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

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[1].

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 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[2] 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[3] 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.

[1]Read only locations and modes (such as immediate addressing) cannot be used for the destination.
[2]Refer to Section 3 on page 6.
[3]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[4].

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.

[4]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[5]).
- 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[6].

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[7].

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.

[5]Section 6 on page 12.
[6]Section B.2 on page 37.
[7]Section B.3 on page 38.

3.2.1 Outputs

The following outputs are displayed on the DE1 board LEDs:

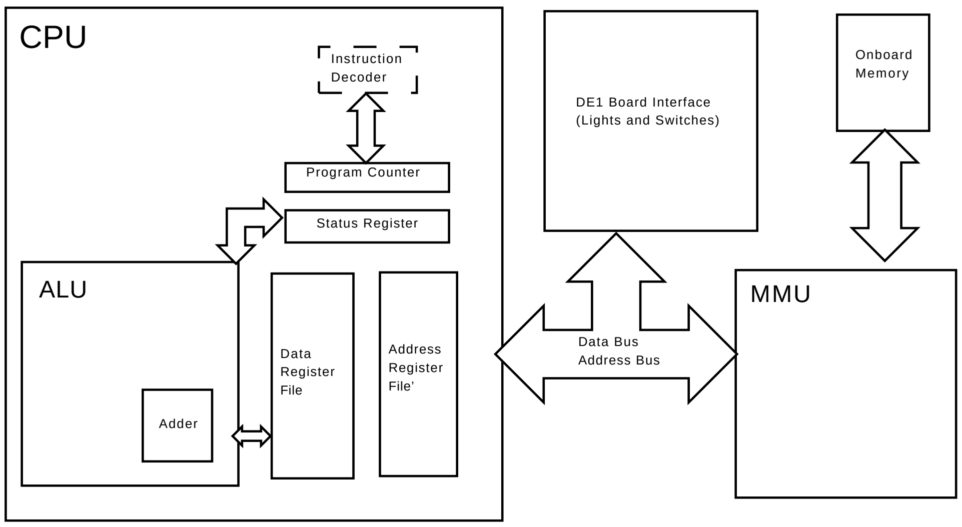
- Hex digits 2 and 3 show the lower byte of the address bus. This is usually the location of the last memory access, or the PC minus 2.
- Hex digits 0 and 1 show the lower byte of the data bus. Note that the data bus is cleared after a request has been fulfilled, so data may only show up here for one 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

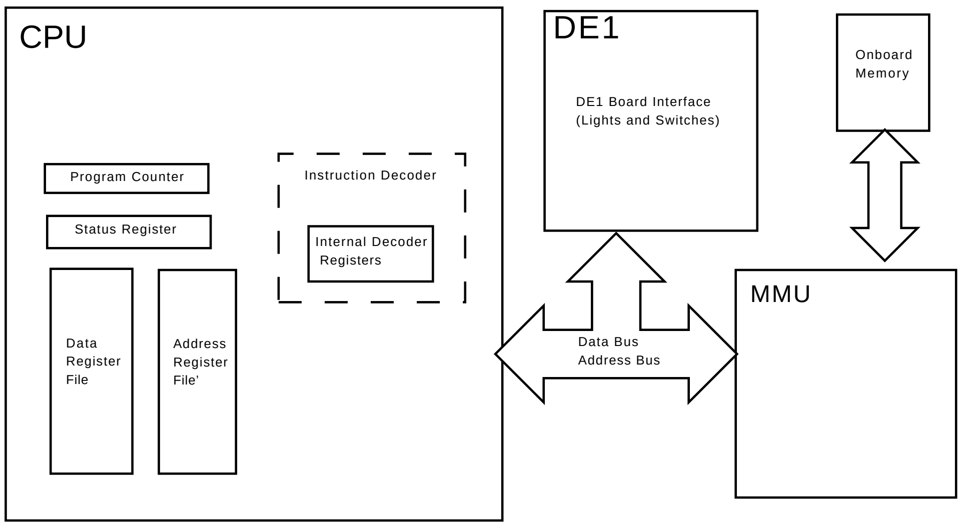
4.1 Early Component Design

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.

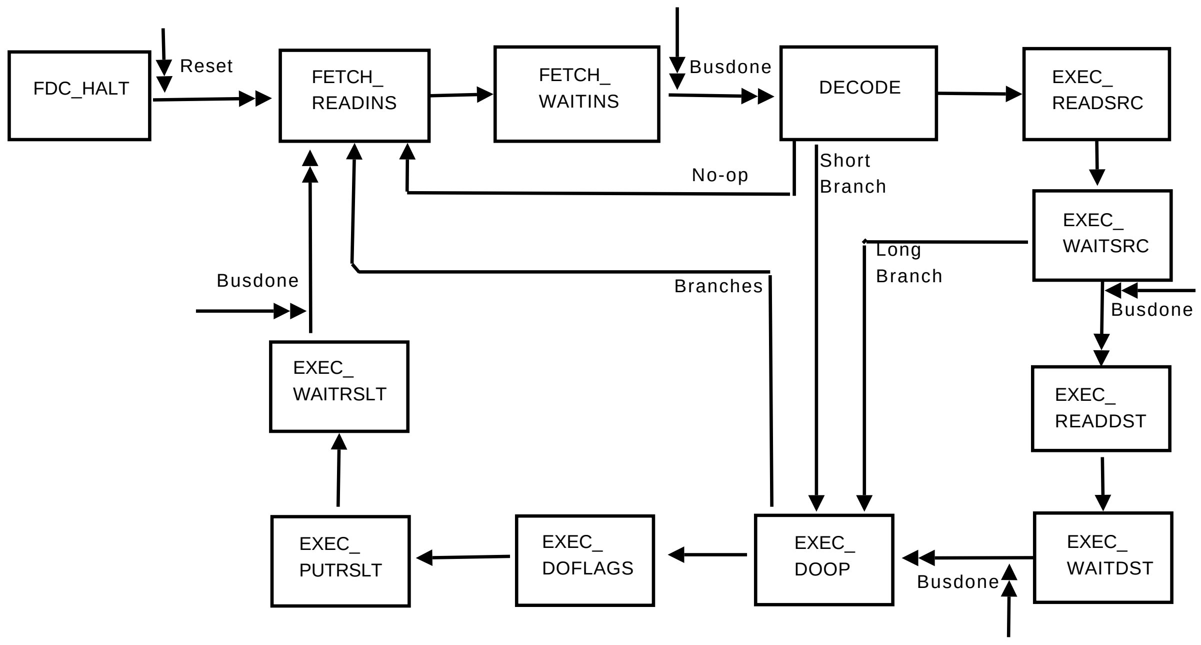
The onboard memory access was not implemented and instead four "Fake MMU" entities[8] simulate memory reads for four different sample programs. These can be selected from the DE1 interface. In simulation these entities will raise a warning if there is an attempt to read from a location that should not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

[8]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 he 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[9]).

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[10] or run in simulation[11].

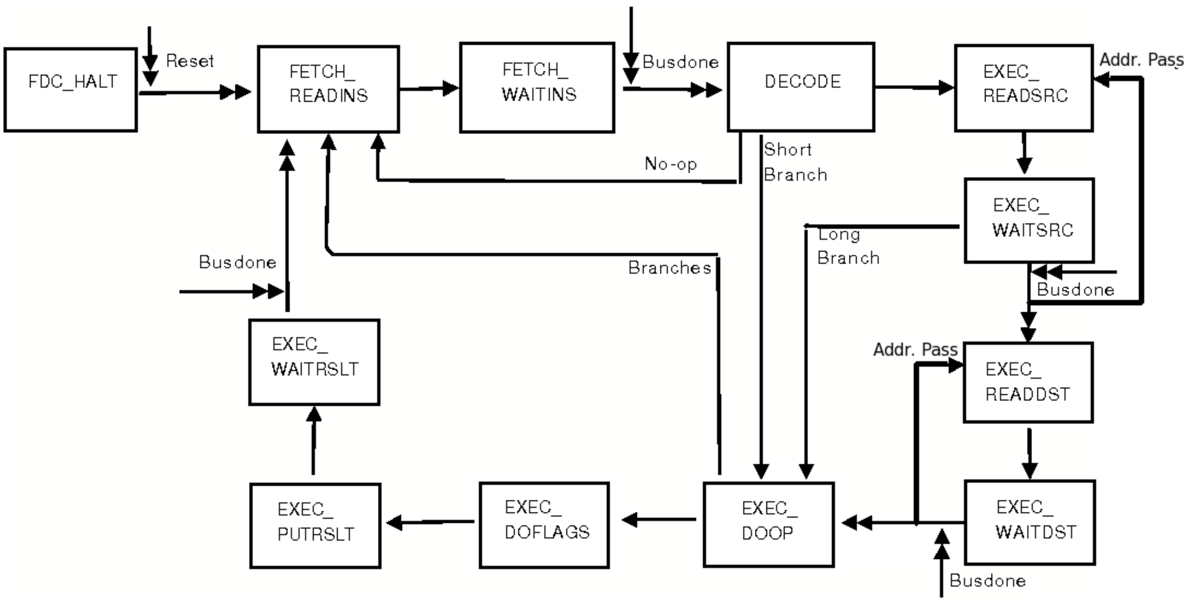
[9] Section 4.1 on the previous page.
[10] Section 3 on page 6.
[11] Section 6 on page 12.

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 sufficent to allow all addressing modes on the 68000 to be implemented.

