A Motorola MC68008 Op-code compatible VHDL Microprocessor

David Lynch
SN : 00019640
Supervisor : Michael Manzke

B.A (Mod) Computer Science Final Year Project

April, 2004

Abstract

The goal of this project is to implement a Motorola 68008 op-code compatible microprocessor capable of functioning on an FPGA. The CPU is to be specified fully in VHDL and is designed to emulate the functionality of the MC68008 in terms of instruction set decoding, operand addressing and bus operation. The ultimate target device is a custom designed FPGA-Board used in education of Computer Science and Engineering Students in TCD. As a result of this a large emphasis has been placed on structured, correct design flow and specification as well as readability of the code.

Acknowledgments

I would like to thank Michael Manzke, my project supervisor for all the help and support given throughout the project. Appreciation is also extended to Ross Brennan for support on his FPGA board and bus-interfacing. Lastly I would like to thank John McCarthy, my Integrated Systems Design lecturer for help in mastering the use of the Xilinx platform and companion tools.[1].

[1]This document has been typeset using LaTeX

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an introduction to the Motorola 68008 VHDL[1] CPU project. The background and motivations behind such a project are outlined. In doing so, current project-board design is introduced along with the improved board design as proposed and implemented by Ross Brennan, together with arguments for the continued use of the MC68008 in an educational context. This chapter also explores the design approach taken and reasons for such an approach.

1.1 Current Project

This project branches directly from the final year project completed by Ross Brennan entitled, *The Design and Evaluation of an FPGA based Microprocessor Project Board* [2]. The core objective of Brennan‘s project was to explore the possibility and feasibility of introducing configurable hardware devices into Architecture courses taken by Computer Science students and Computer Engineering Students with a view to upgrading the project system-boards to take advantage of recent hardware advances.

1.1.1 Computer Architecture Coursework

As a coursework section in the *Computer Architecture I* module of the B.A. (Mod) Computer Science degree at TCD. Students are required to build a functioning microprocessor system using core components. By completing such a project students become acquainted with the fundamentals of CPU-to-peripheral interaction, bus operation, asynchronous communication and board-level programming. The project is centred on the Motorola MC68008 processor with which students are required to interface two 8k static RAM chips, an 8k EEPROM and a Rockwell R6551 Asynchronous communications adapter. Wire wrap is used to connect the devices together after they have been placed on a purpose built system board.

The MC68008 processor is a 16-bit CISC[2] design released in late 1982 as a replica of the MC68000 except with a reduced data-bus width of 8-bits. This reduction was implemented in order to preserve compatibility with legacy 8-bit devices.

[1]Very High Speed Integrated Circuit Hardware Description Language

[2]Complex Instruction Set Computer

As an early CISC computer the 68000 strikes a nice balance between a powerful instruction set, advanced memory addressing capability and simplicity of interface. Largely due to the absence of legacy instruction support, floating point arithmetic, or multi-media extensions commonly found in todays processors, this balance is very desirable in an educational context, particularly when introducing junior freshmen students to low-level programming.

The MC68008 possesses a 20-bit address bus, allowing 1MB of in total addressable memory space. An 8-bit bi-directional data bus allows the MC68008 to communicate external on-board devices using asynchronous bus protocols defined by Motorola. The package is described in Fig. 1.1 and the pin functions detailed in Table 1.1.

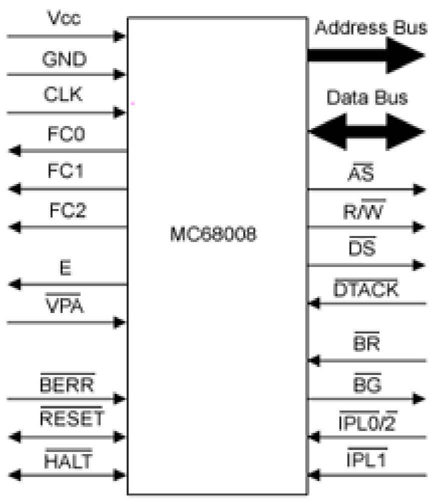


Figure 1.1: The 68008 package

Pins	Description	Function
A19 : A0	Address Bus	
D7 : D0	Data Bus	
/DTACK	Bus Read Acknowledge	Asynchronous bus control
/AS	Address-on-bus	
/DS	Data Strobe	
R/W	Read Write	
/VPA	Valid Peripheral Address	Peripheral Control
E	Peripheral Clock Synchronisation	
/IPL[2:0]	Interrupt Level	External Interrupt Control
/BERR	Bus Error	System Control
/RESET	Reset Signal	
/HALT	Processor Halt	
/BR	Bus Request	Bus Arbitration Control
/BG	Bus Grant	

Table 1.1: 68k Pin Description

Components Used

- 1 x Motorola MC68008 Microprocessor
- 1 x R6551 Serial Interface
- 1 x 2764 8k EPROM
- 2 x HM6116P 2Kx8-bit Static RAM
- miscellaneous components including A 7404 Hex Inverter, 7407 Hex Buffer, 74LS73 J-K Flip Flops, Dual 4-bit binary counter and a resistor pack.

Students are given access to a pre-designed project board. This board has a 5V power supply and a parallel interface for connection with a logic state analyser. A 14.7465Mhz Crystal Oscillator is also integrated to provide clock signals for the ACIA[3] and the MC68008. The board acts as a basis on which to place the 5V TTL pin packages supplied. Once placed in the pin-holders students use copper wire wrap to connect the pin packages together.

The project stages

- Verify the board is generating a 15MHz clock signal using an oscilloscope.
- Use the 4-bit counter and the resistor pack to generate an 8Mhz clock signal for the MC68008 and a 1Mhz clock signal for the ACIAs.
- Use the hex-inverter package, and Schmitt triggers to create a debounce circuit which ensures the RESET signal is kept asserted for at least 3 clock cycles. This ensures correct reset of internal circuitry and the peripherals.
- Memory map the RAM chips, the EEPROM and the ACIA using the GAL[4] to generate appropriate chip-enable signals depending on the processor Address Strobe (/AS), Data Strobe (/DS) and Address Bus $(A19 : A0)$. The students use PALASM to generate equations which are then programmed to the GAL chip.
- Program the EEPROM to provide the processor with a means of testing the bus interface. A suggested approach is through repeated JMP instruction execution.
- Establish and verify the functionality of the RAM. This can be done with a similar loop to the above, with the additional use of the MOVE instruction to test the memory space where each of the RAM chips resides.
- Use the JK Flip flop package provided to generate correct VPA and IPL signalling to enable the ACIA to operate as a peripheral component. Use these signals to provide an EEPROM program to handle interrupts generated by the ACIA and create a transparent link with a PC using a hyper-terminal session.

[3]Asynchronous Communications Device

[4]General Array Logic — A simple programmable device

- Upon successful completion of the above, the students are then asked to create a simple monitor program using the EEPROM and 68k Assembly language.

1.1.2 The FPGA Design

FPGA devices are re-programmable and offer unparalleled flexibility for hardware specification. With recent advances in re-programmable logic devices, such as decreased FPGA device scale and increased speed as well as the evolution of synthesis software tools like Xilinx Foundation[13] it has become increasingly more feasible to develop fully-specified CPUs within a programmable hardware paradigm. The open source movement has seen a spiral in such core implementations.[4]

The advantages of using an FPGA implementation are numerous. The student is no longer only exposed to bus interfacing, but allowed to examine how such a device may be logically implemented using a hardware description language. Furthermore, the flexibility of the FPGA design allows students to compare and contrast specific technologies. For example, the student may wish to switch off the on-chip cache of a microprocessor to measure performance, or, switch between a RISC and CISC CPU to use different instruction sets, all without having to re-wire the microprocessor board set up. Lastly, an FPGA design could replace the 68008 pin-packages which are obsolete in their current form and are becoming increasingly difficult to replace.

Board Hardware

Ross Brennan prototyped a replacement project board that was centred on a Xilinx Virtex-II XV1000-256 FPGA. As 1 Million Gate Device, this FPGA would provide ample basis in which to situate a RISC HDL-CPU [5] as well as space for additional student design or a second CPU. The RAM and GAL components used in the original project are still used here, although adapted to LL3V voltage operation to suit the Xilinx FPGA voltages[6]. A PROM has also been added to the design. This feature allows on-board programming of the FPGA on power up. As the FPGA is based on SRAM technology the programming is lost at power down. The PROM provides a quick way to program the FPGA with a selected architecture without having to connect the system to a synthesis tool. [2, R. Brennan]

When the HDL design is synthesized, a bit file is generated by the synthesis tool. This is then downloaded onto the FPGA using an interface port. However, because the FPGA is based on SRAM, the programmed design is lost at power-down. The PROM remedies this by providing the FPGA with a bit file at power on. The PROM may also contain different designs, e.g. the bit-files for a RISC design and a MC68008 design allowing the user to change between the two without having to synthesize and download a new design from scratch.

[5]The proposed RISC design would use 80% of these resources at full-specification

[6]The original packages are 5V TTL

LEON CPU Core

The LEON[5] core is a fully featured SPARC V8[12] compatible Core specified in VHDL and developed by the European Space Agency. It has a 32-bit data bus, floating point units, on-chip caches and a memory control unit. This core was chosen because it is an OpenSource VHDL implementation that is highly configurable, thus lending itself to the ideal educational environment described above.

