d0
84
85     when x"0000108" => -- bra $100
86         databus <= "000000000000000011000000000000";
87         -- ^ ^-- offset all 0
88         -- +--- branch
89     when x"000010A" => -- bra $100
90         databus <= x"0000FFF4"; -- -12 decimal in word
91         report "Branching to start..." severity NOTE;
92
93     when others =>
94         report "Memory request from unhandled location"
95         severity WARNING;
96     end case;
97     busdone <= '1';
98
99     else -- no busreq
100         busdone <= '0';
101         databus <= x"00000000";
102     end if;--busreq
103 end process busread;
104
105 init_run: process
106 begin
107     reset <= '1';
108     clock <= '0';
109     wait for 1 ms;
110
111     clock <= '1';
112     wait for 1 ms;
113
114     reset <= '0';
115     wait for 1 ms;
116
117     -- from this point, the other stuff does the work
118
119     while (true)
120     loop
121         clock <= '0';
122         wait for 1 ms;
123         clock <= '1';
124         wait for 1 ms;
125     end loop;
126
127     wait for 1000 ms;
128 end process init_run;
end mixed;

```

### C.3 m68k\_fakemmu\_3.vhd

```

1  -- fake/test "mmu" for m68k
   -- returns various values set here depending on data requested.
3  -- test of move, and and branch
   -- Dylan Leigh s3017239
5
6  library ieee;
7  use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
9
10 entity m68k_fakemmu_3 is
11 end m68k_fakemmu_3;
13
14 architecture mixed of m68k_fakemmu_3 is
   component m68k_cpu_core is
15     port (
16         reset: in std_logic; -- active high
17         clock: in std_logic;
19         panic: out std_logic; -- set high when CPU panics and halts
                                   -- nothing changes until manually reset
21
   -- bus controls
23     busrw: out std_logic; -- bus read/write: zero is write
     busreq: out std_logic; -- set to 1 to make a request from the MMU
25     busdone: in std_logic; -- is set to 1 when request fulfilled by the MMU
     busdatasize: out std_logic_vector(1 downto 0);
27         -- 68k convention: 01: byte (8), 10: word(16), 11: long (32)
29
   -- address bus
     addrbus: out std_logic_vector(31 downto 0);
31
   -- data bus
     databus: inout std_logic_vector(31 downto 0)
33     );
   end component;
35
36 signal reset: std_logic;
37 signal clock: std_logic;
38 signal busrw: std_logic;
39 signal busreq: std_logic;
40 signal busdone: std_logic;
41 signal busdatasize: std_logic_vector(1 downto 0);
42 signal addrbus: std_logic_vector(31 downto 0);
43 signal databus: std_logic_vector(31 downto 0);
45
46 begin
   cpucore: m68k_cpu_core
47     port map (
48         reset => reset, clock => clock, -- panic => null,
49         -- Note: we ignore panic as in simulation the CPU core will
           -- raise an assertion itself on a panic.
50         busrw => busrw, busreq => busreq, busdone => busdone,
51         busdatasize => busdatasize, addrbus => addrbus, databus => databus
52     );
53
54 busread: process (busreq)
   begin
55     if rising_edge(busreq)
56     then
57         assert (busrw = '0')
58             report "Bus requested a read."
59             severity WARNING;
60         assert (busdatasize = "10")
61             report "Bus data size request not 16 bits."
62             severity WARNING;

```

```

65  -- memory reads
66  case addrbus is
67  when x"0000100" => -- move.w 10 repeating, d1
68      databus <= "00000000000000000000000010000001111100";
69      -- ^ ^ ^ ^ immediate
70      -- | | +- data reg d1
71      -- | +- word size
72      -- +- move
73
74  when x"0000102" =>
75      databus <= "10101010101010101010101010101010";
76
77  when x"0000104" => -- move.w 01 repeating, d2
78      databus <= "00000000000000000000000010000010111100";
79  when x"0000106" =>
80      databus <= "01010101010101010101010101010101";
81
82  -- and together
83  when x"0000108" => -- and d2 to d1
84      databus <= "0000000000000000000000001100001001000010";
85      -- ^ ^ ^ ^ source data reg 2
86      -- | | +- word length
87      -- | | +- destination is data reg
88      -- | +- which data register
89      -- +- and
90
91  when x"000010A" => -- and lall to d0
92      databus <= "0000000000000000000000001100000001111100";
93      -- ^ ^ ^ ^ immediate
94      -- | | +- word length
95      -- | | +- destination is data reg
96      -- | +- which data register
97      -- +- and
98
99  when x"000010C" =>
100      databus <= x"0000FFFF"; -- the #FFFF
101
102  when x"000010E" => -- bra $100
103      databus <= "000000000000000000000000110000011110000";
104      -- ^ ^ ^ ^ offset
105      -- +- branch
106      report "Branching to start..." severity NOTE;
107
108  when others =>
109      report "Memory request from unhandled location"
110      severity WARNING;
111  end case;
112  busdone <= '1';
113
114  else -- no busreq
115      busdone <= '0';
116      databus <= x"00000000";
117  end if; -- busreq
118  end process busread;
119
120  init_run: process
121  begin
122      reset <= '1';
123      clock <= '0';
124      wait for 1 ms;
125
126      clock <= '1';
127      wait for 1 ms;
128
129      reset <= '0';
130      wait for 1 ms;

```