4.6.1 Final Fetch-Decode-Execute Cycle States



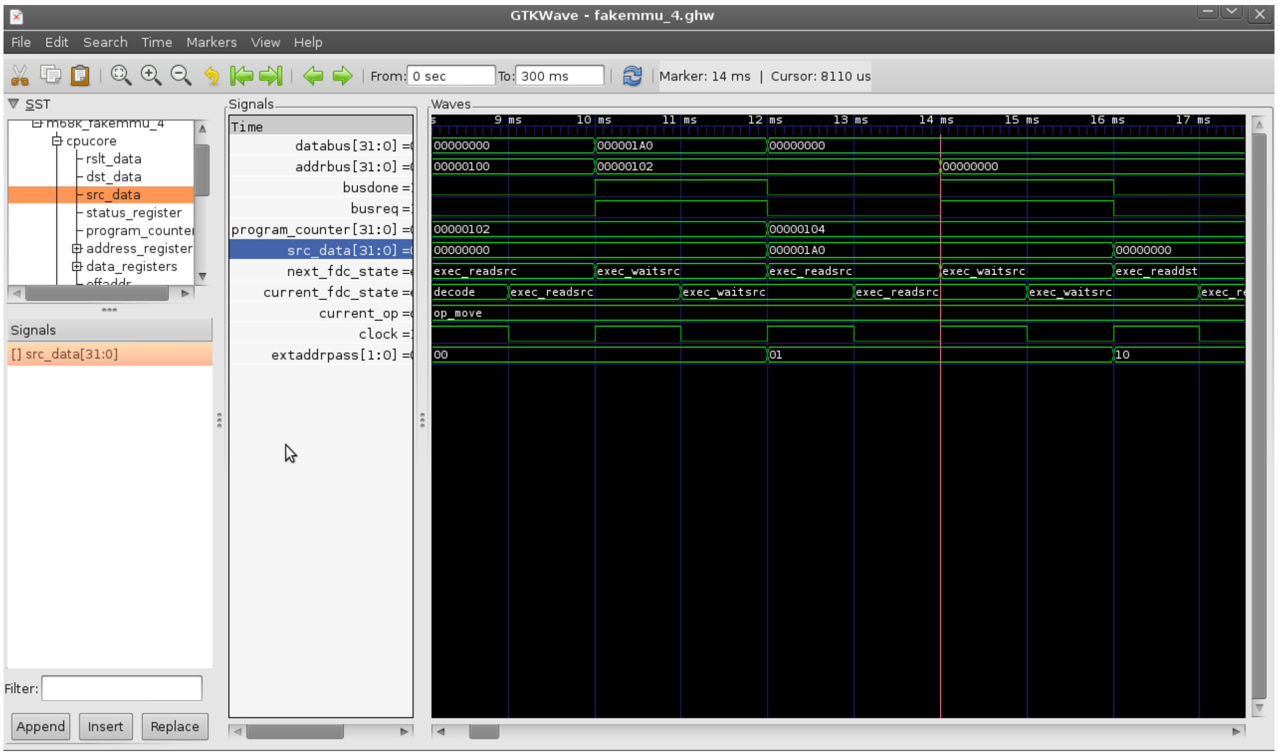
5 Implementation Notes

Implementation was straightforward, with a new test program developed[12] 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[13] 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 recieved, acknowledged and placed in the SRC_DATA register. Transferring SRC_DATA to the address bus during the second EXEC_READSRC pass fixed the error.

[12]See section 6.1.4 on the following page.

[13]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[14]), 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[15]; 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

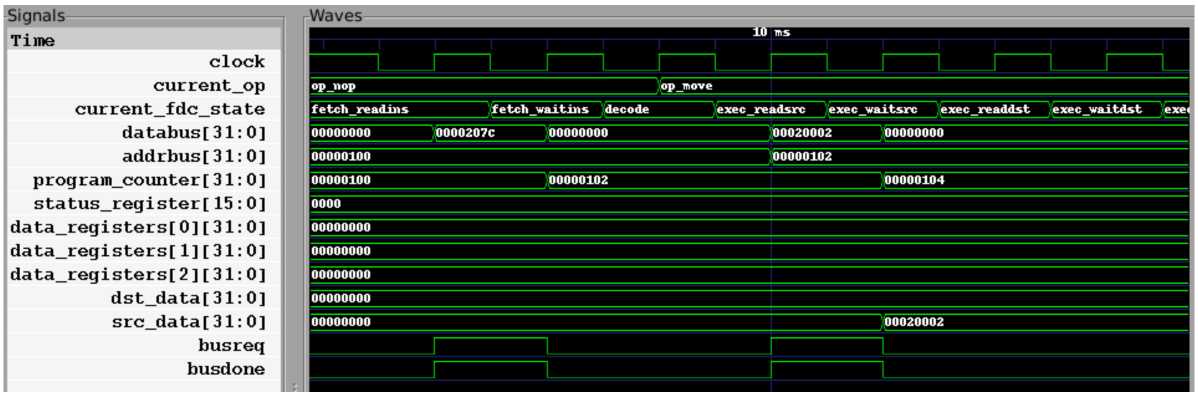
```
org $100
move.w $10A, d1
move.w $1BB, d2
bra $100
org $10A
dw $1111
org $1BB
dw $2222
end
```

[14]Section 1.2 on page 4.
[15]Section C on page 40.

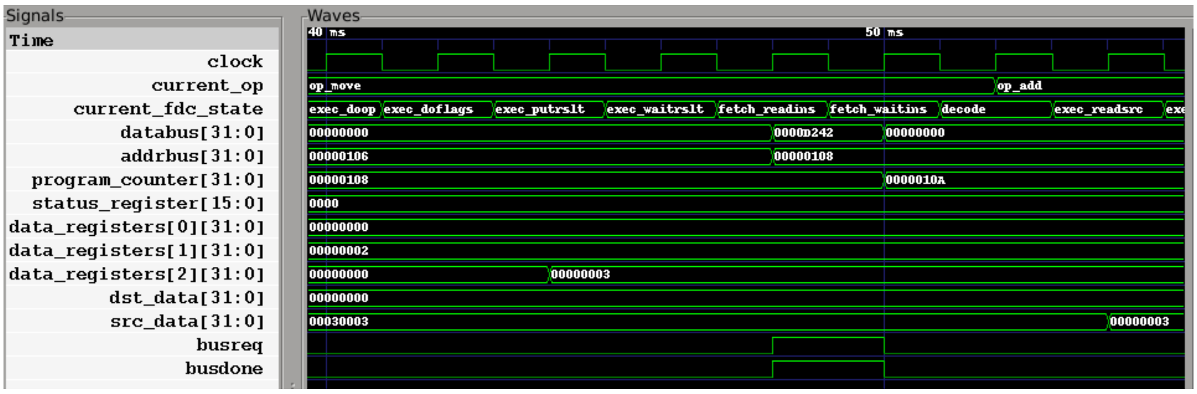
6.2 Simulation Waveforms

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.

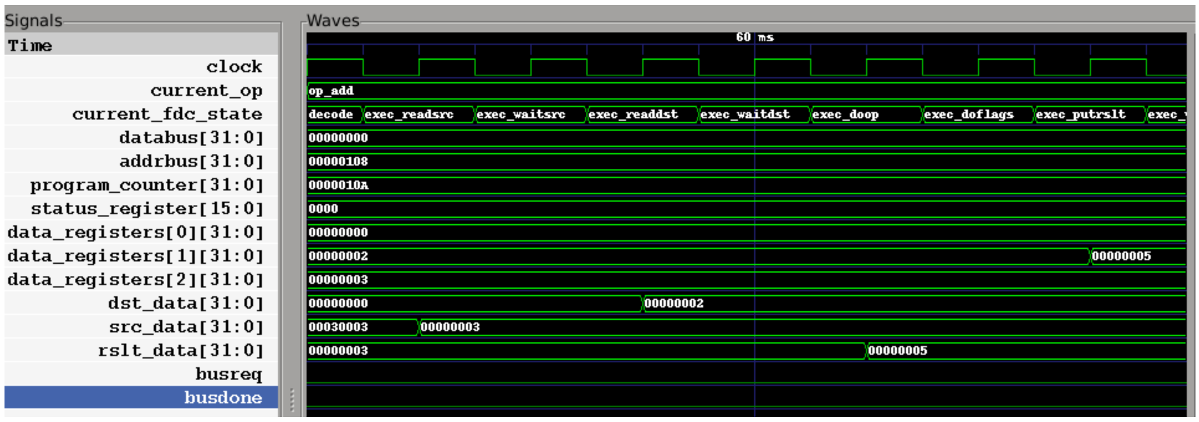
6.2.1 Move/Add/Branch Test



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.

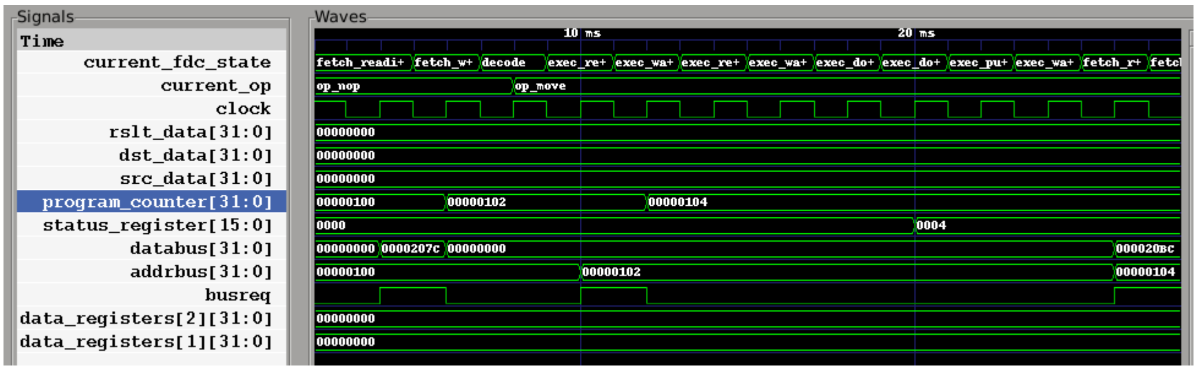


This screenshot shows the end of the move instruction. The data is written to the destination (in this case, data register 2) during the EXEC_PUTRSLT cycle. The next instruction is an add and we can see it decoded here. The operands are registers so we do not see any memory accesses until the source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

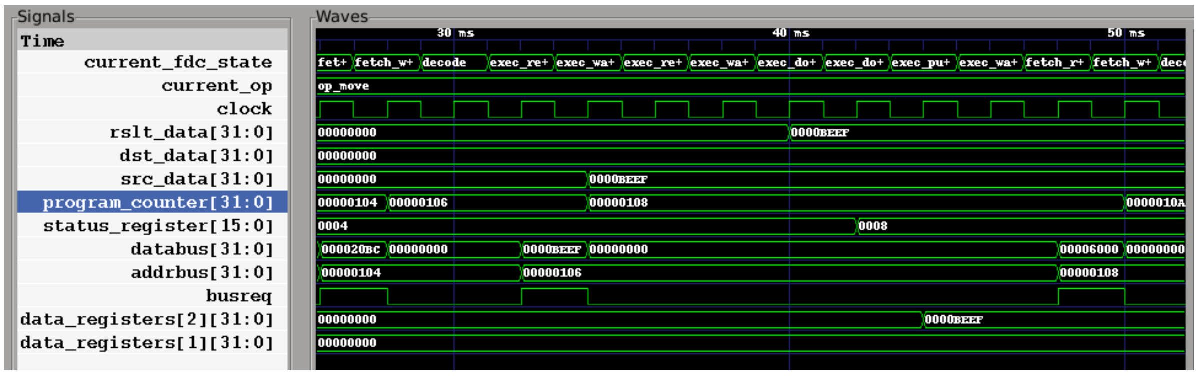


This screenshot shows the execution of the add instruction. The source and destination - both data registers - are read into the src_data and dst_data internal registers. During the EXEC_DOOP cycle these are added and the result is placed in the internal rslt_data register. This is saved to the destination during the EXEC_PUTRSLT cycle.