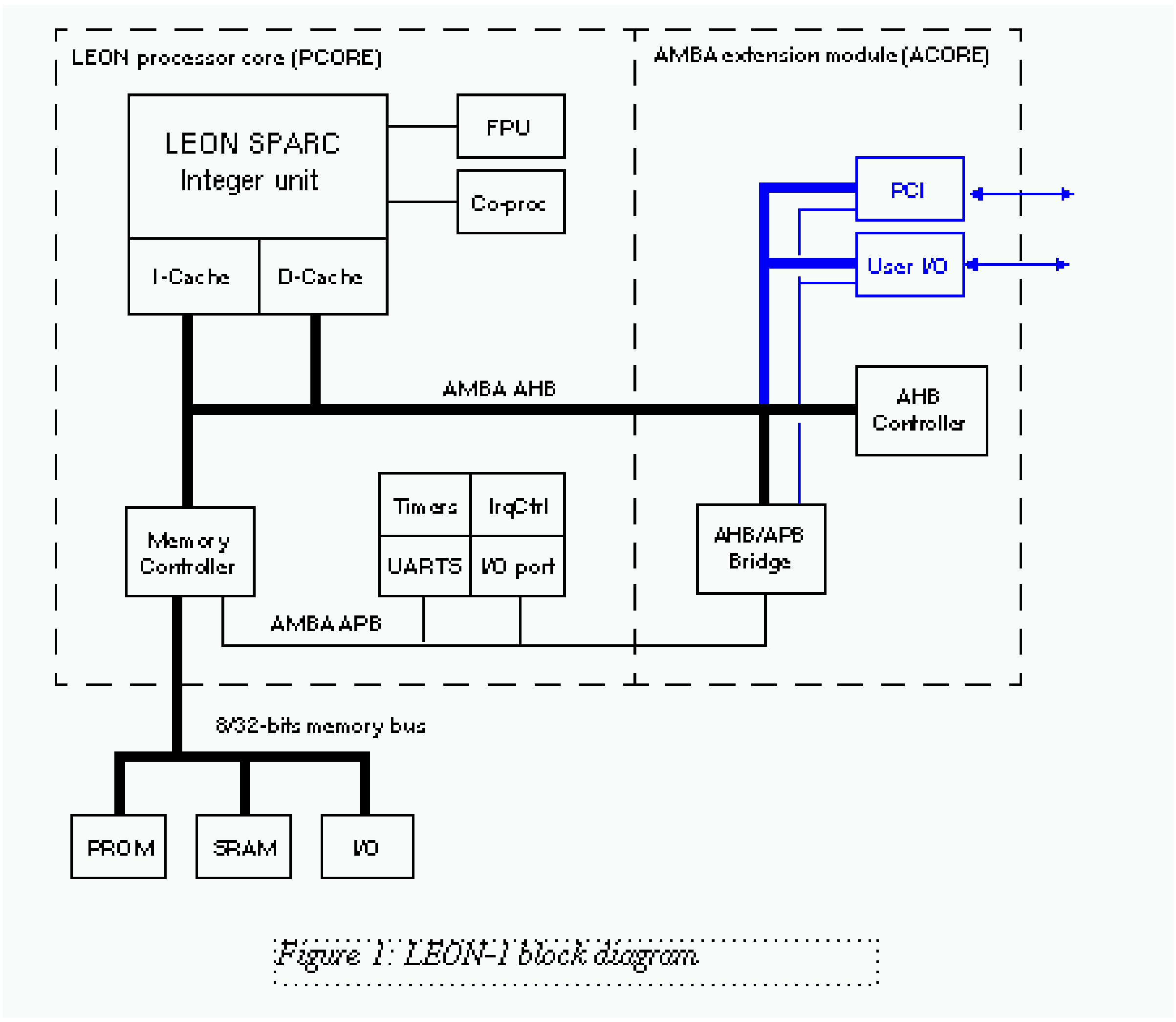


Figure 1.2: The Leon Core. Source : ESA[5]

In order for the LEON to suffice as a replacement for the MC68008 pin-package modifications were necessary. The main objective was to emulate the simplicity of MC68008 bus interaction and to ensure compatibility with the currently used RAM, ROM and GAL devices. The LEON was cut down to those components only necessary for the function of the CPU. Although the LEON is capable of a 58Mhz clock speed [2, Brennan], an 8Mhz clock has been provided. Caches were also removed from the design as caching would inhibit the ability of the student to use an oscilloscope and logic state analyser to monitor bus activity. If the executed program, such as the JMP loop, was read from memory once, it would be cached in one bus cycle. Subsequent bus cycles would not occur, thus causing difficulty when testing for correct bus function. Likewise, The Memory Management unit was removed. The internal UARTS also needed to be removed as interfacing an asynchronous communications device to the processor is a core requirement of the Computer Architecture project therefore use of internal serial communication is not required. For the same reason, internal reset circuitry was removed. The LEON data-bus was reduced to 8-bits, to preserve component compatibility. It is calculated that an approx 95% decrease in performance would result from the changes, however it is noted here that the result is not a disadvantage for the student. A functional system is much more desirable than a fast system in and educational context. [2]

1.2 Motivation

There are several purposes to this project. The main purpose is to provide a VHDL 68008 microprocessor for the above project board. As described by Brennan and Manzke [3], a 68008 HDL CPU would provide students with a CPU on-which they could learn assembly programming using a small, simple but powerful instruction set. More importantly once the processor had a bus interface capable of being used on the existing project board, it opens possibilities for expanding FPGA based microprocessor boards to allow users to configure various parts of the design to their particular needs and use different architectures within the same physical hardware paradigm. The 68008 CPU could also help preserve the use of the Motorola processor as a teaching tool. It is currently heavily used in Computer Science and Computer Engineering hardware modules due to its simplicity when compared to other processors. However, as the processor ages, it has become more and more difficult to obtain replacement 5VTLL packages when necessary. An FPGA design may eventually be a cheaper and simpler way of maintaining 68000 resources within the college.

Also, a motivation is to provide students of Computer Architecture with an understandable implementation of a micro-programmed CISC architecture as implemented using VHDL. Currently, Junior Freshmen studying for the B.A. (Mod) are exposed to digital fundamentals such as adder design and multiplexer design using primitive digital logic gates. Senior Freshmen are exposed to multiple-cycle CPU design and more complex primitives such as Arithmetic Logic Units (ALU). It would be very desirable for students of these courses to be able to break down a CPU implementation into its core components and examine the design. A VHDL design programmed in an efficient and modular manner allows a user to examine a core component, completely ignoring the surrounding implementation. Therefore students can not only see where a primitive component may be used, but also how such a primitive may be implemented using a hardware specification language.

1.3 Design Approach

There were several tasks to be completed before implementation work began. The first was to master the VHDL programming language. This was done by completing a selection of second-year assignments involving simple CPU design. Several VHDL exercises using primitive components were also completed. This coupled learning VHDL with re-familiarisation of digital logic concepts.

Sources used for learning VHDL programming were *http://www.vhdl-online.de*[6], Xilinx[13]and *VHDL, From Simulation to Synthesis* [14]. These resources also provided tutorial on how the Xilinx FPGA design and simulation platform is used.

The next task was to research CPU design implementation. A large part of this research work involved the study of Computer Architecture and Digital Design. *Digital Design*[8] and *Logic and Computer Architecture Fundamentals* both by M. Morris Mano[9] provided coverage of these two topics. Once a comfortable level of VHDL proficiency had been obtained it was necessary to turn attentions to the 68008 microprocessor.

The 68000 Programmers manual[10] and the 68000 users manual[11] were invaluable resources when it came to detailing 68000 internals and operation, especially when it came to instruction decoding and bus cycles. The Motorola instruction set was examined and broken down into smaller, more visibly structured chunks from which decisions about instruction decoding and hardware support were made. Next, information about the 68000 software model was gathered. *The 68000 Microprocessor: Hardware and Software Principles* by *James L. Aontonakos*[1] was found to be a good resource that provided a useful abstraction from the meticulous detail and formality of the Motorola manuals.

Once this initial research had been completed, the hardware design process could start. The design is implemented entirely in VHDL using Xilinx v5.1 tools for programming and synthesis together with ModelSim 5.6 for simulation. The CISC CPU as described by Mano and Kime [9] in *Logic and Computer Design Fundamentals* has been extensively adapted to create a VHDL CPU capable of executing Motorola 68000 op-code. The following modifications were necessary.

- Implement the CISC CPU in VHDL.
- Add support for variable sized operands i.e. 8/16/32-bit and correct condition codes.
- Add support for 68008 Arithmetic.
- Devise and implement enhanced branching capabilities.
- 16-bit 68000 instruction decoding.
- Exception generation and handling.
- 68000 memory addressing mode support.
- Add an FPGA board compatible Bus Interface.

The CPU has been built in an incremental fashion starting with the register file. Each individual module has been programmed and tested under a behavioural simulation to verify functionality and post-synthesis simulation to verify the result could operate on an FPGA. The design approach taken is in-line with the VHDL design flow specified in *VHDL From Simulation To Synthesis* [14]. Firstly, the hardware was specified and programmed. Secondly, a behavioural simulation was performed to verify functionality of the design. Next, a post-synthesis simulation was carried out to verify the design still functions correctly in the target environment. This design flow was applied to every component of the project from full-adder up to CPU package.

1.3.1 VHDL Programming

The design is implemented wholly in VHDL, mainly to maintain compatibility with the existing LEON design on the FPGA board. Also, second year Computer Science students are exposed to VHDL programming and it makes sense to maintain this trend in hardware education within the course. It is also argued here that VHDL is a more reader-friendly language than the alternative, Verilog.

VHDL is a hardware specification language that infers a hardware model by reducing a hardware design specified in a software model to primitive components. This automatic hardware compilation is a very powerful way, although not always resource-optimal way, of creating a functional hardware designs from a specification without having to wire primitive gates manually.

RTL Style

A key requirement for this project is synthesizability[7] of design. Not only would a synthesizable design be downloadable to the target project board but would prove that a somewhat efficient and modular design is in place. In hardware design specification for FPGA it is a necessity to keep the hardware that may be generated as a result of coding in mind. In doing this the synthesis tool is helped greatly with inferring an efficient, functional design free of undesirable side effects such as unwanted latches and multi-sources.

Although VHDL supports many language constructs and methodologies only a subset of it is synthesizable. The only VHDL language constructs (aside from declaration constructs) used in the design are *process*, *case*,*if,then,else*,and *concurrent signal assignment*. This project adheres to these constraints not only to create an efficient, synthesizable design, but a readable one capable of being modified and enhanced in the future. With this in mind a Register Transfer Level of abstraction has been kept consistent throughout the design. As a result both synchronous and combinatorial logic has been defined through use of concurrently running processes. The Xilinx synthesis tool will deduce synchronous logic for processes that rely on a global clock. Otherwise, the synthesis tool will try to infer combinatorial logic. The code listing below shows an example of using a process to infer a multiplexer.

[7]A synthesizable design is capable of being implemented correctly on an FPGA or other target device

Care has been taken to avoid the inference of unwanted latches at every stage of the project thus ensuring the Model is always in a known state during operation. In fact, the only use of latches is in the bus interface when latching lower order bytes of a higher order operation. Wherever data needs to be stored, registers have been instantiated and wired manually using port-maps.

```
-- Source : Control.vhd
-- Infer a multiplexer to manage the MicroCCR load enable
Manage_T_CCR : process(TC)
begin
        case TC is
        when '0' =>
                T_CR_Load_Enable <= '0';
        when '1' =>
                T_CR_Load_Enable <= '1';
        when others =>
                T_CR_Load_Enable <= 'X';
        end case;
end process;
```

1.4 Motorola 68000 Microrocessor Overview

The 68000 is internally a 32-bit CPU. Eight 32 bit data registers, numbered D0-D7, are provided for data manipulation. In addition, 32-bit address registers A0-A6 are designed to hold memory addresses in order to aid data movement within the system and increase the complexity of memory addressing modes. Register A7 is a special case of such a register. It is known as the system stack pointer (SSP) when handling system control instructions (i.e. in supervisor mode), and user stack pointer (USP) in user programming mode (i.e. executing user code). A 32 bit program counter, of which the lower 24-bits are valid (20 in the 68008,due to its decreased address bus size), provides memory addressing for the currently executing instruction. A 16-bit system register holds information necessary for system functions such as branching condition codes and interrupt information. Figure 1.3 shows a graphical outline of the system register contents. The lower 8-bits, or user byte, hold condition codes such as overflow which are generated by the functional unit. The higher order byte, or the system byte, holds the current interrupt level priority mask used in peripheral control, the trace bit (whether or not to call a trace exception after each instruction) and the supervisor bit which is set when performing system control tasks.