```
131      -- from this point, the other stuff does the work
133
134      while (true)
135      loop
136          clock <= '0';
137          wait for 1 ms;
138          clock <= '1';
139          wait for 1 ms;
140      end loop;
141
142      wait for 1000 ms;
143  end process init_run;
end mixed;
```

## C.4 m68k\_fakemmu\_4.vhd

```

1  -- fake/test "mmu" for m68k
2  -- returns various values set here depending on data requested.
3  -- this one tests absolute addressing
4  -- Dylan Leigh s3017239

6  library ieee;
7  use ieee.std_logic_1164.all;
8  use ieee.numeric_std.all;

10 entity m68k_fakemmu_4 is
11 end m68k_fakemmu_4;

12
13 architecture mixed of m68k_fakemmu_4 is
14     component m68k_cpu_core is
15         port (
16             reset: in std_logic; -- active high
17             clock: in std_logic;

18
19             panic: out std_logic; -- set high when CPU panics and halts
20                                     -- nothing changes until manually reset

21
22             -- bus controls
23             busrw: out std_logic; -- bus read/write: zero is write
24             busreq: out std_logic; -- set to 1 to make a request from the MMU
25             busdone: in std_logic; -- is set to 1 when request fulfilled by the MMU
26             busdatasize: out std_logic_vector(1 downto 0);
27                 -- 68k convention: 01: byte (8), 10: word(16), 11: long (32)

28
29             -- address bus
30             addrbus: out std_logic_vector(31 downto 0);
31             -- data bus
32             databus: inout std_logic_vector(31 downto 0)
33         );
34     end component;

35
36     signal reset: std_logic;
37     signal clock: std_logic;
38     signal busrw: std_logic;
39     signal busreq: std_logic;
40     signal busdone: std_logic;
41     signal busdatasize: std_logic_vector(1 downto 0);
42     signal addrbus: std_logic_vector(31 downto 0);
43     signal databus: std_logic_vector(31 downto 0);
44
45 begin
46     cpucore: m68k_cpu_core
47         port map (
48             reset => reset, clock => clock, -- panic => null,
49             -- Note: we ignore panic as in simulation the CPU core will
50             -- raise an assertion itself on a panic.
51             busrw => busrw, busreq => busreq, busdone => busdone,
52             busdatasize => busdatasize, addrbus => addrbus, databus => databus
53         );
54
55     busread: process (busreq)
56     begin
57         if rising_edge(busreq)
58         then
59             assert (busrw = '0')
60                 report "Bus requested a read."
61                 severity WARNING;
62             assert (busdatasize = "10")
63                 report "Bus data size request not 16 bits."
64                 severity WARNING;

```



```

66     case addrbus is
67         when x"00000100" => -- move.w $1a0, d1
68             databus <= "000000000000000000000000100000001111000";
69                 -- ^ ^ ^ ^ absolute source
70                 -- | | +--- to data reg d1
71                 -- | +--- word size
72                 -- +--- move
73
74         when x"00000102" =>
75             databus <= x"000001A0"; -- the address 1a0
76
77         when x"00000104" => -- move.w $1bb, d2
78             databus <= "00000000000000000000000010000010111000";
79                 -- ^ ^ ^ ^ absolute source
80                 -- | | +--- to data reg d2
81                 -- | +--- word size
82                 -- +--- move
83
84         when x"00000106" =>
85             databus <= x"000001bb"; -- the address
86
87         -- more to come here when more opcodes implemented
88         -- add and save in d0
89
90         when x"00000108" => -- bra $100
91             databus <= "0000000000000000000000001100000000000000";
92                 -- ^ ^ offset all 0
93                 -- +--- branch
94
95         when x"0000010A" => -- bra $100 offset
96             databus <= x"0000FFF4"; -- -12 decimal in word
97             report "Branching to start..." severity NOTE;
98
99         -- addresses for data
100        when x"000001A0" =>
101            databus <= x"00001111"; -- data at 1A0
102
103        when x"000001BB" =>
104            databus <= x"00002222"; -- data at 1BB
105
106        when others =>
107            report "Memory request from unhandled location"
108            severity WARNING;
109    end case;
110    busdone <= '1';
111
112    else -- no busreq
113        busdone <= '0';
114        databus <= x"00000000";
115    end if; -- busreq
116 end process busread;
117
118 init_run: process
119 begin
120     reset <= '1';
121     clock <= '0';
122     wait for 1 ms;
123
124     clock <= '1';
125     wait for 1 ms;
126
127     reset <= '0';
128     wait for 1 ms;
129
130     -- from this point, the other stuff does the work
131     while (true)

```