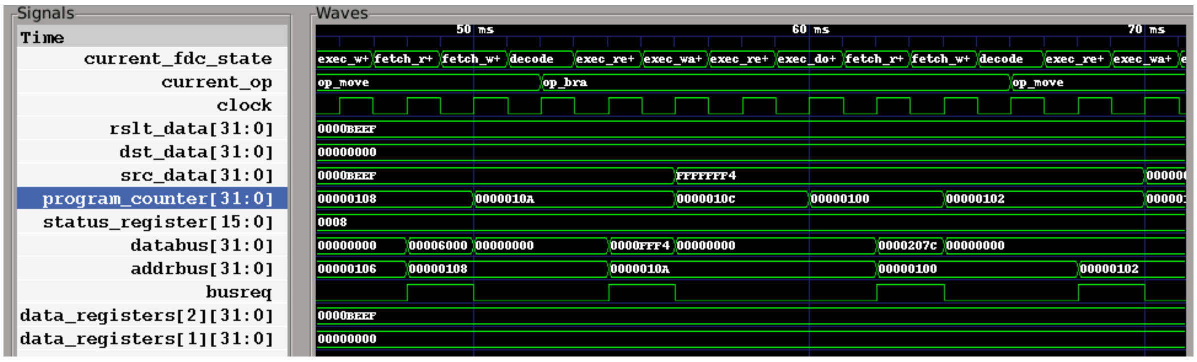
6.2.2 Move Flag and Branch Test



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.

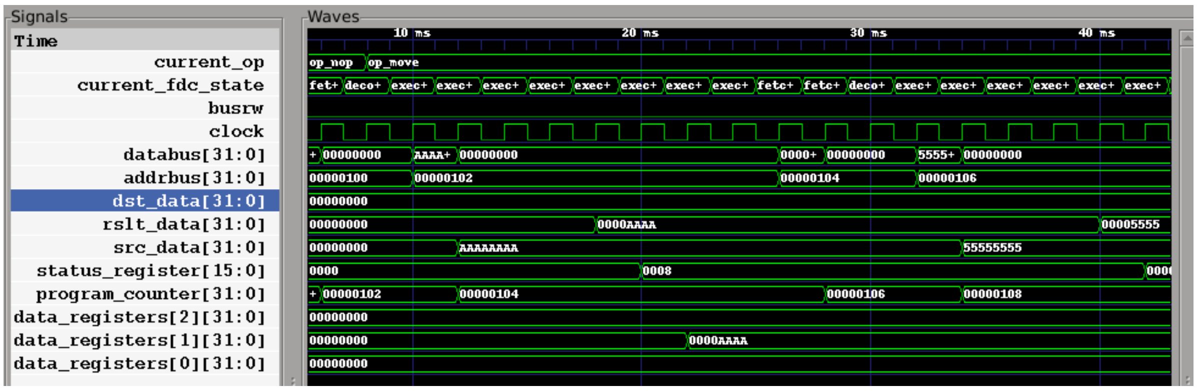


The second move, $BEEF to d2, is visible here. We can see the instruction being read, decoded, the immediate value being read and placed in the src_data internal register, and then in the rslt_data register. The status register changes from 4 to 8, indicating that now the negative flag is set - as a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

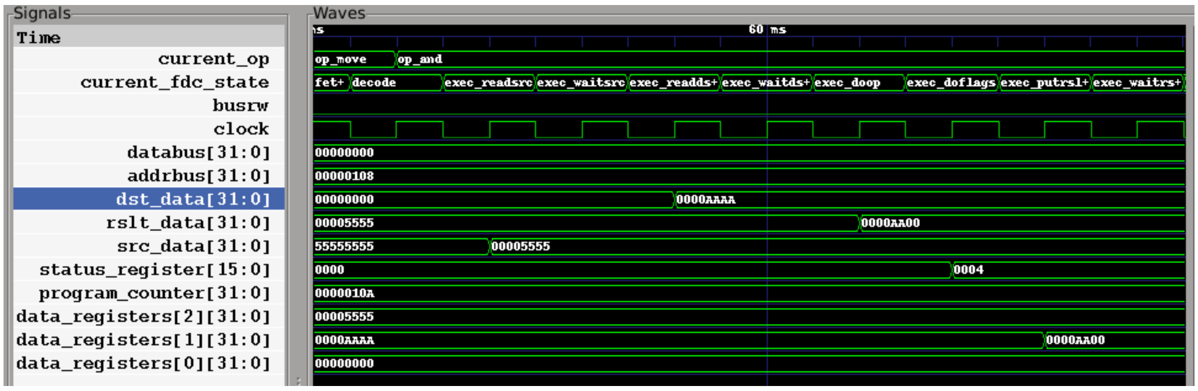


The full set of cycles for a long branch instruction can be seen here. After the instruction is decoded the next word is read ($FFF4 or -12 decimal). This is sign extended to 32 bits (the $FFFFFFF4 in src_data) and added to the program counter in the EXEC_DOOP state, returning it to the start of the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

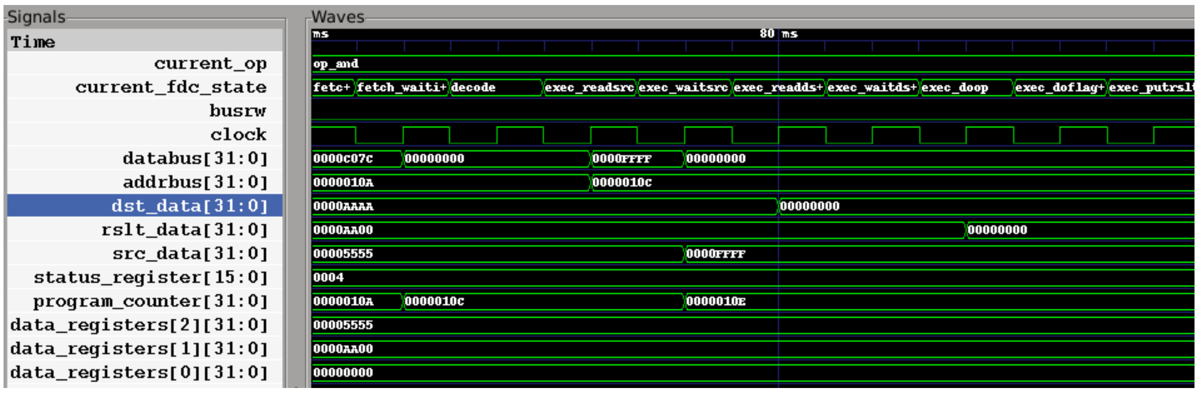
6.2.3 Move/And/Branch Test



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

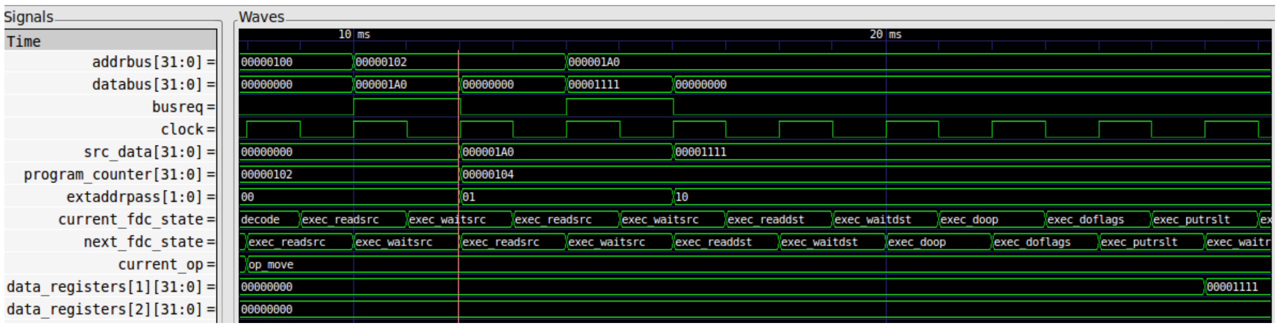


The next instruction is a *byte* size and operation on d1 and d2 - and 1010 with 0101. The lower byte is set to 0 and the zero flag is set.

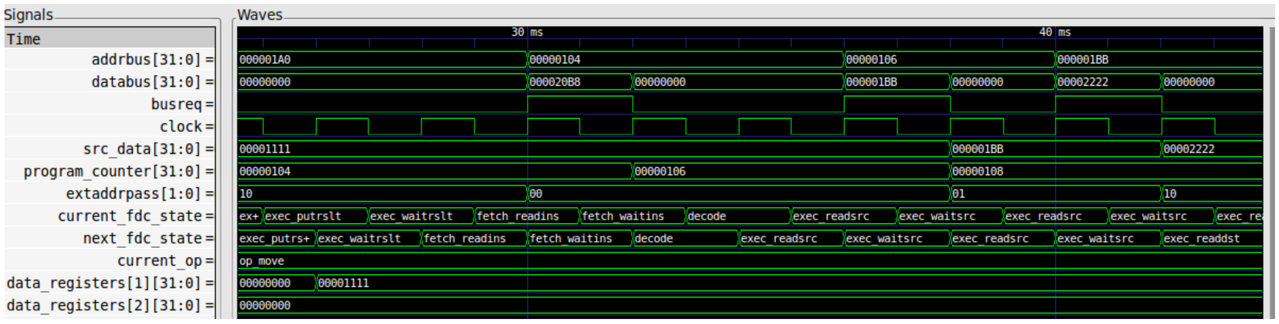


The next instruction is an and with immediate data ($FFFF) and d0 (0), again producing zero.