Twelve addressing modes are available, detailed in table 1.2. Direct mode addressing transfers the data to or from any of the data or address registers. Indirect addressing is performed using an address register which contains the address in memory that that data to be manipulated resides. Post increment, pre-decrement and displacement variations of indirect addressing are implemented as modes useful during loop instructions and stack manipulation. Immediate addressing involves specifying an operand directly i.e. by encoding it in an extension to a core instruction word. Absolute addressing is similar with the exception that the immediate data specifies a location in a specific part of memory.

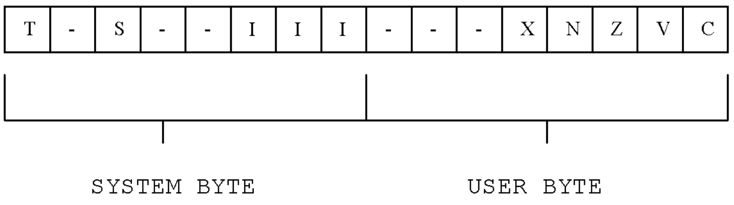


Figure 1.3: The 68000 System Register

Mode	**Use**
Data Register Direct	Data register holds data
Address Register Direct	Address register holds data
Address Register Indirect	Using data pointed to by an Ar
Address Register Direct with Post-Increment	Increment the Ar after fetch/write
Address Register Indirect with Pre-Decrement	Decrement the Ar before fetch/write
Address Register Direct with Displacement	Data at X+/-(Ar)
Address Register Direct with Index	Data at X+/-(Ar)+/-(Dn or An)
Absolute Short	A 16-bit address specifies dest/srce
Absolute Long	A 32-bit address specified dest/srce
PC Indirect with displacement	Data at PC +/- (Dn or An)
PC Indirect with index	Data at PC +/- (An) +/- (Dn or An)
Immediate	Data encoded in instruction

Table 1.2: Addressing Modes [1]

The Motorola 68000 family instruction set is very much a CISC entity. The 68000 boasts powerful and numerous addressing modes and multiple-cycle instructions such as multiply and divide. Such features are not found in comparable RISC processors, such as the SPARC-II, which is a load/store architecture with a minimal instruction set. The eighty-one (counting branch as one and omitting no-op) instructions available to the 68000 programmer can be grouped loosely into four categories. Data Transfer Group, Arithmetic, Logical and Bit-manipulation group, program control group and system control group.[1] The data transfer group includes instructions to move data between memory and the processor and also between internal registers. There are also instructions, such as LINK and PEA that aid memory manipulation and the use of stack data structures. A complete list of this grouping is shown in table 1.3.

As is typical of a CISC instruction set, arithmetic instructions are plentiful. The 68000 is capable of manipulating 8, 16 or 32 bit signed operands. Instructions include ADD, SUB, MULU (Multiply Unsigned), DIVS (Divide Signed),TAS (Test and Set) and CMP(Compare). The 68000 is also capable of performing simple arithmetic on binary coded decimal. Shift operations are also available to the programmer. ASd (Arithmetic Shift) and LSd (Logical Shift) as well as ROd (Rotate) are the main shift instructions.

Program control accomplished using the instructions illustrated in table 1.4. These instructions aid the implementation of subroutines and conditional branching.

Instruction	Description
EXG	Exchange Registers
LEA	Load Effective Address
LINK	Link
MOVE	Move Data
MOVEA	Move Address
MOVEM	Move Multiple
MOVEP	Move Peripheral
MOVEQ	Move Quick
PEA	Push Effective Address
SWAP	Swap Register Halves
UNLK	Unlink Stack Frame

Table 1.3: Data Transfer Instructions [1]

Instruction	Description
Bcc	Branch on Condition defined by cc
DBcc	Decrement and Branch
Scc	Set Condition Codes defined by cc
BRA	Branch Always
BSR	Branch to Subroutine
JMP	Jump to Address
RTR	Return and restore
RTS	Return from Subroutine

Table 1.4: Program Flow Control Instructions

The last group deals with system control and features instructions that reset the processor, call external subroutines, handle exceptions and manipulate stack pointers. Examples include RESET, AND to CCR, ILLEGAL and TRAP.

An instruction can have no, one or two operands which can be encoded in the instruction or specified using one of the addressing modes. *Instruction.size op1,op2* is the general syntax for two operand instructions. Instruction is any two operand instruction defined by the set, size is one of B, W, L for Byte, Word and Long operation and Op1 and Op2 are effective address that specify the location of the operands and the location to place the result, which is in most cases op2. One and zero operand instructions take the same form, e.g *RESET* or *NEG (A0)*. A more detailed discussion of the 68000 instruction set is considered as the design of the

VHDL CPU instruction decoder is explained in chapter 2. An exact dissection of the instruction set can be found in the Motorola 68000 Programmers Manual [10].

Chapter 2

Hardware Design

Although implemented in VHDL this project is essentially hardware based. This chapter explores in detail the 68008 is terms of the hardware implemented to create the functioning model. Firstly, the functionality of the design is explained followed by how such a design is implemented on the VHDL platform. Initially the design of the data-path including functional unit and register file is considered. Next, Bus Interfacing as well as sequencing and control hardware are examined. Where code listings occur, the source file is quoted and can be found on the accompanying CD in either the 68000-VHDL folder (Xilinx 5.1i) or the X_6800-VHDL folder (Xilinx 4.1). Appendix B contains a listing of these files.

2.1 Overview

The hardware design follows loosely the outline of a CISC CPU as detailed in *Logic and Computer Design Fundamentals* [9, Mano, Kime Pgs: 511 to 542]. However, extensive modifications were necessary to make an MC68008 op-code compatible CPU and to ensure a Bus Interface was implemented that could support the CPU on an FPGA board. The data-path is modified to perform arithmetic on 8-bit, 16-bit and 32-bit signed or unsigned integers. It was also necessary to add support for the correct generation of condition codes as specified by Motorola [11]. The control path is modified to cater for 16-bit Motorola instruction decoding, the addressing modes outline earlier and CPU interrupt and exception handling. Control ROM capacity is increased and the control word expanded to 64 bits. More complex sequencing has also been added, to cater for branching within micro-programs and the insertion of wait stages when interfacing the CPU with peripherals. All internal data transfer occurs through a 32 bit bus. The lower order byte of this bus is valid for a byte (8-bit) operation and the lower order word is valid for word (16-bit) operation. The remaining bits are either 0 or to be ignored. If a register is being written using a byte or word operation the lower 8 or 16 bits are the only ones affected by the operation. An overview of the hardware put in place is found in the schematic on the following page.

2.2 Functional Unit

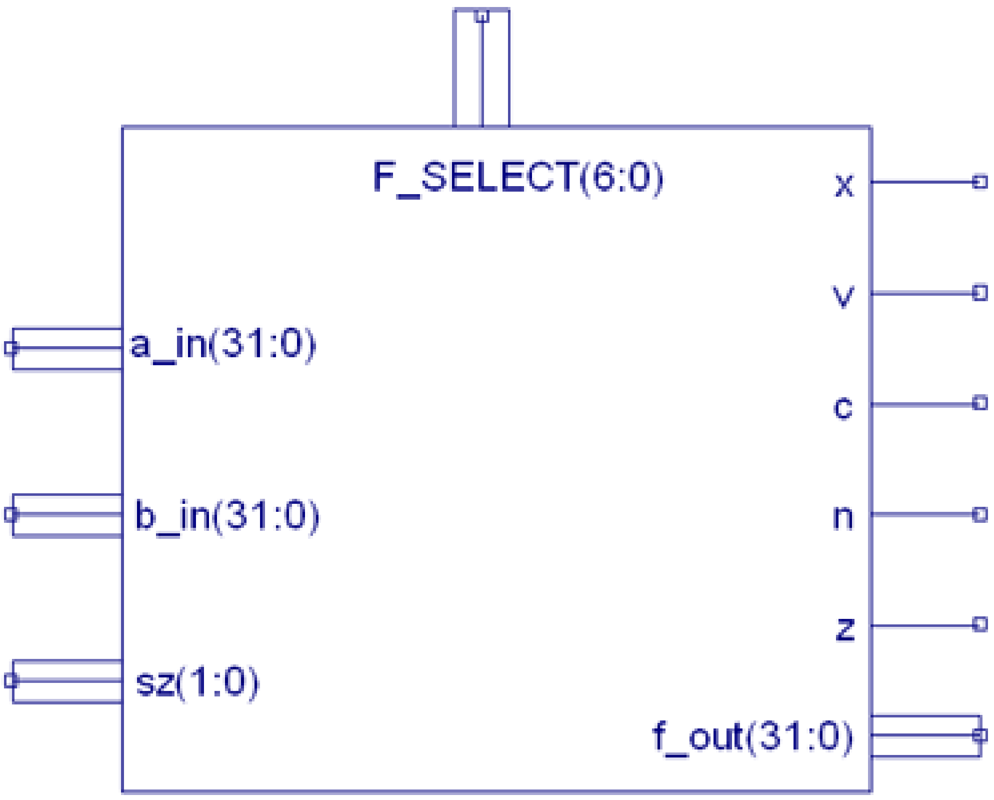


Figure 2.1: Functional Unit Block Diagram

It is necessary to be able to perform the essential arithmetic and shift operations in this module. Condition code generation must also be considered. The Functional Unit consists of two main components, an ALU and a shifter. Two inputs, A and B feed the ALU operands while The B input feeds the shifter. A size control input signal is also needed to distinguish between operation sizes. This is particularly important for the rotate instructions of the shifter and for correct condition code generation. In keeping with the Motorola instruction set 00 represents a byte operation, 01 a word operation and 10 a long-word operation. The ALU and shifter are fed simultaneously and each generates a result and appropriate condition codes. It is then the function of MUX F and MUX FC to choose the correct condition codes to send to out of the functional unit and to the condition code registers. An operation control inputs selects the desired ALU function. This is detailed in table 2.1. F Select is a 7 bit word that controls the entire operation of the unit. Bit 6 is fed to MUX F and MUX FC to distinguish between ALU and shift operations. Bits 5 and 4 respectively carry the extend bit (X) and the carry bit (C) from the system register to the shifter. Bits 3 down to 0 choose ALU operation, and bits 2 down to 0 select a shifter operation. A 32-bit result is always generated, however the correctness of the generated result is dependant on the size of the operation. This is also true for the condition codes generated. The operations supported provide functionality identical to the Motorola instructions add, subtract, logic and shift operations and a basis on which to perform more complex operations such as multiply, divide and add binary coded decimal through the use of micro-programming.