```
    loop
132       clock <= '0';
          wait for 1 ms;
134       clock <= '1';
          wait for 1 ms;
136     end loop;

138     wait for 1000 ms;
end process init_run;
140 end mixed;
```

## D Appendix D: Makefile

```

1 # Makefile for building m68k with GHDL
2 # Dylan Leigh s3017239

4 #
5 # Macros/Variables
6 #

8 # does not include testbenches
9 VHLSRCS= m68k_cpu_core.vhd
10 #VHLSRCS= gen_register.vhd regfile_8.vhd attic also has alu
11 VHDLBOJS= ${VHLSRCS:.vhd=.o}
12
13 #testbench sources
14 #TESTSRCS= test_gen_register.vhd attic
15 TESTSRCS= m68k_fakemmu_1.vhd m68k_fakemmu_2.vhd m68k_fakemmu_3.vhd m68k_fakemmu_4.vhd
16 TESTOBS= ${TESTSRCS:.vhd=.o}
17 TESTEXES= ${TESTSRCS:.vhd=}
18
19 #
20 # High level targets
21 #
22
23 .PHONY: default all alltests runtests clean
24
25 # builds all components and all testbenches and runs testbenches
26 default: runtests

28 # Note: following the common usage for makefiles, "all" does not mean
29 # "everything" but to build all of the final product
30 # (i.e. no testbenches or extra stuff).
31 all: ${VHDLBOJS}
32
33 # analyse tests
34 alltests: all ${TESTOBS}

36 # elaborate and execute tests
37 runtests: alltests
38     ghdl --elab-run m68k_fakemmu_1 --stop-time=300ms --wave=fakemmu_1.ghw
39     ghdl --elab-run m68k_fakemmu_2 --stop-time=200ms --wave=fakemmu_2.ghw
40     ghdl --elab-run m68k_fakemmu_3 --stop-time=300ms --wave=fakemmu_3.ghw
41     ghdl --elab-run m68k_fakemmu_4 --stop-time=300ms --wave=fakemmu_4.ghw
42
43 #
44 # Implicit targets
45 #
46
47 # clear out all suffixes
48 .SUFFIXES:
49 # # list only those we use
50 .SUFFIXES: .o .vhd .

52 .vhd.o: $<
53     ghdl -a $<
54
55 .o.: $<
56     ghdl -e $@

58 #
59 # Misc targets
60 #

62 clean:
63 #     ghdl --remove
64     rm -f *.ghw *.o work-obj93.cf ${TESTEXES}

```

## E Appendix E: Timing and Performance

This section applies to the original EEET2192 implementation. The new version of the CPU has not yet been tested on the Altera Quartus system.

### E.1 Timing

Worst case  $t_{co}$  was 19.107ns, implying an  $f_{max}$  of 52.337 MHz. The system has been tested in hardware running directly off the 50Mhz clock signal without any problems.

Timing was not a major consideration as there were no speed performance requirements; the focus was on implementing features and compatibility, without aiming for any particular clock speed. There were no tradeoffs made for timing reasons, and no changes to the design for timing issues.

### E.2 Logic Elements Used

1107 combinatorial functions and 774 logic registers were used, in total 1298 logic elements. This is only 7% of the available elements on the EP2C20F484C7 device - no changes needed to be made to the design.

As the arithmetic addition was performed with IEEE NUMERIC\_STD functions, the synthesizer was able to make use of the dedicated arithmetic logic elements within the device.

### E.3 CPU Performance

As stated earlier the CPU runs at 50MHz without problems. Some improvements could be made to the number of clock cycles taken to execute an instruction. Many of the CPU states are waiting for memory requests to complete, and many operations could feasibly be performed in parallel, or within other states.