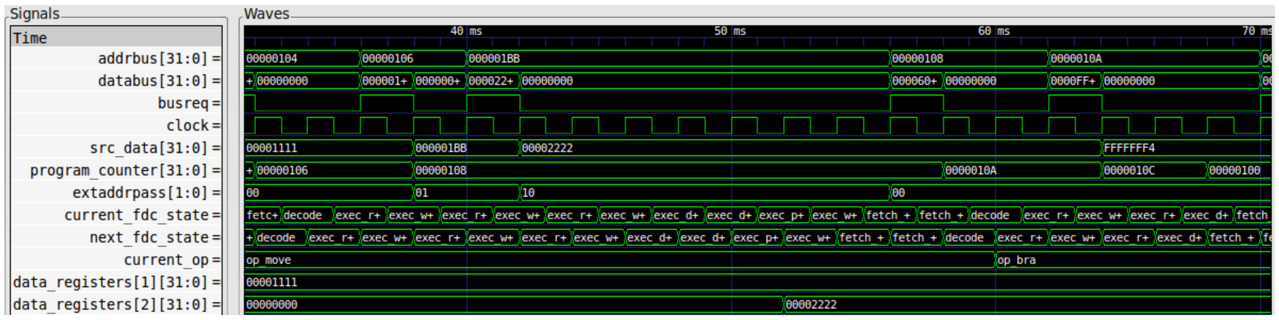
6.2.4 Absolute Addressing Test



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 [16]) 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

[16]Section 4.3 on page 9.

Part III
Appendices

A Appendix A: CPU Source Code

A.1 m68k_cpu_core.vhd

```vhdl
-- CPU core for 68k
-- Dylan Leigh s3017239

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_cpu_core is
  port (
    reset: in std_logic; -- active high
    clock: in std_logic;

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

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

    -- address bus
    addrbus: out std_logic_vector(31 downto 0);
    -- data bus
    databus: in std_logic_vector(31 downto 0);
    -- memory writes TODO databus: inout std_logic_vector(31 downto 0)

    -- for debugging
    flags: out std_logic_vector(4 downto 0)
  );
end m68k_cpu_core;

architecture behaviour of m68k_cpu_core is

  type FDC_State_Type is (FETCH_READINS, FETCH_WAITINS,
                          DECODE,
                          -- note source or destination may be immediate data,
                          -- which still has to be read from memory (and
                          -- waited on)
                          EXEC_READSRC, EXEC_WAITSRC, -- EXEC_PUTSRCALU,
                          EXEC_READDST, EXEC_WAITDST, -- EXEC_PUTDSTALU,
                          EXEC_DOOP, EXEC_DOFLAGS,
                          EXEC_PUTRSLT, EXEC_WAITRSLT,
                          FDC_HALT);
  signal current_fdc_state, next_fdc_state : FDC_State_Type;

  -- Note we are using "instruction" to refer to the whole thing
  -- including operands and "opcode" to refer to the opcode part.
  -- "Operation" is what the opcode represents
  -- internal use only
```

```
    type CPUOp is (OP_MOVE, OP_ADD, OP_AND, OP_BRA, OP_NOP);
    signal current_op: CPUOp;
    signal current_inst: std_logic_vector(15 downto 0);

    signal opdatasize: std_logic_vector(1 downto 0);

    -- effective address codes, either copied directly from the instruction or
    -- determined from it during decode
    -- internal use only
    signal src_addrcode: std_logic_vector(5 downto 0);
    signal dst_addrcode: std_logic_vector(5 downto 0);
    -- internal use only - set for the second memory access
    -- pass, for extended adressing TODO
    signal extaddrpass: std_logic_vector(1 downto 0);
    -- internal use only - effective address location
    signal effaddr: std_logic_vector(31 downto 0);

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

    -- internal registers
    signal src_data: std_logic_vector(31 downto 0);
    signal dst_data: std_logic_vector(31 downto 0);
    signal rslt_data: std_logic_vector(31 downto 0);

begin
    -- TODO component registers

    flags <= status_register(4 downto 0);

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

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

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

```
         addrbus <= x"00000100";
         -- TODO no mem writes databus <= x"00000000";
         busreq <= '0';
         busrw <= '0';
         busdatasize <= "00";
         panic <= '0';

      else -- set next state based on current state
         -- start huge current state statement of doom
         if rising_edge(clock) then
            case current_fdc_state is

               when FETCH_READINS =>
                  addrbus <= program_counter;
                  busrw <= '0'; --read
                  busdatasize <= "10"; --word size
                  busreq <= '1'; --request
                  next_fdc_state <= FETCH_WAITINS;
                  extaddrpass <= "00";

               when FETCH_WAITINS =>
                  if busdone = '1' then
                     program_counter <= std_logic_vector(
                                        unsigned(program_counter) + 2);
                     busreq <= '0'; --end request
                     current_inst <= databus(15 downto 0);
                     next_fdc_state <= DECODE;
                  end if; -- busdone

               when DECODE =>
                  -- start outer big decode case statement
                  -- Note - this is more complex than just one case statement on
                  -- one set of bits as some operations only have 2 unique bits,
                  -- and some have 8 unique bits
                  case current_inst(15 downto 12) is
                     -- TODO when "0000" => -- cmpi (immediate)
                        -- current_op <= OP_CMPI;
                        -- TODO decode effective addresses
                        -- next_fdc_state <= EXEC_READSRC;

                     when "0001" => -- move.b
                        current_op <= OP_MOVE;
                        opdatasize <= "01"; -- byte
                        -- decode effective addresses
                        src_addrcode <= current_inst(5 downto 0);
                        dst_addrcode <= current_inst(11 downto 6);
                        next_fdc_state <= EXEC_READSRC;

                     when "0010" => -- move.w
                        current_op <= OP_MOVE;
                        opdatasize <= "10"; -- word
                        -- decode effective addresses
                        src_addrcode <= current_inst(5 downto 0);
                        dst_addrcode <= current_inst(11 downto 6);
                        next_fdc_state <= EXEC_READSRC;

                     when "0011" => -- move.l
                        current_op <= OP_MOVE;
                        opdatasize <= "11"; -- long
                        -- decode effective addresses
                        src_addrcode <= current_inst(5 downto 0);
```

```
                dst_addrcode <= current_inst(11 downto 6);
                next_fdc_state <= EXEC_READSRC;

            when "0100" => -- nop
                if current_inst(11 downto 0) = "111001110001"
                then -- no-op
                    current_op <= OP_NOP;
                    next_fdc_state <= FETCH_READINS;
                else -- PANIC due to unimplemented opcode
                    next_fdc_state <= FDC_HALT;
                    panic <= '1';
                    report "CPU Panic - unimplemented opcode"
                        severity FAILURE;
                end if;

            when "0110" => -- branch
                if current_inst(11 downto 8) = "0000"
                then -- branch
                    current_op <= OP_BRA;
                    -- decode destination
                    if current_inst(7 downto 0) = x"00"
                    then -- 16 bit branch - read another word
                        src_addrcode <= "111100"; -- immediate
                        next_fdc_state <= EXEC_READSRC;
                    else -- 8 bit branch - handle here
                        if current_inst(7) = '1'
                        then -- sign extend
                            src_data <= x"FFFFFF" & current_inst(7 downto
                                0);
                        else
                            src_data <= x"000000" & current_inst(7 downto
                                0);
                        end if; -- sign extend
                        next_fdc_state <= EXEC_DOOP;
                    end if; -- 8 bit destination

                else -- PANIC due to unimplemented opcode
                    next_fdc_state <= FDC_HALT;
                    panic <= '1';
                    report "CPU Panic - unimplemented opcode"
                        severity FAILURE;
                end if;

            -- TODO when "1011" => -- cmp (not immediate)
                -- current_op <= OP_CMP;
                -- TODO decode effective addresses
                -- next_fdc_state <= EXEC_READSRC;

            when "1101" => -- add.b/w/l
                current_op <= OP_ADD;
                -- detemines source dest order of operands
                if (current_inst(8) = '1')
                then
                    next_fdc_state <= FDC_HALT;
                    panic <= '1';
                    report "CPU Panic - unimplemented adressing mode"
                        severity FAILURE;
                else
                    opdatasize <= current_inst(7 downto 6);
                    src_addrcode <= current_inst(5 downto 0);
                                -- dest is a data register
                        dst_addrcode <= "000" & current_inst(11 downto 9);
                     end if;
                     next_fdc_state <= EXEC_READSRC;

                  when "1100" => -- and.b/w/l
                     current_op <= OP_AND;
                     -- decode effective addresses
                     -- detemines source dest order of operands
                     if (current_inst(8) = '1')
                     then
                        next_fdc_state <= FDC_HALT;
                        panic <= '1';
                        report "CPU Panic - unimplemented adressing mode"
                           severity FAILURE;
                     else
                        opdatasize <= current_inst(7 downto 6);
                        src_addrcode <= current_inst(5 downto 0);
                                    -- dest is a data register
                        dst_addrcode <= "000" & current_inst(11 downto 9);
                     end if;
                     next_fdc_state <= EXEC_READSRC;

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

            when EXEC_READSRC => -- TODO direct memory
               case src_addrcode(5 downto 3) is
                  when "000" => -- data register
                     -- this should be replaced with an integer/unsigned
                     -- expression on the array index
                     case src_addrcode(2 downto 0) is
                        when "000" => src_data <= data_registers(0);
                        when "001" => src_data <= data_registers(1);
                        when "010" => src_data <= data_registers(2);
                        when "011" => src_data <= data_registers(3);
                        when "100" => src_data <= data_registers(4);
                        when "101" => src_data <= data_registers(5);
                        when "110" => src_data <= data_registers(6);
                        when "111" => src_data <= data_registers(7);
                     end case; -- src_addrcode register section
                     next_fdc_state <= EXEC_WAITSRC;

                  when "001" => -- address register
                     -- this should be replaced with an integer/unsigned
                     -- expression on the array index
                     case src_addrcode(2 downto 0) is
                        when "000" => src_data <= address_registers(0);
                        when "001" => src_data <= address_registers(1);
                        when "010" => src_data <= address_registers(2);
                        when "011" => src_data <= address_registers(3);
                        when "100" => src_data <= address_registers(4);
                        when "101" => src_data <= address_registers(5);
                        when "110" => src_data <= address_registers(6);
                        when "111" => src_data <= address_registers(7);
                     end case; -- src_addrcode register section
                     next_fdc_state <= EXEC_WAITSRC;
```

```
when "111" => -- could be immediate,absolute,offset
   case src_addrcode(2 downto 0) is

      when "000" => -- word addr absolute
         case extaddrpass is
            when "00" =>
               addrbus <= program_counter;
               busrw <= '0'; --read
               -- note we don't use opdatasize as that is
               -- the size of the final value we want not
               -- the address it is at.
               busdatasize <= "10";
               busreq <= '1'; --request
               next_fdc_state <= EXEC_WAITSRC;

            when "01" => -- pass 2
               -- read from location we just got
               addrbus <= src_data;

               busrw <= '0'; --read
               if opdatasize = "11"
               then -- long
                  busdatasize <= "11";
               else  --word size, byte is in lower half
                  busdatasize <= "10";
               end if; --datasize
               busreq <= '1'; --request
               next_fdc_state <= EXEC_WAITSRC;