F Select	Function
00000	Transfer A
00001	Increment A
00010	A + B
00011	A + B + 1
00100	Negate B
00101	Decrement A
00110	A B (A + (/B + 1))
00111	Transfer A
0100x	A and B
0101x	A or B
0110x	A xor B
0111x	Negate A
1x000	Arithmetic Shift Right
1x001	Arithmetic Shift Left
1x010	Rotate Right
1x011	Rotate Left
1x100	Rotate Right with extend (ROXR)
1x101	Rotate Left with extend (ROXL)
1x110	Logical Shift Right
1x111	Logical Shift Left

Table 2.1: Functional Unit Operation Select

This block is also responsible for generation of the negative (N) and zero (Z) condition codes. The Z condition is generated by examining the result and setting a flag if the lower order 8, 16 or 32 bits are all zero. In twos complement arithmetic, the most significant bit determines the sign. If it is 1 then the integer can be interpreted as negative.[8]

2.2.1 The Arithmetic and Logic Unit

The ALU is 32-bits wide and controlled by a 4 bit select input (GSELECT) and a 2 bit size input (SZ). Bit 3 of the control signal chooses between logical and arithmetic operation within the unit. The arithmetic has been implemented using an array of full adders. Following *Digital Design* [8, Mano pgs 119-120] a HDL description has encapsulated the functionality of a 1 bit adder, correctly generating a sum and carry out given a carry in and inputs X and Y and Z. This full adder was then instantiated 32 times using the *generate* VHDL synthesis function. This function allows multiple instantiations of the same entity using a loop counter. Other than the first carry in, which is zero, each carry out is fed into the carry in of the more significant adder.

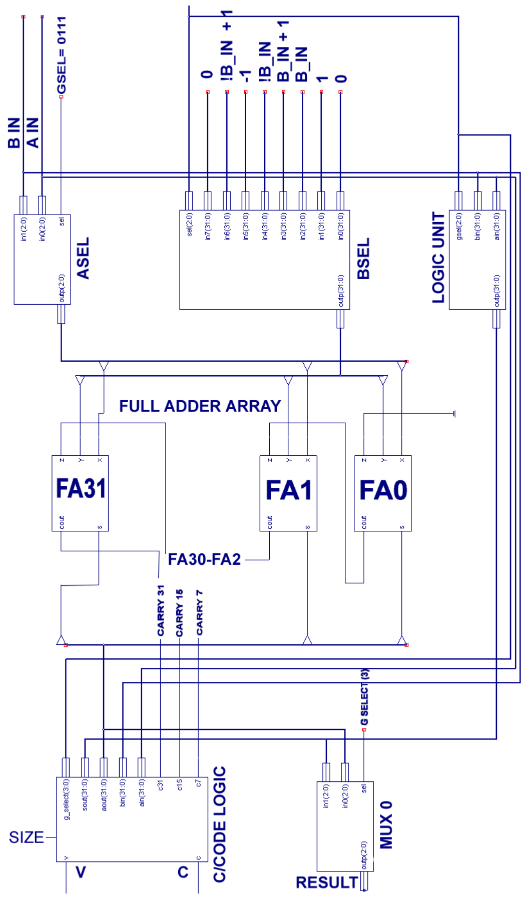


Figure 2.2: ALU Block Diagram

Performing operations

Addition is performed by feeding each individual bit of A and B in to the X and Y inputs of the full adders. The carry will propagate from LSB to MSB and be set in the most significant carry out. Subtraction is performed by the twos complement method. The B operand in first negated, and then incremented to obtain its twos complement form and the fed into the Y inputs. The A input is fed as normal into the X inputs When decrementing A, a twos complement minus 1 is sent into the Y inputs of the full adders. Similarly, for single operand operations such as transfer a zero is applied to the Y inputs.

The logic is implemented by feeding each input into the gates in question. These gates are automatically generated by the synthesis tool given correct HDL

```
-- Source : ALU32.vhd
BIN <=LongZero  when G_SELECT="0000"  else        -- Transfer A
          LongOne when G_SELECT="0001"  else        -- Increment A
          B_IN      when G_SELECT="0010"  else        -- Add B to A
          B_IN+1    when G_SELECT="0011"  else        -- A + B + 1
          Not B_IN when G_SELECT="0100"  else         -- !B
          Minus1    when G_SELECT="0110"  else        -- Decrement A
          (not B_IN)+1 when G_SELECT="0101"  else     -- 2s Compliment of B
          LongZero;
```

Figure 2.3: BIN control CSA. BIN Feeds the Y input of the adder array

specification. For example, `AOUT <= AIN xor BIN` is the VHDL statement that will perform exclusive-or on the inputs AIN and BIN

Condition Codes

The MC68008 uses 5 condition codes to perform branching functions. Overflow, or V, is set when a sign change occurs on an operand. In twos complement arithmetic, which relies on a modulo numbering system, this flag can be generated by an XOR on the MSB entered to functional unit input with the MSB of the desired operation result. A 1 will indicate the sign has changed. The Carry or C flag indicates a carry out of an addition or a borrow out of a subtraction. The C flag is set when the most significant carry of an ALU operation is 1 except for subtraction. The borrow in subtraction is only set if the input operand A is greater than input operand B, otherwise carries generated by twos complement subtraction are ignored and C is 0. Extend or X is set to the value of C for arithmetic functions only and is simply fed out as a replica of the C signal. It remains unassigned for logical operations. The position of the most significant bit of the result in the ALU is dependant on the size input. When generating Carry and Overflow condition codes for Word and Byte operations it is necessary to consider the 16th and the 8-th input and carry bits respectively. Any output more significant than these should be discarded as they are not guaranteed to be correct.

Implementation and Example

To implement the above two multiplexers decide what enters the X input and what enters the Y input of the adder array. These MUXs are controlled by G Select. 1 for increment, -1 for decrement and (not B) + 1 for subtract are three of the choices for the Y input. When only one input is required a Zero is sent through B to the adder array. A bit-vector of 32 bits is used to store the carries, which then can be operated on using concurrent assignments. A MUX decides which result to send to the ALU output.

In an example operation we feed 2 into input A and 3 into input B. Subtraction is desired so GSELECT is set to 0110. Twos complement of B is fed from the MUX into the adder array while A is fed directly. The size of the operation is set to byte

(00). Since 2-3 is -1 the output is 11111111 or -1 in byte twos complement form. Carry is set because a borrow has occurred, similarly a sign change has occurred on A and therefore the V flag is set.

2.2.2 The Shifter

The Motorola supports arithmetic shift through ASL and ASR, logical shift through LSL and LSR, bitwise rotation through ROL and ROR and rotation with the extend bit (X) acting as an expansion to the operand in question through ROXL and ROXR. Arithmetic shifting influences the C and X condition codes. When shifting left the MSB sets carry and extend. A zero is shifted in from the right. Shifting right, the LSB sets the carry and the MSB is shifted in from the left. When rotating the carry is set to be the bit that has been rotated from the MSB to the LSB or LSB to the MSB, the extend bit is not effected unless rotating with extend. In this instance X is set from C but acts as the LSB or MSB of the operation. Logical shift is exactly the same as arithmetic shift except 0 is filled in from the left when shifting right. This functionality is encapsulated in the following diagrams.

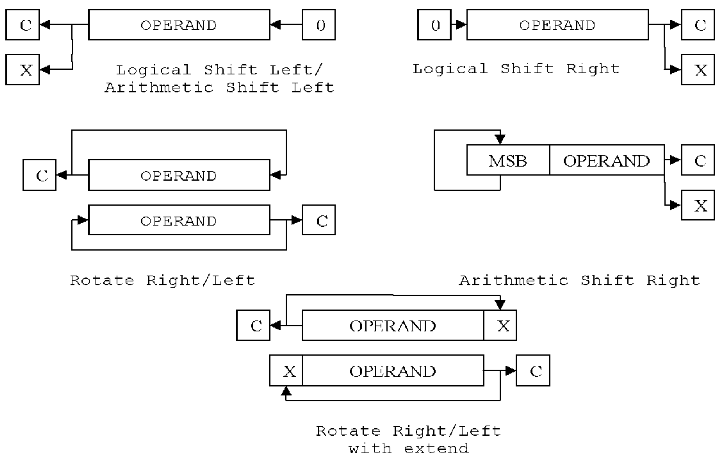


Figure 2.4: 68000 Shifting. Source : MC68000 Programmers Reference

Implementation and example

The VHDL CPU shifter takes carry in and extend in from the system register. Depending on the size of the operation, combinatorial logic will shift the input left or right and produce a result. A requirement of shifting is that writing to the system register is enabled. This is to ensure condition codes are set correctly and correct operation of the ROXL instruction. The shifting operations have been implemented in VHDL using combinatorial logic. An example listing is found in figure 2.4. This code piece ensures proper rotation under each operand size. The shifter code causes the Xilinx synthesis tool to infer a set of multiplexers that generates a barrel-shifter capable of shifting and left or right once per clock cycle.

```
-- Source : Shifter.vhd
-- Rotate Right setting carryout to be the rotated bit
case SZ is
when "00" =>
        S_OUT(6 downto 0)        <= S_IN(7 downto 1);
        S_OUT(7)                 <= S_IN(0);
        C_OUT                    <= S_IN(0);
        X_OUT                    <= '0';
        V_OUT                    <= '0';
        S_OUT(31 downto 8)       <= "000000000000000000000000";
when "01" =>
        S_OUT(14 downto 0)       <= S_IN(15 downto 1);
        S_OUT(15)                <= S_IN(0);
        C_OUT                    <= S_IN(0);
        X_OUT                    <= '0';
        V_OUT                    <= '0';
        S_out(31 downto 16)      <= "0000000000000000";
when "10" | "11" =>
        S_OUT(30 downto 0)       <= S_IN(31 downto 1);
        S_OUT(31)                <= S_IN(0);
        C_OUT                    <= S_IN(0);
        V_OUT                    <= '0';
        X_OUT                    <= '0';
when others =>
        S_OUT(31 downto 0)       <= 32_X;
        V_OUT                    <= 'X';
        C_OUT                    <= 'X';
        X_out                    <= 'X';
 end case;
```

Figure 2.5: VHDL Rotate Right using *case* for combinatorial logic

In an example using the shift-and-add method of multiplication, we are required to shift left and shift right at different stages. Given two byte operands `00001000` and `00000011` we are to obtain the result 00011000. Firstly, `00000011` is shifted right one, setting C. This is accomplished by F Select `10110`. The carry flag is set and we can branch to the addition part of the algorithm given that code. Next we shift `00001000` left one to get the next partial addition. This is accomplished using F Select `10111`.