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

      when "001" => -- long addr absolute
         case extaddrpass is
            when "00" =>
               addrbus <= program_counter;
               busrw <= '0'; --read
               -- note we don't use opdatasize as that is
               -- the size of the final valjue we want not
               -- the address it is at.
               busdatasize <= "11";
               busreq <= '1'; --request
               next_fdc_state <= EXEC_WAITSRC;

            when "01" => -- pass 2
               -- read from location we just got
               addrbus <= src_data;

               busrw <= '0'; --read
               if opdatasize = "11"
               then -- long
                  busdatasize <= "11";
               else  --word size, byte is in lower half
                  busdatasize <= "10";
               end if; --datasize
               busreq <= '1'; --request
```

```
                                 next_fdc_state <= EXEC_WAITSRC;

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

                        when "100" => -- immediate
                           addrbus <= program_counter;
                           busrw <= '0'; --read
                           if opdatasize = "11"
                           then -- long
                              busdatasize <= "11";
                           else --word size, byte is in lower half
                              busdatasize <= "10";
                           end if; --datasize
                           busreq <= '1'; --request
                           next_fdc_state <= EXEC_WAITSRC;
                        -- end when immediate

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

                  when others =>
                     next_fdc_state <= FDC_HALT;
                     panic <= '1';
                     report "CPU Panic - unimplemented addressing mode"
                        severity FAILURE;
               end case; -- src_addrcode 5 downto 3

            when EXEC_WAITSRC => -- TODO direct memory
               case src_addrcode(5 downto 3) is
                  when "000" => -- data register
                     -- TODO this is necessary for the clocked registers
                         later
                     next_fdc_state <= EXEC_READDST;
                     -- TODO seperate ALU next_fdc_state <= EXEC_PUTSRCALU;
                  when "001" => -- address register clock
                     -- TODO this is necessary for the clocked registers
                         later
                     next_fdc_state <= EXEC_READDST;
                     -- TODO seperate ALU next_fdc_state <= EXEC_PUTSRCALU;

                  when "111" => -- immediate or direct
                     if busdone = '1'
                     then
                        busreq <= '0'; --end request
                        case src_addrcode(2 downto 0) is
                           when "000" => -- word absolute address
                              case extaddrpass is
                                 when "00" => -- FIXME test
                                    program_counter <= std_logic_vector(
                                          unsigned(program_counter) + 2);
                                    src_data <= x"0000" & databus(15 downto
                                       0);
```

```
                                          -- set to 01 here - we've made 1 pass
                                          extaddrpass <= "01";

                                          next_fdc_state <= EXEC_READSRC;

                                       when "01" => -- FIXME test
                                          if current_op = OP_BRA
                                          then
                                             -- sign extend
                                             src_data <= x"FFFF" & databus(15
                                                downto 0);
                                          else
                                             src_data <= databus; -- regardless of
                                                data size
                                                                   -- both of these
                                                                      are 32 bits
                                          end if; -- current op is branch
                                          extaddrpass <= "10"; -- 2nd pass done
                                          next_fdc_state <= EXEC_READDST;
                                          -- TODO sep ALU next_fdc_state <=
                                             EXEC_PUTSRCALU;

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

                                 when "001" => -- long absolute address TODO
                                    program_counter <= std_logic_vector(
                                          unsigned(program_counter) + 4);
                                    -- TODO
                                    next_fdc_state <= FDC_HALT;
                                    panic <= '1';
                                    report "CPU Panic - unimplemented addressing
                                       mode"
                                       severity FAILURE;

                                 when "100" => -- immediate data
                                    if opdatasize = "11"
                                    then -- long
                                       program_counter <= std_logic_vector(
                                             unsigned(program_counter) + 4);
                                    else  --word size, byte is in lower half
                                       program_counter <= std_logic_vector(
                                             unsigned(program_counter) + 2);
                                    end if; --datasize

                                    if current_op = OP_BRA
                                    then
                                       -- sign extend
                                       src_data <= x"FFFF" & databus(15 downto 0);
                                    else
                                       src_data <= databus; -- regardless of data
                                          size
                                                             -- both of these are 32
                                                                bits
                                    end if; -- current op is branch
                                    next_fdc_state <= EXEC_READDST;
                                    -- TODO sep ALU next_fdc_state <=
                                       EXEC_PUTSRCALU;

                                 when others =>
                                    next_fdc_state <= FDC_HALT;
                                    panic <= '1';
                                    report "CPU Panic - unimplemented addressing
                                       mode"
                                       severity FAILURE;
                              end case; --immediate or direct
                           end if; -- busdone

                        when others =>
                           next_fdc_state <= FDC_HALT;
                           panic <= '1';
                           report  "CPU Panic - unimplemented addressing mode"
                              severity FAILURE;
                     end case; --src_addrcode

                  when EXEC_READDST => -- TODO direct memory
                     case current_op is
                        when OP_BRA =>
                           -- Instruction has only one operand
                           -- Note that for branh, we already skip this state
                           next_fdc_state <= EXEC_DOOP;
                        when others =>
                           case dst_addrcode(5 downto 3) is
                              when "000" => -- data register
                                 -- this should be replaced with an integer/
                                    unsigned
                                 -- expression on the array index
                                 case dst_addrcode(2 downto 0) is
                                    when "000" => dst_data <= data_registers(0);
                                    when "001" => dst_data <= data_registers(1);
                                    when "010" => dst_data <= data_registers(2);
                                    when "011" => dst_data <= data_registers(3);
                                    when "100" => dst_data <= data_registers(4);
                                    when "101" => dst_data <= data_registers(5);
                                    when "110" => dst_data <= data_registers(6);
                                    when "111" => dst_data <= data_registers(7);
                                 end case; -- dst_addrcode register section
                                 next_fdc_state <= EXEC_WAITDST;

                              when "001" => -- address register
                                 -- this should be replaced with an integer/
                                    unsigned
                                 -- expression on the array index
                                 case dst_addrcode(2 downto 0) is
                                    when "000" => dst_data <= address_registers(0);
                                    when "001" => dst_data <= address_registers(1);
                                    when "010" => dst_data <= address_registers(2);
                                    when "011" => dst_data <= address_registers(3);
                                    when "100" => dst_data <= address_registers(4);
                                    when "101" => dst_data <= address_registers(5);
                                    when "110" => dst_data <= address_registers(6);
                                    when "111" => dst_data <= address_registers(7);
                                 end case; -- dst_addrcode register section
                                 next_fdc_state <= EXEC_WAITDST;

                              when "111" => -- could be immediate or direct
                                 if src_addrcode(2 downto 0) = "100"
                                 then -- immediate data invalid as destination
                                    next_fdc_state <= FDC_HALT;
                                    panic <= '1';
                                    report "CPU Panic - immediate addressing used
                                       as dest."
                                       severity FAILURE;
                                 else
                                    next_fdc_state <= FDC_HALT;
                                    panic <= '1';
                                    report  "CPU Panic - unimplemented addressing
                                       mode"
                                       severity FAILURE;
                                 end if; -- when 111

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

                  when EXEC_WAITDST => -- TODO direct memory
                     case dst_addrcode(5 downto 3) is
                        when "000" => -- data register
                           -- TODO this is necessary for the clocked registers
                              later
                           next_fdc_state <= EXEC_DOOP;
                           -- TODO seperate ALU next_fdc_state <= EXEC_PUTDSTALU;
                        when "001" => -- address register clock
                           -- TODO this is necessary for the clocked registers
                              later
                           next_fdc_state <= EXEC_DOOP;
                           -- TODO seperate ALU next_fdc_state <= EXEC_PUTDSTALU;
                        when others =>
                           next_fdc_state <= FDC_HALT;
                           panic <= '1';
                           report  "CPU Panic - unimplemented addressing mode"
                              severity FAILURE;
                      end case; --src_addrcode

                  when EXEC_DOOP =>
                     case current_op is
                        when OP_MOVE =>
                           case opdatasize is
                              when "11" => -- long
                                 rslt_data <= src_data;
                              when "10" => -- word
                                 rslt_data <= dst_data(31 downto 16) &
                                              src_data(15 downto 0);
                              when "01" => -- byte
                                 rslt_data <= dst_data(31 downto 8) &
                                              src_data(7 downto 0);
                              when others =>
                                 next_fdc_state <= FDC_HALT;
                                 panic <= '1';
                                 report  "CPU Panic - invalid operation data size"
                                    severity FAILURE;
                           end case;
                           next_fdc_state <= EXEC_DOFLAGS;
```

```

      when OP_BRA =>
         program_counter <= std_logic_vector(
                            unsigned(program_counter)
                            + unsigned(src_data));
         -- Go direct to to next instruction, no flags change
         next_fdc_state <= FETCH_READINS;

      when OP_ADD =>
         case opdatasize is
            when "11" => -- long
               rslt_data <= std_logic_vector(unsigned(src_data)
                                              + unsigned(dst_data));
            when "10" => -- word
               rslt_data <= dst_data(31 downto 16) &
                            std_logic_vector(
                               unsigned(src_data(15 downto 0))
                             + unsigned(dst_data(15 downto 0)));
            when "01" => -- byte
               rslt_data <= dst_data(31 downto 8) &
                            std_logic_vector(
                               unsigned(src_data(7 downto 0))
                             + unsigned(dst_data(7 downto 0)));
            when others =>
               next_fdc_state <= FDC_HALT;
               panic <= '1';
               report "CPU Panic - invalid operation data size"
                  severity FAILURE;
         end case;
         -- TODO: this needs to be done in compnent ALU which
         -- also determines carry/overflow flags
         next_fdc_state <= EXEC_DOFLAGS;

      when OP_AND =>
         case opdatasize is
            when "11" => -- long
               rslt_data <= src_data and dst_data;
            when "10" => -- word
               rslt_data <= dst_data(31 downto 16) &
                             (src_data(15 downto 0) and
                              dst_data(15 downto 0));
            when "01" => -- byte
               rslt_data <= dst_data(31 downto 8) &
                             (src_data(7 downto 0) and
                              dst_data(7 downto 0));
            when others =>
               next_fdc_state <= FDC_HALT;
               panic <= '1';
               report "CPU Panic - invalid operation data size"
                  severity FAILURE;
         end case;
         next_fdc_state <= EXEC_DOFLAGS;

      when others =>
         next_fdc_state <= FDC_HALT;
         panic <= '1';
         report "CPU Panic - unimplemented operation"
            severity FAILURE;
   end case;

when EXEC_DOFLAGS =>
```

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

                    case current_op is -- negative/zero - nearly all ops
                        when OP_MOVE | OP_AND | OP_ADD =>
                            case opdatasize is
                                when "01" => -- byte
                                    if rslt_data(7 downto 0) = x"00"
                                    then --zero
                                        status_register(2) <= '1'; --zero
                                        status_register(3) <= '0'; --negative
                                    else
                                        status_register(2) <= '0'; --zero
                                        if rslt_data(7) = '1'
                                        then
                                            status_register(3) <= '1'; --negative
                                        else
                                            status_register(3) <= '0'; --negative
                                        end if; -- negative
                                    end if; -- zero
                                when "10" => -- word
                                    if rslt_data(15 downto 0) = x"0000"
                                    then --zero
                                        status_register(2) <= '1'; --zero
                                        status_register(3) <= '0'; --negative
                                    else
                                        status_register(2) <= '0'; --zero
                                        if rslt_data(15) = '1'
                                        then
                                            status_register(3) <= '1'; --negative
                                        else
                                            status_register(3) <= '0'; --negative
                                        end if; -- negative
                                    end if; -- zero
                                when "11" => -- long
                                    if rslt_data(31 downto 0) = x"00000000"
                                    then --zero
                                        status_register(2) <= '1'; --zero
                                        status_register(3) <= '0'; --negative
                                    else
                                        status_register(2) <= '0'; --zero
                                        if rslt_data(31) = '1'
                                        then
                                            status_register(3) <= '1'; --negative
                                        else
                                            status_register(3) <= '0'; --negative
                                        end if; -- negative
                                    end if; -- zero
                        when others =>
                           next_fdc_state <= FDC_HALT;
                           panic <= '1';
                           report "CPU Panic - invalid operation data size"
                              severity FAILURE;
                     end case; -- datasize for n/z flags
                     next_fdc_state <= EXEC_PUTRSLT;

                  when others =>
                     next_fdc_state <= FDC_HALT;
                     panic <= '1';
                     report "No N/Z flags coded for op, possible state error
                        "
                        severity WARNING;
               end case; -- current_op for n/z

            when EXEC_PUTRSLT => -- TODO direct memory
               case dst_addrcode(5 downto 3) is
                  when "000" => -- data register
                     -- this should be replaced with an integer/unsigned
                     -- expression on the array index
                     case dst_addrcode(2 downto 0) is
                        when "000" => data_registers(0) <= rslt_data;
                        when "001" => data_registers(1) <= rslt_data;
                        when "010" => data_registers(2) <= rslt_data;
                        when "011" => data_registers(3) <= rslt_data;
                        when "100" => data_registers(4) <= rslt_data;
                        when "101" => data_registers(5) <= rslt_data;
                        when "110" => data_registers(6) <= rslt_data;
                        when "111" => data_registers(7) <= rslt_data;
                     end case; -- dst_addrcode register section
                     next_fdc_state <= EXEC_WAITRSLT;

                  when "001" => -- address register
                     -- this should be replaced with an integer/unsigned
                     -- expression on the array index
                     case dst_addrcode(2 downto 0) is
                        when "000" => address_registers(0) <= rslt_data;
                        when "001" => address_registers(1) <= rslt_data;
                        when "010" => address_registers(2) <= rslt_data;
                        when "011" => address_registers(3) <= rslt_data;
                        when "100" => address_registers(4) <= rslt_data;
                        when "101" => address_registers(5) <= rslt_data;
                        when "110" => address_registers(6) <= rslt_data;
                        when "111" => address_registers(7) <= rslt_data;
                     end case; -- dst_addrcode register section
                     next_fdc_state <= EXEC_WAITRSLT;

                  when others =>
                     next_fdc_state <= FDC_HALT;
                     panic <= '1';
                     report "CPU Panic - unimplemented addressing mode"
                        severity FAILURE;
               end case; -- src_addrcode

            when EXEC_WAITRSLT => -- TODO direct memory
               case dst_addrcode(5 downto 3) is
                  when "000" => -- data register
                     -- TODO this is necessary for the clocked registers
                        later
                     next_fdc_state <= FETCH_READINS;
```

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

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

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

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

B Appendix B: DE1 Interface Source Code

B.1 m68k_de1.vhd

```
-- DE1 board <-> 68k cpu interface
-- Dylan Leigh s3017239

-- Controls:
--
-- SW 9-8: select program:
--          00: move/add/branch test
--          01: move flags test
--          10: move/and/branch test
--          11: all memory is a nop
--
-- SW 0-7: Binary clock speed selector
--         selects divider for auto clock
--
-- KEY3: Reset
-- KEY2: Hold for auto clock
-- KEY0: Clock (for manual clock)
--
-- Outputs:
-- HEX: Lower bytes of address and data busses

-- LEDR9: panic (cpu halts until reset)
-- LEDR2-8: UNUSED
-- LEDR1 - bus write (should not be lit inside any sample programs)
-- LEDR0 - Bus request (should be same as busdone)

-- LEDG7 - Bus done (should be same as bus request)
-- LEDG5-6: Bus data size (should be 10 in the sample programs)
-- LEDG0-4: SR Flags

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_de1 is
    port (
    -- All off the board
    KEY: in std_logic_vector(3 downto 0);
    SW: in std_logic_vector(9 downto 0);

    -- have to use "0 downto 0" as default pinout uses vectors for all
    CLOCK_50: in std_logic;

    HEX0: out std_logic_vector(6 downto 0);
    HEX1: out std_logic_vector(6 downto 0);
    HEX2: out std_logic_vector(6 downto 0);
    HEX3: out std_logic_vector(6 downto 0);

    LEDG: out std_logic_vector(7 downto 0);
    LEDR: out std_logic_vector(9 downto 0)
      );
end m68k_de1;

architecture mixed of m68k_de1 is
  -- logic for displaying 4 bits as a hex digit on the 7 segment display.
  component hex7seg
      port
      (
          inbits: in std_logic_vector(3 downto 0);
          hexout: out std_logic_vector(0 to 6)
      );
  end component;
```

```
    -- clock divider
    component clockdiv is
     Port (   rst  : in std_logic;
              clk  : in std_logic;
              clkout : out std_logic;
              speed : in std_logic_vector (2 downto 0)
         );
    end component;

    -- cpu
     component m68k_cpu_core is
        port (
           reset: in std_logic; -- active high
           clock: in std_logic;

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

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

           -- address bus
           addrbus: out std_logic_vector(31 downto 0);
           -- data bus
           databus: in std_logic_vector(31 downto 0);
           -- memory writes TODO databus: inout std_logic_vector(31 downto 0)

           -- outputs for debugging purposes
           flags: out std_logic_vector(4 downto 0)
        );
    end component;

     signal clock: std_logic;
     signal busrw: std_logic;
     signal busreq: std_logic;
     signal busdone: std_logic;
     signal busdatasize: std_logic_vector(1 downto 0);
     signal addrbus: std_logic_vector(31 downto 0);
     signal databus: std_logic_vector(31 downto 0);
     signal flags: std_logic_vector(4 downto 0);

     signal divclk: std_logic;

begin
     cpu: m68k_cpu_core port map (
           reset => not KEY(3), clock => clock, panic => LEDR(9),
           busrw => busrw, busreq => busreq, busdone => busdone,
           busdatasize => busdatasize,
           addrbus => addrbus, databus => databus, flags => flags);

     -- clock divider
     div: clockdiv port map (rst => not KEY(3), clk => CLOCK_50,
                                 clkout => divclk, speed => SW(3 downto 1));

     -- board stuff
     -- clock manually on key0 or continuously when key1 pressed
     clock <= (SW(0) and divclk) or (not KEY(0));
     -- LEDR(7) <= divclk; -- distracting
     LEDR(4) <= clock;
     LEDR(1) <= busrw;
     LEDR(0) <= busreq;
     LEDG(7) <= busdone;
     LEDG(4 downto 0) <= flags;
     LEDG(6 downto 5) <= busdatasize;
     h0: hex7seg port map (databus(3 downto 0), HEX0);
     h1: hex7seg port map (databus(7 downto 4), HEX1);
     h2: hex7seg port map (addrbus(3 downto 0), HEX2);
     h3: hex7seg port map (addrbus(7 downto 4), HEX3);

     -- memory read responses
     process (clock)
     begin
         if rising_edge(clock) then
             if (busreq = '1') then
                 -- this program is all reads, word size
                 assert (busrw = '0')
                     report "CPU requested a write"
                     severity WARNING;
                  assert (busdatasize = "10")
                     report "Bus data size request not 16 bits"
                     severity WARNING;

                  case SW(9 downto 8) is -- select program
                      when "00" => -- from fakemmu_1
                        -- memory reads
                        case addrbus is
                           when x"00000100" => -- move.w #02, d1
                              databus <= "00000000000000000010000001111100";
                                                         --   ^ ^ ^      ^ immediate
                                                         --   | | +-- data reg d1
                                                         --   | +-- word size
                                                         --   +-- move
                           when x"00000102" =>
                               databus <= x"00020002"; -- the #$20002

                           when x"00000104" => -- move.w #03, d2
                              databus <= "00000000000000000010000010111100";
                           when x"00000106" =>
                              databus <= x"00030003"; -- the #$30003

                         -- add together
                           when x"00000108" => -- add d2 to d1
                               databus <= "00000000000000001101001001000010";
                                                         --    ^   ^  ^^ ^source data reg 2
                                                         --    |   |  |+- word length
                                                         --    |   |  +- destination is data
                                                               reg
                                                         --    |    +- which data register
                                                         --    +- add

                           when x"0000010A" => -- add #5 to d0
                               databus <= "00000000000000001101000001111100";
                                                         --    ^   ^  ^^ ^immediate
                                                         --    |   |  |+- word length
                                                         --    |   |  +- destination is data
                                                               reg
                                                         --    |    +- which data register
                                                         --    +- add

                           when x"0000010C" => -- the #$50005
                              databus <= x"00050005"; -- the #$50005

                           when x"0000010E" => -- bra $100
                               databus <= "00000000000000000110000011110000";
                                                          --    ^       ^-- offset
                                                          --    +-- branch
                               report "Branching to start..." severity NOTE;
                     when others =>
                         report "Memory request from unhandled location"
                         severity WARNING;
                 end case;
             when "01" => -- from fakemmu_2
                 case addrbus is
                     when x"00000100" => -- move.w #0, d1
                         databus <= "00000000000000000010000001111100";
                                         --  ^ ^ ^     ^ immediate
                                         --  | | +-- data reg d1
                                         --  | +-- word size
                                         --  +-- move
                     when x"00000102" =>
                          databus <= x"00000000"; -- the #0

                     when x"00000104" => -- move.w #$DEADBEEF, d2
                         databus <= "00000000000000000010000010111100";
                     when x"00000106" =>
                         databus <= x"DEADBEEF"; -- the immediate data

                   -- more to come here when more opcodes implemented
                   -- add and save in d0

                     when x"00000108" => -- bra $100
                          databus <= "00000000000000000011000000000000";
                                          --   ^        ^-- offset all 0
                                          --   +-- branch
                     when x"0000010A" => -- bra $100
                          databus <= x"0000FFF4"; -- -12 decimal in word
                          report "Branching to start..." severity NOTE;

                     when others =>
                          report "Memory request from unhandled location"
                          severity WARNING;
                 end case;
             when "10" => -- fakemmu_3
                 case addrbus is
                     when x"00000100" => -- move.w 10 repeating, d1
                         databus <= "00000000000000000010000001111100";
                                         --  ^ ^ ^     ^ immediate
                                         --  | | +-- data reg d1
                                         --  | +-- word size
                                         --  +-- move
                     when x"00000102" =>
                         databus <= "10101010101010101010101010101010";

                     when x"00000104" => -- move.w 01 repeating, d2
                         databus <= "00000000000000000010000010111100";
                     when x"00000106" =>
                         databus <= "01010101010101010101010101010101";

                   -- and together
                     when x"00000108" => -- and d2 to d1
                          databus <= "00000000000000001100001001000010";
                                         --   ^    ^   ^^ ^source data reg 2
                                         --   |    |   |+- word length
                                         --   |    |   +- destination is data
                                              reg
                                         --   |     +- which data register
                                         --   +- and

                     when x"0000010A" => -- and 1all to d0
                          databus <= "00000000000000001100000001111100";
                                         --   ^    ^   ^^ ^immediate
                                         --   |    |   |+- word length
```

```
                                                     --    |    |  +- destination is data
                                                              reg
                                                     --    |   +- which data register
                                                     --   +- and

                                  when x"0000010C" =>
                                      databus <= x"0000FFFF"; -- the #$FFFF

                                  when x"0000010E" => -- bra $100
                                       databus <= "00000000000000000110000011110000";
                                                      --    ^        ^-- offset
                                                      --   +-- branch
                                       report "Branching to start..." severity NOTE;

                                  when others =>
                                       report "Memory request from unhandled location"
                                       severity WARNING;
                              end case;
                           when "11" => -- always nop FIXME
                                      databus <= "00000000000000000100111001110001";
                      end case; --
                      busdone <= '1';
                else -- no busreq
                      busdone <= '0';
                      databus <= x"00000000";
                end if;  --busreq
           end if;  --clock rising edge
     end process;
end mixed;
```