2.2.3 Control Registers

The microprocessor uses two registers to control the flow of micro-program and user-program execution. In order to maintain compatibility with existing 68000 code the system register has been designed to match it. As with the 68000 the user byte contains the condition codes, X,N,Z,V,C. The system byte contains support for seven levels of interrupts in IPL2,1 and 0. This holds the current interrupt level and is used in processing external interrupts after every instruction. The S bit, or supervisor bit, is designed to inform the system whether or not it is executing in user or supervisor mode. In the 68000 programming model certain instructions, such as RESET, may only be executed in supervisor mode. This bit is set when executing system level operations such as handling exceptions. The trace bit (T) is set when the CPU is operating in trace mode. A trace exception loads in the trace vector handler which may contain code to push the system registers to a stack. This is useful in debugging software as the programmer can see exactly what is contained in all the system registers after every instruction execution.

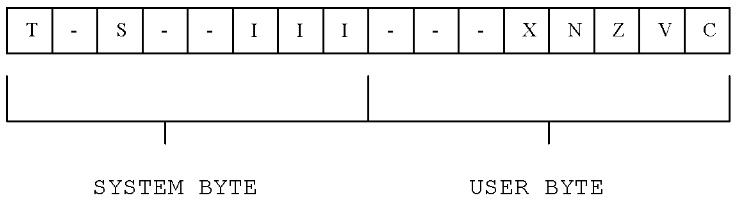


Figure 2.6: The System Register

The second control register is the Micro-CCR. This register is 4 bits wide and is identical to the SR for bits 0 to 3. The function of the Micro-CCR is to provide a separate means of tracking conditions generated by micro-instructions. In Microcode that performs multiple cycle operations it may be necessary to consider condition codes generated without affecting the condition codes in the system register. This distinction is illustrated by taking the example above. In a micro-coded shift and add multiplication we can branch to an addition micro-subroutine if the bit in question is a 1. This could be done by shifting and using the carry as explained earlier. However, since unsigned multiplication in the 68000 will never produce a carry or overflow, it is necessary to ensure that the system register carry is not set. Thus, the Micro-CCR provides the codes necessary.

Implementation

The system register is implemented as a 16-bit positive edge triggered register. Load control of the SR is implemented using combinatorial logic in VHDL and s controlled by the CWORD field MC. Motorola instructions will set condition codes in one of 5 ways. Either the SR remains unchanged, C,V,Z,N are changed, C,V,Z,N,X are changed, Z is changed only, N is changed only or the SR is loaded from the internal data bus. Also, the trace and supervisor bits must be modifiable and the interrupt priority mask must be changeable. The Micro-CCR has simpler loading logic. TC is the CWORD field that dictates whether or not the CCR is loaded with the codes that are provided by the functional unit.

MC	Function
0000	Not Loading SR
0001	Changing C,V,Z,N
0010	Changing C,V,Z,N,X
0011	Changing Z only
0100	Changing N only
0101	Set IPL Mask
0110	Set Trace Bit
0111	Clear Trace Bit
1000	Set supervisor Bit
1001	Clear Supervisor Bit
1111	Load SR

Table 2.2: SR Load Control

2.3 Data Storage and Movement

The processor overview diagram shows how data is moved within the processor. There is an internal 32-bit data-bus that circulates data to the register file, through the ALU and to the control-path. MUX D is responsible for choosing between external data and data from the ALU and placing it on the internal data-bus where it feeds the rest of the processor. It is controlled by the CWORD field MD and when set to 1 reads external data from the Bus Interface.

2.3.1 Register File

The register file contains a bank of 32 registers. Each register is 32-bits wide and is has a load control dependant on the CWORD SZ field. When the size is byte the lower order 8 bits are loaded and the rest of the register remains unchanged. This is also the case for word writes, except the lower order 16-bits are considered. The register file contains the registers D0-D7, A0-A6 the System Stack Pointer and the User Stack Pointer (A7). It also contains an internal bank of registers only addressable by micro programs. A Memory Data register and Memory Address register are provided to allow the creation of a memory read and memory write micro-routines. A micro-program can simply place the data that should be written to memory and the address to where it is to be written into these registers and branch to a micro-routine. The RRW field of the CWORD controls whether a register is written or not.

In the Motorola 68000 each instruction will contain at least one effective address (EA) which will act as a source or destination for operand fetching and write back. In the same vein as the memory registers EA Data and EA Address are provided to allow the exploitation of common operand addressing modes within instructions. The vector address register stores the base address of a to-be-executed

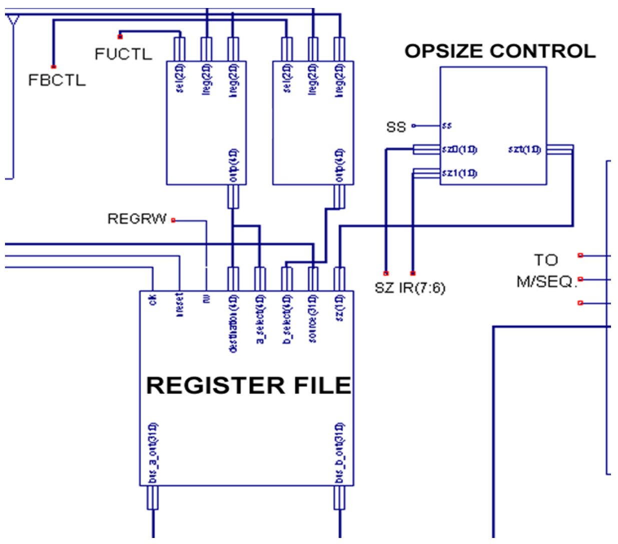


Figure 2.7: Register File and Control

exception handler. The remainder of the register file consists of temporary registers T0 to T8 and a Zero provider. The temporary registers are to be used in the storing of intermediate results of micro-operations and micro-programs. The Zero register simply provides a constant zero for clearing registers.

Register	Address
D0-D7	00xxx
A0-A7	01xxx
System Stack Pointer	10000
EA Address	10001
EA Data	10010
Memory Address	10011
Memory Data	10100
T0 T8	10100 11100
Vector Address Register - 11110 Zero Provider	11111

Table 2.3: Internal Register Map

Register Addressing and Selection

Register writing is controlled by an RW signal. A 5 bit destination control is provided to choose a register to write. There are two output ports from the register file to cater for the input of 2 operands into the ALU and to cater for simultaneous selection of an address register and a data register for placement onto the data and address busses.

Since destination and source selection is encoded in a 68000 instruction word the register file must be capable of being controlled by the CWORD and by the instruction currently being executed.

The multiplexer FUCTL,FBCTL and SZCTL dictate which entity is controlling the register file and how. When SZCTL is set to 0 then the CWORD field SZ controls the size of an operation. When SZCTL is set to 1 the SZ field is controlled by the 68000 instruction word bits 7 and 8, the bits that specify the different operation sizes in the 68000 instruction set.

FUCTL feeds the register destination and port A output selection while FBCTL controls the B output selection. These multiplexers are instrumental in generating the correct register file addresses for destination and selection. When FUCTL and FUBCTL select lines are 000 the CWORD field ASEL,BSEL and DEST control register file addressing and all 32 registers are accessible. In a typical 68000 two operand instruction the instruction bits 3:0 and 11:9 control destination and source registers. When FUBCTL and FUCTL are not 000 these are the bits that control destination and source selection.

The maximum number of registers addressable with 3 bits is 7. It is therefore necessary to modify these bits in order to enable access to the Address Registers, located from register 8 to 15. When FUBCTL and FUCTL selections are 001 and 010 a 1 is concatenated onto the instruction bits 3:0 and 11:9 to allow the instruction to access a total of 16 registers. It is noted that it is still not possible for the instruction to access registers reserved for internal operation, this can only be done by the CWORD. The CWORD must be aware of the addressing mode in order to append the 1 to any instruction address and access the address registers. Table 2.3 gives a list of data registers while Table 2.4 defines FUBCTL and FUCTL operation.

Binary	**FUCTL**	**FBCTL**
000	BSEL from CWORD	DEST/ASEL from CWORD
001	IR[11:9] to B Out	IR[11:9] to A Out and Data Reg. Destination
010	IR[3:0] to B Out	IR[3:0] to A out and Data Reg. Destination
011	IR[11:9] to B Out	IR[11:9] to A out and Addr. Reg. Destination
100	IR[3:0] to B Out	IR[3:0] to A out and Addr. Reg. Destination

Table 2.4: Destination and Source Selection in the Register File

2.3.2 Data Bus and Address Bus Selection

It is necessary in any CPU to provide a means of selecting what is put on the data bus and the address bus. This function has been tied into what operands are selected for entry into the Functional Unit. MUX A controls the address bus and input A of the ALU while MUX B controls the data bus and the B input of the ALU.

MUX A and B assignments

Controlled by the CWORD field MA, this multiplexer has 4 inputs. Firstly, it provides a means of transferring what is selected on the A port, and the B port of the into the ALU and the address bus. This is done through input 0, A out and 3, B out. This allows any register to be used for addressing, any register to be passed to the ALU or through the ALU to the IR or PC and back into the register file. The second input is the PC. The program counter needs to be placed on the address bus during instruction fetching and immediate operand fetching. The last input to MUX A is a vector address input. It is necessary to load the PC with a vector address when executing and exception handler routine. MUX A provides a means by which this address can be transferred from the vector decoder into the register file.

Mux B is fed the B output of the register file. The next input is the displacement register. This register is used in conjunction with the PC input of MUX A when executing instructions that adjust the PC using a 32-bit displacement. The current IPL mask is also an input to MUX B as it is necessary to compare it with the requested IPL when processing interrupts. Lastly, the entire system register has been specified as an input. This is to allow for any arithmetic manipulation of the SR that may be deemed necessary and allows the SR to be directly put on the data bus when saving the SR during interrupt processing.

MUXes A and B are implemented in code using processes to infer combinatorial logic.

2.3.3 The IPL control Module

This module is a small entity that stores a pending interrupt request. If multiple interrupts are requested between interrupt handling then this module will latch the highest value request. In practical terms this is done using a comparator connected to the IPL pins and a 3 bit register. This hardware is inferred using the VHDL constructs detailed below. The signal MUXI is connected to an insanitation of a 3-bit positive edge triggered register.

```
-- Source : Control.vhd
IPLControl : process(P_IPL_w,SIG_R_out)
begin
        if P_IPL_w < SIG_R_out then
                MUX_I    <= P_IPL_w;
        else
                MUX_I    <= SIG_R_out;
        end if;
end process;
```

2.4 The Bus Interface

It is necessary for any CPU to communicate with memory and peripheral devices. As outlined in the introduction, any CPU that is to be put on the FPGA board must use a bus interface compatible with the existing bus implementation. The bus interface protocols as defined by Motorola for 68008 8-bit operation are compatible with the ROM, RAM and ASIC and therefore are sufficient for this purpose. This design does not strictly follow the timing of the Motorola 68008 bus cycle as defined in the *Programmers Manual*[10], as it contains certain cycles that are not applicable in this design context (such as empty clock cycles). The atomic Read-Modify-Write cycle has been omitted here as it is only deemed useful in multiprocessor situations when using the Test and Set instruction in creating locks. There is scope, however, to add this in at a later time. Similarly bus arbitration cycles have been omitted. However, the strobe generation, insertion of wait states and data bus operation are identical.