B.2 hex7seg.vhd

```
-- vim: sw=4 ts=4 et

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
    PORT
    (
        inbits: in std_logic_vector(3 downto 0);
        hexout: out std_logic_vector(0 to 6)
    );
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
    WITH inbits SELECT
        hexout <= "1000000" WHEN "0000", -- 0
                  "1111001" WHEN "0001", -- 1
                  "0100100" WHEN "0010", -- 2
                  "0110000" WHEN "0011", -- 3
                  "0011001" WHEN "0100", -- 4
                  "0010010" WHEN "0101", -- 5
                  "0000010" WHEN "0110", -- 6
                  "1111000" WHEN "0111", -- 7
                  "0000000" WHEN "1000", -- 8
                  "0011000" WHEN "1001", -- 9
                  "0001000" WHEN "1010", -- a
                  "0000011" WHEN "1011", -- b
                  "1000110" WHEN "1100", -- c
                  "0100001" WHEN "1101", -- d
                  "0000110" WHEN "1110", -- e
                  "0001110" WHEN "1111", -- f
                  "1111111" WHEN OTHERS;
END Behavior;
```

B.3 clockdiv.vhd

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

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL; -- Two very useful
use IEEE.STD_LOGIC_UNSIGNED.ALL; -- IEEE libraries

entity clockdiv is
    Port (  rst : in std_logic;
            clk : in std_logic;
            clkout : out std_logic;
            speed : in std_logic_vector (2 downto 0)
        );
end clockdiv;


architecture behavioral of clockdiv is
    signal temp: std_logic_vector(25 downto 0);
begin
    process (clk, rst)
    begin
        if (rst = '1') then
            temp <= "00000000000000000000000000";
        elsif rising_edge(clk) then
            temp <= temp + 1;
        end if;
        case speed is
            when "000" =>
                if temp(25) = '1' then
                    clkout <= '1';
                else
                    clkout <= '0';
                end if;
            when "001" =>
                if temp(24) = '1' then
                    clkout <= '1';
                else
                    clkout <= '0';
                end if;
            when "010" =>
                if temp(23) = '1' then
                    clkout <= '1';
                else
                    clkout <= '0';
                end if;
            when "011" =>
                if temp(22) = '1' then
                    clkout <= '1';
                else
                    clkout <= '0';
                end if;
            when "100" =>
                if temp(21) = '1' then
                    clkout <= '1';
                else
                    clkout <= '0';
                end if;
            when "101" =>
                if temp(19) = '1' then
                    clkout <= '1';
                else
```

```
                        clkout <= '0';
                    end if;
                when "110" =>
                    if temp(18) = '1' then
                        clkout <= '1';
                    else
                        clkout <= '0';
                    end if;
                when "111" =>
                    if temp(17) = '1' then
                        clkout <= '1';
                    else
                        clkout <= '0';
                    end if;
            end case;
        end process;
end behavioral;
```

C Appendix C: Test MMU Files

C.1 m68k_fakemmu_1.vhd

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

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_fakemmu_1 is
end m68k_fakemmu_1;

architecture mixed of m68k_fakemmu_1 is
   component m68k_cpu_core is
      port (
         reset: in std_logic; -- active high
         clock: in std_logic;

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

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

         -- address bus
         addrbus: out std_logic_vector(31 downto 0);
         -- data bus
         databus: inout std_logic_vector(31 downto 0);

         -- for debugging
         flags: out std_logic_vector(4 downto 0)

      );
   end component;

   signal reset: std_logic;
   signal clock: std_logic;
   signal busrw: std_logic;
   signal busreq: std_logic;
   signal busdone: std_logic;
   signal busdatasize: std_logic_vector(1 downto 0);
   signal addrbus: std_logic_vector(31 downto 0);
   signal databus: std_logic_vector(31 downto 0);

begin
   cpucore: m68k_cpu_core
      port map (
         reset => reset, clock => clock, -- panic => null,
         -- Note: we ignore panic as in simulation the CPU core will
         -- raise an assertion itself on a panic.
         busrw => busrw, busreq => busreq, busdone => busdone,
         busdatasize => busdatasize, addrbus => addrbus, databus => databus
      );

   busread: process (busreq)
   begin
      if rising_edge(busreq)
      then
```

```
        assert (busrw = '0')
           report "Bus requested a read."
           severity WARNING;
        assert (busdatasize = "10")
           report "Bus data size request not 16 bits."
           severity WARNING;

          -- memory reads
          case addrbus is
             when x"00000100" => -- move.w #02, d1
                databus <= "00000000000000000010000001111100";
                                          --   ^ ^ ^     ^ immediate
                                          --   | | +-- data reg d1
                                          --   | +-- word size
                                          --   +-- move
             when x"00000102" =>
                 databus <= x"00020002"; -- the #$20002

             when x"00000104" => -- move.w #03, d2
                databus <= "00000000000000000010000010111100";
             when x"00000106" =>
                databus <= x"00030003"; -- the #$30003

            -- add together
             when x"00000108" => -- add d2 to d1
                 databus <= "00000000000000001101001001000010";
                                          --    ^   ^  ^^ ^source data reg 2
                                          --    |   |  |+- word length
                                          --    |   |  +- destination is data reg
                                          --    |   +- which data register
                                          --    +- add

             when x"0000010A" => -- add #5 to d0
                 databus <= "00000000000000001101000001111100";
                                          --    ^   ^  ^^ ^immediate
                                          --    |   |  |+- word length
                                          --    |   |  +- destination is data reg
                                          --    |   +- which data register
                                          --    +- add

             when x"0000010C" => -- the #$50005
                databus <= x"00050005"; -- the #$50005

             when x"0000010E" => -- bra $100
                 databus <= "00000000000000000110000011110000";
                                           --    ^       ^-- offset
                                           --    +-- branch
                 report "Branching to start..." severity NOTE;

             when others =>
                 report "Memory request from unhandled location"
                 severity WARNING;
          end case;
          busdone <= '1';

       else -- no busreq
          busdone <= '0';
          databus <= x"00000000";
       end if;--busreq
    end process busread;

    init_run: process
    begin
       reset <= '1';
       clock <= '0';
       wait for 1 ms;
```

```
      clock <= '1';
      wait for 1 ms;

      reset <= '0';
      wait for 1 ms;

      -- from this point, the other stuff does the work

      while (true)
      loop
         clock <= '0';
         wait for 1 ms;
         clock <= '1';
         wait for 1 ms;
      end loop;

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

C.2 m68k_fakemmu_2.vhd

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

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_fakemmu_2 is
end m68k_fakemmu_2;

architecture mixed of m68k_fakemmu_2 is
  component m68k_cpu_core is
    port (
      reset: in std_logic; -- active high
      clock: in std_logic;

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

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

      -- address bus
      addrbus: out std_logic_vector(31 downto 0);
      -- data bus
      databus: inout std_logic_vector(31 downto 0)
    );
  end component;

  signal reset: std_logic;
  signal clock: std_logic;
  signal busrw: std_logic;
  signal busreq: std_logic;
  signal busdone: std_logic;
  signal busdatasize: std_logic_vector(1 downto 0);
  signal addrbus: std_logic_vector(31 downto 0);
  signal databus: std_logic_vector(31 downto 0);

begin
  cpucore: m68k_cpu_core
    port map (
      reset => reset, clock => clock, -- panic => null,
      -- Note: we ignore panic as in simulation the CPU core will
      -- raise an assertion itself on a panic.
      busrw => busrw, busreq => busreq, busdone => busdone,
      busdatasize => busdatasize, addrbus => addrbus, databus => databus
    );

  busread: process (busreq)
  begin
    if rising_edge(busreq)
    then
      assert (busrw = '0')
        report "Bus requested a read."
        severity WARNING;
      assert (busdatasize = "10")
        report "Bus data size request not 16 bits."
        severity WARNING;
```

```
            -- fake memory reads
            case addrbus is
               when x"00000100" => -- move.w #0, d1
                  databus <= "00000000000000000010000001111100";
                                          -- ^ ^ ^      ^ immediate
                                          -- | | +-- data reg d1
                                          -- | +-- word size
                                          -- +-- move
               when x"00000102" =>
                   databus <= x"00000000"; -- the #0

               when x"00000104" => -- move.w #$BEEF, d2
                  databus <= "00000000000000000010000010111100";
               when x"00000106" =>
                  databus <= x"0000BEEF"; -- the immediate data

              -- more to come here when more opcodes implemented
              -- add and save in d0

               when x"00000108" => -- bra $100
                   databus <= "00000000000000000110000000000000";
                                           --   ^       ^-- offset all 0
                                           --   +-- branch
               when x"0000010A" => -- bra $100
                   databus <= x"0000FFF4"; -- -12 decimal in word
                   report "Branching to start..." severity NOTE;

               when others =>
                   report "Memory request from unhandled location"
                   severity WARNING;
            end case;
            busdone <= '1';

      else -- no busreq
            busdone <= '0';
            databus <= x"00000000";
      end if;--busreq
   end process busread;

   init_run: process
   begin
      reset <= '1';
      clock <= '0';
      wait for 1 ms;

      clock <= '1';
      wait for 1 ms;

      reset <= '0';
      wait for 1 ms;

      -- from this point, the other stuff does the work

      while (true)
      loop
         clock <= '0';
         wait for 1 ms;
         clock <= '1';
         wait for 1 ms;
      end loop;

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

C.3 m68k_fakemmu_3.vhd