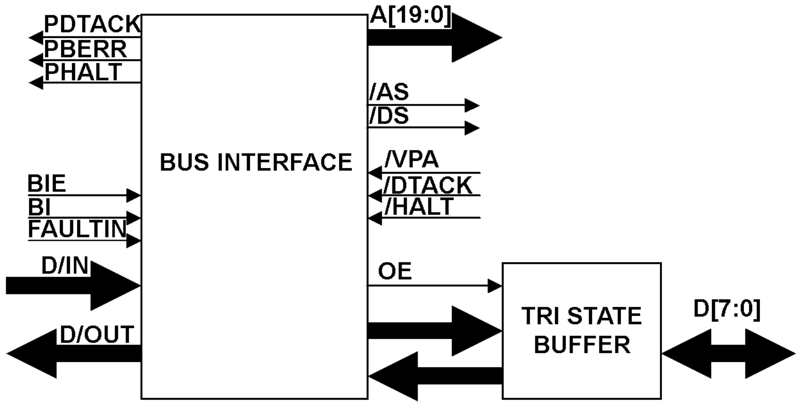


Figure 2.8: Bus Interface Block Diagram

2.4.1 Implementation

To provide abstraction from the CROM, the bus interface has been designed as a separate entity capable of performing independent bus cycles. Control signals /DS (Data Strobe), /AS (Address Strobe) are generated as necessary. It receives address data and operand data from MUXes A and B. The bus interface is connected directly to the address bus, and through a tri-state buffer to the data bus. A separate tri-state buffer module is used for the data-bus as its abstraction makes it easier to handle bi-directional communication. The address bus is simply placed in Hi-Z when it is not being used and would not benefit from such an abstraction. When data is read from the data bus it is passed to MUX D and enters the data and control paths. Due to the fact that data bus is only 8-bits wide the bus interface needs extended functionality to handle reads and writes of words and long words. This essentially entails the BI being capable of multiple read and write cycles in a single operating mode.

The bus cycles generate signals in a finite sequence and it was decided that a VHDL finite state machine would be the simplest and most readable way of designing such an interface. The bus interface is implemented in VHDL using a two-process state machine. The first process handles clocking. On the positive edge of each clock cycle it updates a CurrentState register with a NextState register. The second process generates the combinatorial logic to handle state changes. The bus interface advances states depending on assigned inputs and outputs.

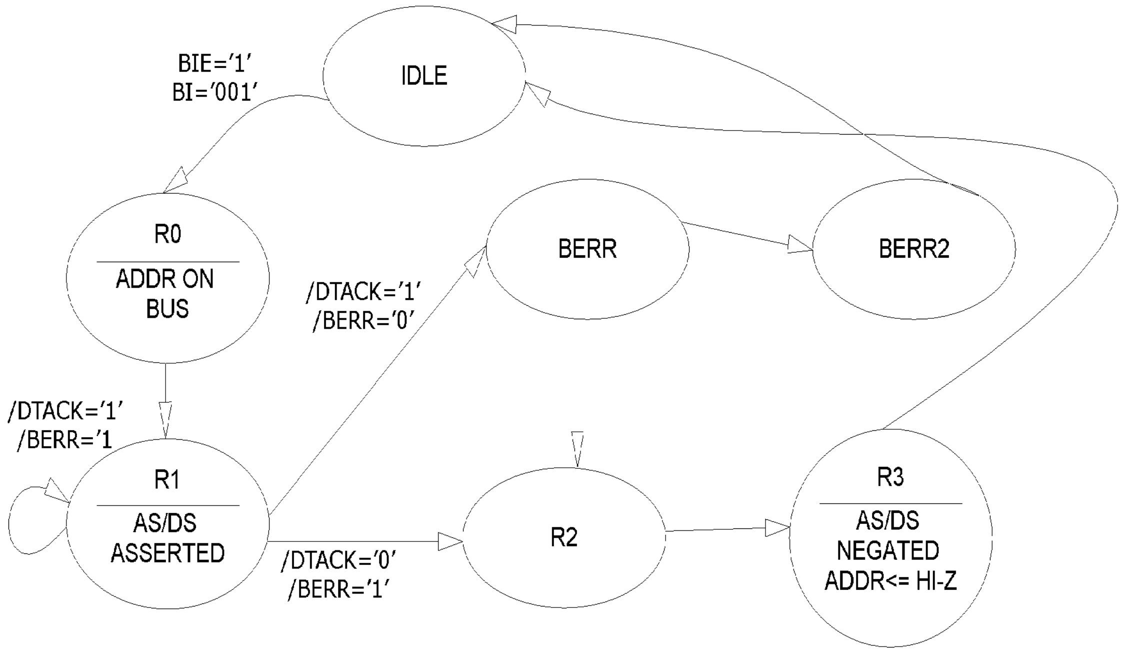


Figure 2.9: Sample Read Operation State flow (Simplified)

When BIE is high the module is either performing a write cycle, a read cycle or an interrupt acknowledge cycle depending on the input field BI. While these operations are being performed the CAR address remains constant and the micro-sequencer only advances the CAR when the bus interface generates a finished signal (PDTACK). The reason for this implementation is to allow for the insertion of wait stages in bus cycles. Since it can not be certain how many wait states will be inserted, simple micro-program NOPs will not suffice and the progression of the micro-program must be blocked until the bus cycle has completed. Any form of read or write cycle must use the third form of the CWORD, as outline later in this document.

2.4.2 Read and Write Cycles

In order to perform a read cycle BIE is set to enable. It is necessary to choose an operation using the BI input and provide the bus interface with a pointer to the location of the base byte i.e. an address. According to the operation size the bus interface will start at state RO and generate the bus signals as indicated in figure 2.11. The BI stays in the state R1 until DTACK is pulled low by the peripheral device, indicating data is on the bus. The lower byte is then latched in R2 by ensuring the tri-state buffer output enable is pulled low in order to read the data driven by the peripheral device. If the operation is a read-byte operation then PDTACK is pulled low, causing the CAR to be incremented and the data to read to the CPU registers. In order for PDTACK to effect the CAR in such a way it is necessary for the wait-for-DTACK condition to be set in the CCOND field and the address of the next micro-instruction to be specified using the 3rd form of the

```
-- Source : BusInterface.vhd
if ENABLE='0' then-- Wait Till We're Asked.

ADDR_OUT        <= "ZZZZZZZZZZZZZZZZZZZZ"; -- Not using A_Bus
DATA_BUS_OUT    <= "00000000"; -- Or data bus
D_OE            <= '0'; -- disable tristate output
LatchInt        <= '0'; -- not latching interrupt level
P_nBERR         <= '1'; -- no bus error exception
P_nHALT_w       <= '1'; -- no halt
p_nDTACK        <= '1'; -- no dtack to the microsequencer
L_LByte         <= "00000000"; -- not latching any bytes
L_HByte         <= "00000000";
H_LByte         <= "00000000";
H_HByte         <= "00000000";
OpSize          <= Byte; -- default opsize
RnW             <= '1'; -- not writing
nAS             <= '1'; -- no address on bus
nDS             <= '1'; -- no data from us
NextState       <= idle; -- wait for enable
```

Figure 2.10: Setting outputs for the idle state

CWORD. If the operation is word or long-word then this cycle is repeated 2 or 4 times each time incrementing the address pointer and latching the higher order bytes. PDTACK is not pulled low until the full operation is complete. This way a read-byte or read-longword operation does not have to enable the bus interface 2 or 4 times, it is done independently.

In performing a write cycle the CPU sends the address and data to be written into the bus interface and selects a write operation and again waits on PDTACK. The strobes are set in as illustrated in figure 2.12 and the write cycle begins. Like the read cycle the data is written in one-byte increments. When writing to the data bus, the tri-state buffer is set to output enable causing the data bus to be driven out of the high-impedance state.

2.4.3 Interrupt Acknowledge and Bus Error Cycles

68008 supports 3 levels of interrupts, however in keeping with the 68000 7 levels are supported by the project hardware. Interrupts are processed at the end of every instruction, and those generated during an instruction execution are labelled as pending by the IPL control module and processed after the execution. When handling interrupts the bus interface must perform an interrupt acknowledge cycle. When instructed to do so by the control word the Bus Interface module will start an interrupt acknowledge cycle as illustated with figure 2.13. /AS and /DS are pulled low, R/W high and the requested interrupt level is fed into the bus interface and on to the address bus lines A0 to A2. If the peripheral wishes to request its own vector number /DTACK is pulled low and the vector number is placed on the data bus. It

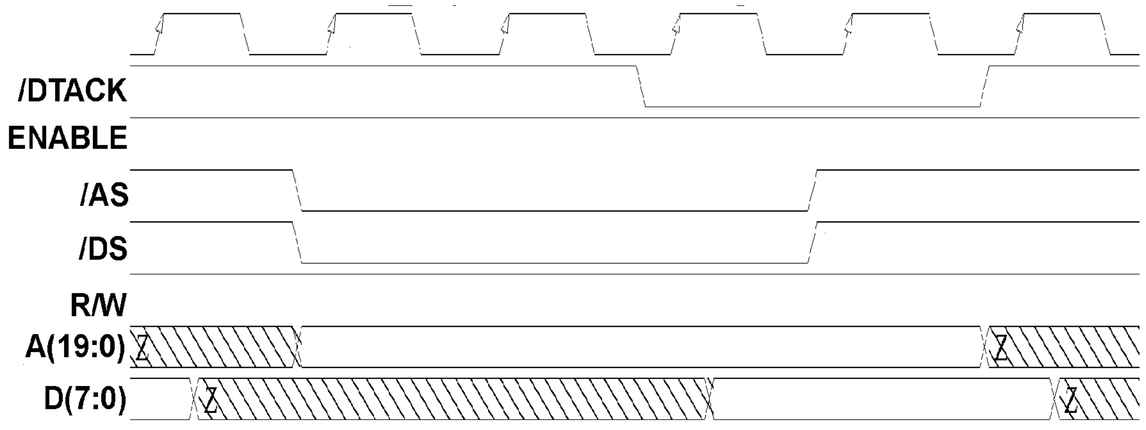


Figure 2.11: Bus Read Cycle

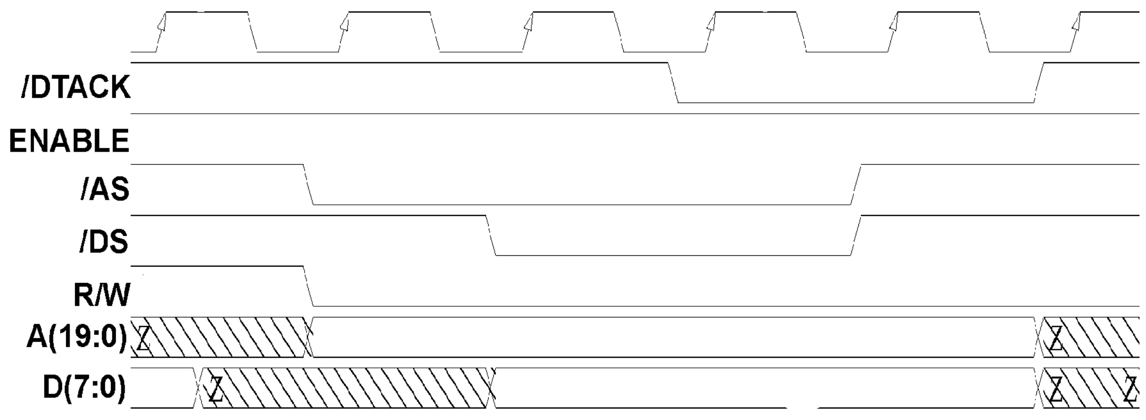


Figure 2.12: Bus Write Cycle

is now up to the micro-program to load and execute the appropriate vector.

When opting for auto-vectoring, i.e. when the interrupting device requests a pre-defined vector address, the peripheral pulls /VPA low instead of /DTACK. The bus interface will generate one of seven vector address and send it to the CPU to be loaded into the Vector Register. The vector handler can now be loaded and executed.

The last function performed by the bus interface is Bus Error control. If the bus error flag is pulled low by an external device during a bus cycle then PBERR is pulled low and a bus error exception is generated. During this error handling FAULTIN is pushed high. If a second bus error occurs during bus fault exception processing then a double bus fault has occurred and the processor enters the halt state only to be restarted by reset. This in done by pulling PHALT low.

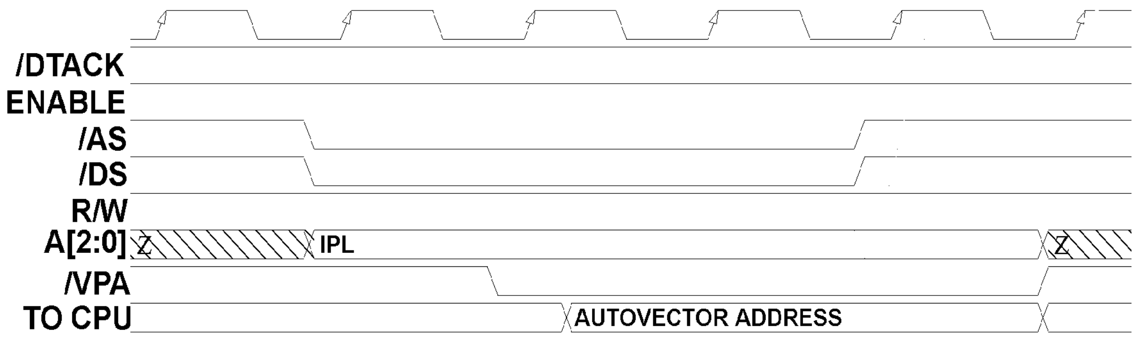


Figure 2.13: Auto-vector peripheral bus cycle

BI Input	**Function**
000	Read Byte
001	Read Word
010	Read Long-Word
011	Write Byte
100	Write Word
101	Write Long-word
110	Interrupt Acknowledge
111	Reserved for Read-Modify write

Table 2.5: BI Select

2.4.4 Bus Interface Hardware

All Bus Interface hardware is inferred automatically with the exception of an 8-bit register which is a temporary storage register designed to hold the interrupt vector. Latches are used to store intermediate read cycle values, such as the first three bytes of a long-word memory read.

2.5 Sequencing and Control

The most complex section of the CPU hardware design is the control path. The control path contains logic to generate control signals for the data path and is responsible for program sequencing and specifying how operands are manipulated in the design.

There are two common design approaches when considering control unit design. Hardwiring is physically implementing control using a state machine designed with often complex logic. Hardwired design involves considering the states necessary to provide functionality in the data-path and subsequently deducing a state machine capable of providing this functionality. This approach is the fastest and most hardware-economical approach to the design. However, this approach is also a very inflexible way of creating a design. Specifying hardware control for a CISC instruction set, which by definition has complexity built in at processor level, is a difficult task. Moreover, once the design has been implemented it is difficult to expand control without re-wiring the control design in order to add in new states.

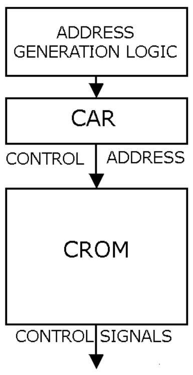


Figure 2.14: CROM and Control Address Register

Micro-programming is a less hardware-economical but more flexible approach. An on chip control ROM is used to provide control signals to the data-path. The control words are provided by an address kept in a Control Rom Address Register. This register is combined with program flow logic to provide Micro-program flow functionality. The CROM can be programmed to provide the functionality necessary an can be easily modified and expanded without modification of the surrounding architecture [9, Mano, Kime] This is the methodology that has been used in the CPU design. It is felt that a micro-programmed design would be easier to understand for anybody wishing to examine the design as it is based on a simple hardware specification. Also, and more importantly, a CROM design would be reprogrammable and expandable in future giving the design an increased adaptability. Since the FPGA is based on SRAM look-up tables for interconnection of primitive components there is no performance advantage in opting for a hardwired approach. A motivation of this project is to show how a CISC design may be implemented. The fact that Micro-coding is used in CISC designs such as the Pentium IV and the 68008 was another deciding factor in choosing micro-coding over hardwired control[7]

2.5.1 Control Path Registers

A Motorola instruction consists of a single 16-bit word plus an optional set of extension words that are used to specify addressing mode information and displacement information in branching. As a result a 16-bit instruction register is provided and is loaded with the information on the internal data bus when the CWORD field IL is high. This register is the outputted to the instruction decoder which is responsible for CAR address generation.

The program counter is a 32-bit register, 20 bits of which generates a valid memory address for instruction and immediate operand fetching. It is implemented using a register and combinatorial logic that generates a load signal depending on the PL CWORD field. The PL can be incremented in one clock cycle by setting PL to be 01. When PL is 00 the register holds its value. If a 11 is set the PC is loaded with whatever data is on the internal data-bus. This is necessary to facilitate use of jumping and branch instructions that apply a displacement to the PC.

```
-- Source : Control.vhd
PCControl : process(PL_w,PCout_w,PCInc_out,Data_In)
begin
 case PL_w is
        when "00" =>
                MUX_P    <= PCout_w;        -- Hold PC
        when "01" =>
                MUX_P <= PCInc_out;         -- Increment PC
        when "10" =>
                MUX_P <= Data_In;           -- Load PC
        when "11" =>
                MUX_P    <= Data_In;        -- Load PC
        when others =>
                MUX_P <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
 end case;
end process;
```

When processing branching instructions such as BRA or BCC a signed extension word of 32-bits is used to specify a displacement to be added to, or subtracted from the PC. In order to facilitate this addition or subtraction in a quick manner, without having to use a temporary register in the file, a displacement register has been provided. Fed into MUX B, it is possible to select the displacement register as a B input to the functional unit, together with the PC into the A input from MUX

A and perform the subtraction or addition and subsequent write back to the PC in a single clock cycle.

2.5.2 Micro-sequencing

A micro-sequencer is the lynchpin of any micro-programmed architecture. It is responsible for the generation of the next CAR address thus defining the next set of control signals to be applied to the data-path.

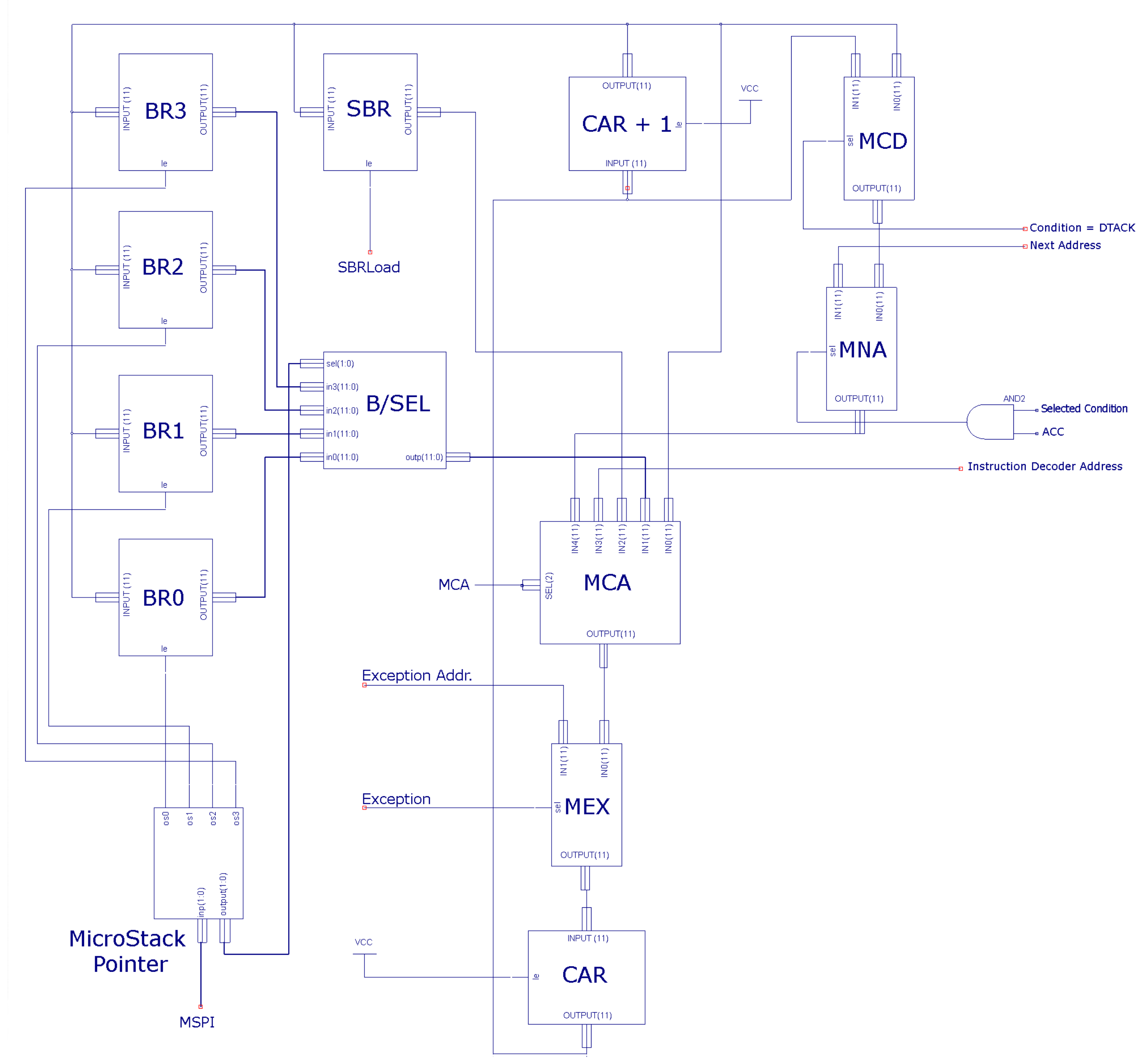


Figure 2.15: Microsequencer Block Diagram

The CROM has been designed as 4096 words deep, meaning a 12-bit address is required. This gives ample space for micro-coding the 68008 instruction set provided the micro-programs are written in a modular fashion and take advantage of common operations between code. For example, the micro-code for a memory write will be the same for every instruction write-back, therefore it makes sense to re-use commonly invoked micro-addresses. A 12-bit Control Address Register (CAR) holds the address of the currently executing microinstruction. The loading of the CAR is controlled by the CAR increment register and multiplexers MEX, MCA, MNA, MCD and B/SEL. At every stage the CAR increment register contains the value CAR + 1. The CAR is loaded on the positive edge of the clock, so it is necessary to make the CAR increment register negative edge triggered, to allow for propagation through the register and back to the CAR in a single clock cycle when a CAR increment is desired. The CWORD field MCA controls the multiplexer that loads the CAR. Table 2.6 details the effect of this field on the CAR.