```vhdl
-- fake/test "mmu" for m68k
-- returns various values set here depending on data requested.
-- test of move, and and branch
-- Dylan Leigh s3017239

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_fakemmu_3 is
end m68k_fakemmu_3;

architecture mixed of m68k_fakemmu_3 is
   component m68k_cpu_core is
      port (
         reset: in std_logic; -- active high
         clock: in std_logic;

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

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

         -- address bus
         addrbus: out std_logic_vector(31 downto 0);
         -- data bus
         databus: inout std_logic_vector(31 downto 0)
      );
   end component;

   signal reset: std_logic;
   signal clock: std_logic;
   signal busrw: std_logic;
   signal busreq: std_logic;
   signal busdone: std_logic;
   signal busdatasize: std_logic_vector(1 downto 0);
   signal addrbus: std_logic_vector(31 downto 0);
   signal databus: std_logic_vector(31 downto 0);

begin
   cpucore: m68k_cpu_core
      port map (
         reset => reset, clock => clock, -- panic => null,
         -- Note: we ignore panic as in simulation the CPU core will
         -- raise an assertion itself on a panic.
         busrw => busrw, busreq => busreq, busdone => busdone,
         busdatasize => busdatasize, addrbus => addrbus, databus => databus
      );

   busread: process (busreq)
   begin
      if rising_edge(busreq)
      then
         assert (busrw = '0')
            report "Bus requested a read."
            severity WARNING;
         assert (busdatasize = "10")
            report "Bus data size request not 16 bits."
            severity WARNING;
```

```

           -- memory reads
           case addrbus is
              when x"00000100" => -- move.w 10 repeating, d1
                 databus <= "00000000000000000010000001111100";
                                         --  ^ ^ ^     ^ immediate
                                         --  | | +-- data reg d1
                                         --  | +-- word size
                                         --  +-- move
              when x"00000102" =>
                 databus <= "10101010101010101010101010101010";

              when x"00000104" => -- move.w 01 repeating, d2
                 databus <= "00000000000000000010000010111100";
              when x"00000106" =>
                 databus <= "01010101010101010101010101010101";

             -- and together
              when x"00000108" => -- and d2 to d1
                  databus <= "00000000000000001100001001000010";
                                         --   ^   ^  ^^ ^source data reg 2
                                         --   |   |  |+- word length
                                         --   |   |  +- destination is data reg
                                         --   |   +- which data register
                                         --   +- and

              when x"0000010A" => -- and 1all to d0
                  databus <= "00000000000000001100000001111100";
                                         --   ^   ^  ^^ ^immediate
                                         --   |   |  |+- word length
                                         --   |   |  +- destination is data reg
                                         --   |   +- which data register
                                         --   +- and

              when x"0000010C" =>
                 databus <= x"0000FFFF"; -- the #$FFFF

              when x"0000010E" => -- bra $100
                  databus <= "00000000000000000110000011110000";
                                          --   ^       ^-- offset
                                          --   +-- branch
                  report "Branching to start..." severity NOTE;

              when others =>
                  report "Memory request from unhandled location"
                  severity WARNING;
           end case;
           busdone <= '1';

      else -- no busreq
           busdone <= '0';
           databus <= x"00000000";
      end if;--busreq
   end process busread;

   init_run: process
   begin
      reset <= '1';
      clock <= '0';
      wait for 1 ms;

      clock <= '1';
      wait for 1 ms;

      reset <= '0';
      wait for 1 ms;

      -- from this point, the other stuff does the work

      while (true)
      loop
         clock <= '0';
         wait for 1 ms;
         clock <= '1';
         wait for 1 ms;
      end loop;

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

C.4 m68k_fakemmu_4.vhd

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

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity m68k_fakemmu_4 is
end m68k_fakemmu_4;

architecture mixed of m68k_fakemmu_4 is
  component m68k_cpu_core is
    port (
      reset: in std_logic; -- active high
      clock: in std_logic;

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

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

      -- address bus
      addrbus: out std_logic_vector(31 downto 0);
      -- data bus
      databus: inout std_logic_vector(31 downto 0)
    );
  end component;

  signal reset: std_logic;
  signal clock: std_logic;
  signal busrw: std_logic;
  signal busreq: std_logic;
  signal busdone: std_logic;
  signal busdatasize: std_logic_vector(1 downto 0);
  signal addrbus: std_logic_vector(31 downto 0);
  signal databus: std_logic_vector(31 downto 0);

begin
  cpucore: m68k_cpu_core
    port map (
      reset => reset, clock => clock, -- panic => null,
      -- Note: we ignore panic as in simulation the CPU core will
      -- raise an assertion itself on a panic.
      busrw => busrw, busreq => busreq, busdone => busdone,
      busdatasize => busdatasize, addrbus => addrbus, databus => databus
    );

  busread: process (busreq)
  begin
    if rising_edge(busreq)
    then
      assert (busrw = '0')
        report "Bus requested a read."
        severity WARNING;
      assert (busdatasize = "10")
        report "Bus data size request not 16 bits."
        severity WARNING;
```

```
          case addrbus is
             when x"00000100" => -- move.w $1a0, d1
                databus <= "00000000000000000010000001111000";
                                        --   ^ ^ ^     ^ absolute source
                                        --   | | +-- to data reg d1
                                        --   | +-- word size
                                        --   +-- move
             when x"00000102" =>
                 databus <= x"000001A0"; -- the address 1a0

             when x"00000104" => -- move.w $1bb, d2
                databus <= "00000000000000000010000010111000";
                                        --   ^ ^ ^     ^ absolute source
                                        --   | | +-- to data reg d2
                                        --   | +-- word size
                                        --   +-- move
             when x"00000106" =>
                databus <= x"000001bb"; -- the address

            -- more to come here when more opcodes implemented
            -- add and save in d0

             when x"00000108" => -- bra $100
                 databus <= "00000000000000000110000000000000";
                                         --   ^       ^-- offset all 0
                                         --   +-- branch

             when x"0000010A" => -- bra $100 offset
                 databus <= x"0000FFF4"; -- -12 decimal in word
                 report "Branching to start..." severity NOTE;

-- addresses for data
             when x"000001A0" =>
                 databus <= x"00001111"; -- data at 1A0

             when x"000001BB" =>
                 databus <= x"00002222"; -- data at 1BB

             when others =>
                 report "Memory request from unhandled location"
                 severity WARNING;
          end case;
          busdone <= '1';

     else -- no busreq
          busdone <= '0';
          databus <= x"00000000";
     end if;--busreq
  end process busread;

  init_run: process
  begin
     reset <= '1';
     clock <= '0';
     wait for 1 ms;

     clock <= '1';
     wait for 1 ms;

     reset <= '0';
     wait for 1 ms;

     -- from this point, the other stuff does the work

     while (true)
```

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

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

D Appendix D: Makefile

```
# Makefile for building m68k with GHDL
# Dylan Leigh s3017239

#
# Macros/Variables
#

# does not include testbenches
VHDLSRCS= m68k_cpu_core.vhd
#VHDLSRCS= gen_register.vhd regfile_8.vhd attic also has alu
VHDLOBJS= ${VHDLSRCS:.vhd=.o}

#testbench sources
#TESTSRCS= test_gen_register.vhd attic
TESTSRCS= m68k_fakemmu_1.vhd m68k_fakemmu_2.vhd m68k_fakemmu_3.vhd m68k_fakemmu_4.vhd
TESTOBJS= ${TESTSRCS:.vhd=.o}
TESTEXES= ${TESTSRCS:.vhd=}

#
# High level targets
#

.PHONY: default all alltests runtests clean

# builds all components and all testbenches and runs testbenches
default: runtests

# Note: following the common usage for makefiles, "all" does not mean
# "everything" but to build all of the final product
# (i.e. no testbenches or extra stuff).
all: ${VHDLOBJS}

# analyse tests
alltests: all ${TESTOBJS}

# elaborate and execute tests
runtests: alltests
        ghdl --elab-run m68k_fakemmu_1 --stop-time=300ms --wave=fakemmu_1.ghw
        ghdl --elab-run m68k_fakemmu_2 --stop-time=200ms --wave=fakemmu_2.ghw
        ghdl --elab-run m68k_fakemmu_3 --stop-time=300ms --wave=fakemmu_3.ghw
        ghdl --elab-run m68k_fakemmu_4 --stop-time=300ms --wave=fakemmu_4.ghw

#
# Implicit targets
#

# clear out all suffixes
.SUFFIXES:
# # list only those we use
.SUFFIXES: .o .vhd .

.vhd.o: $<
        ghdl -a $<

.o.: $<
        ghdl -e $@

#
# Misc targets
#

clean:
#       ghdl --remove
        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 tco 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.