MCA	**Function**
000	Increment CAR
001	Load From MicroStack
010	Load From Single Branch Register
011	Load CAR from Instruction Decoder
100	Load CAR using Motorola Conditions

Table 2.6: MCA Selection

The Micro-Stack and SBR

To aid the use of micro-routines a Micro-Stack architecture has been implemented. There are 5 registers provided in which are placed return addresses when implementing branching within the micro-routine. Registers BR0 to BR3 are part of the Micro-Stack and provide 4 levels of branching within micro-routines. The Micro-sequencer contains a two bit counter, the Micro-Stack pointer, which controls the loading and selection of these four registers. When calling a micro-subroutine the MSPI field must be set to 01 in order to increment the stack pointer. In the next clock cycle BRLoadControl must be set to 01. This causes the next CAR address + 1 to be placed into a specified load register. It is now possible to load the CAR with a micro-routine address specified by next address. During the execution of the sub-routine, the micro-stack can be used again for the same purpose. When returning from a subroutine it is necessary for the micro-routine to load the CAR with a return address. This is done by setting BRControl to 11 and MCA to 01. The address popped off the stack is equal to the calling address + 1. The calling micro-program must now decrement the stack pointer by setting MSPI to 11 before it can continue processing. This micro-architecture mimicks how a stack may be implemented in software. A pointer holds the location of the next point on a stack. A micro-stack operation i.e. push or pop, is an atomic micro-operation but in practice at this low level takes two clock cycles, once to ensure the pointer is at the correct location, one to perform a push.

This implementation provides a very powerful means of branching within microcode. For example, when an instruction is decoded the CAR is loaded with an address for the micro-program that must be executed. This micro-program will most of the time need to fetch operands on which to perform operations. If an effective address is designated as the source, one of the addressing modes must be used to fetch this operand. With the micro-stack implementation it is possible for an instruction micro-program to branch to an addressing mode micro-routine, e.g. fetch address register indirect, which will place the result in the EA Data register. When finished fetching operands the instruction and continue execution and write-back.

The Single Branch Register, or SBR is a branching register that is not part of the stack. Its intended use is for memory access. Since a memory access will never contain a branch, it is spared the need to use up 3 cycles manipulating the stack pointer and micro-stack. Instead it simply loads and unloads the Single Branch Data Register as necessary. This also means that even micro-routines with addresses at the top of the stack may use the Bus Interface for memory reading and writing. SL loads the SBR and when MCA is 10 the CAR is loaded with the value in the SBR.

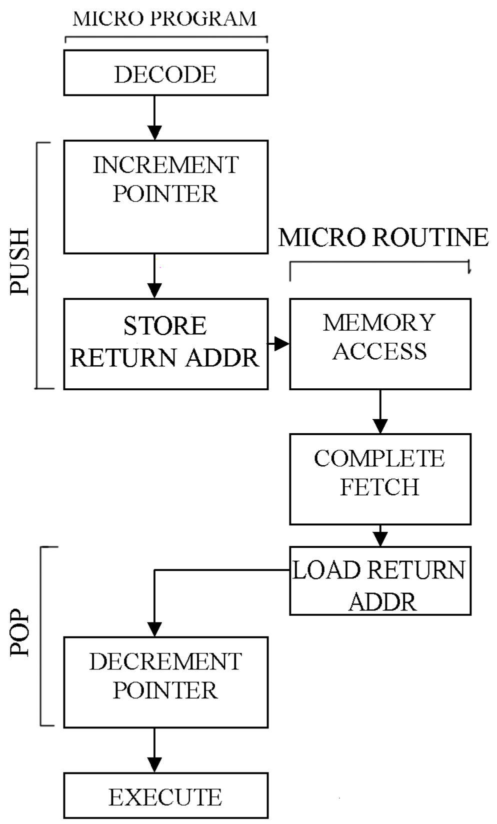


Figure 2.16: Micro-Branching Flow

Conditional Branching

MNA, MCD and the Next-Address input allow conditional loading of the CAR. It is necessary to support a variety of conditional load conditions that may arise from both the Micro-CCR and the User Byte of the SR. When the MCA is set to 100 the inputs CCOND and ACC decide what to load the register with next. Table 2.7 details the operational values of the 5 bit CWORD field CCOND. When performing a branch in a micro-routine it is necessary to provide the micro-sequencer with an address to load if the branch is taken. If the condition chosen is high at the time specified, and the allow condition (ACC) field is 1 then the CAR is loaded with a branch address from the NA field. If the condition is a zero at the time specified then the CAR is simply incremented.

Value	Description
00000	Branch Carry Set (Micro CCR)
00001	Branch Overflow Set (Micro CCR)
00010	Branch Zero Set (Micro CCR)
00011	Branch Negative (Micro CCR)
00100	Branch Carry Clear (Micro CCR)
00101	Branch Overflow Clear (Micro CCR)
00110	Branch Positive (Micro CCR)
00111	BCS Motorola (C)
01000	BVS Motorola (V)
01001	BEQ Motorola (Z)
01010	BMI Motorola (N)
01011	BCC Motorola (/C)
01100	BVC Motorola (/V)
01101	BNE Motorola (/Z)
01110	BPL Motorola (/N)
10011	Wait for Bus Interface Finished Set
10110	BHI Motorola (/C*/Z) Branch Higher
10111	BLS Motorola (C + Z) Branch Low or Same
11000	BGE Motorola (N*V + /N*/V) Branch greater than or equal
11001	BLT Motorola (N*/V + /N*V) Branch Less than
11010	BGT Motorola (N*V*/Z + /N*/V*/Z) Branch Greater Than
11011	BLE Motorola (Z + N*/V + /N*V) Branch Lest than or equal
11100	Force Load of NA

Table 2.7: CCOND Possible Branching Conditions and CCR Codes

In the shift-and-add multiplication example detailed previously it was suggested that the multiplicand be shifted left one place setting the carry to be a 1 if the LSB is a 1. It was suggested that a branch be taken if the carry is not set to avoid adding a partial product un-necessarily. The following describes how this is implemented using the micro-sequencer and the Micro-CCR Carry and Zero conditions.

```
ADDRESS CCOND     MCA        NA          DESCRIPTION
```

```
$4     0000   000   N/A    —— Shift the multiplier left
$5     1010   100   $7     —— Branch if Carry Clear to $7
$6     0000   000   N/A    —— Otherwise add partial product
$7     0000   000   N/A    —— Shift the sum of products left
$8     0000   000   N/A    —— Subtract 1 from a loop counter
$9     1010   100   $4     —— Branch Zero Clear to the start
$10    1010   100   $11    —— Write Back Operation using BI
$11    0101   100   $24    —— Jump to fetch next instruction
```

There is one exception to this general application. When CCOND is set to wait for DTACK, the current CAR address is reloaded, rather than CAR + 1. This is designed to allow the bus interface to complete its cycle fully. In this instance the next CAR address must be specified by using the Next-Address input. When performing a bus cycle that requires the insertion of wait stages it is necessary to have the address of the next instruction to be executed in the next-address field.

Handling Bus Error and Halt Exceptions

The final multiplexer feeding the CAR is concerned with the loading of exception handlers. There are three possible exception handler addresses that may be loaded into the CAR. If a Bus Error occurs, the bus interface will generate the exception signal. This causes the CAR to be loaded with the bus error exception handler. This handler can simply load the vector register with the bus error vector number. This loading in necessary since if the BI is active, the car NA will remain stationary. The only other exception that is supported using this method of loading is the HALT handler. A programmer may wish to push information to the stack before entering the halt mode.

An Internal Exception field IE has been placed in the CWORD for future use. The use of this function is to enable the generation of exceptions within micro-programs. The Motorola generates exceptions upon the results of certain operations. For example a divide by zero will cause and exception handler to be run. Similarly if the TRAP instruction is called a handling program must be loaded. The IE field allows these software exception handlers to exploit commonality. Each exception may place its exception handler address on the stack and push high the IE field. This IE field may load an exception handler that pulls this vector address off the system stack and processes as necessary.

2.5.3 Instruction Decoding

An MC68000 Instruction consists of at least one 16-bit word. In order to facilitate various addressing modes there may be up to 11 extension words. These words may contain information such as immediate operands, source and destination effective address information or branching displacement information.

An instruction may specify zero, one or two effective addresses within an instruction word. An effective address is a 6-bit substring of the full instruction word. It provides information as to the source and/or destination of operands as well as the addressing mode that is used in order to fetch those operands. The majority of instructions will use a single effective address which may specify the destination of the operation or the source operand.

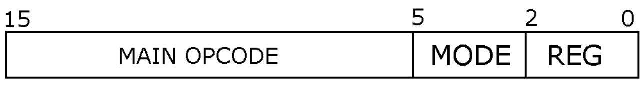


Figure 2.17: Effective Address Field

The mode field of the effective address gives information as to the addressing mode of the effective address. The register field will either specify a register in which the operands or information about the operands is contained. It may also contain information relating to addressing mode. Table 2.8 shows how the effective address may be decoded.

MODE	**REGISTER**	**DESCRIPTION**
000	xxx	Data Register Direct
001	xxx	Address Register Direct
010	xxx	Address Register Indirect
011	xxx	Address Register Indirect with Postincrement
100	xxx	Address Register Indirect with Pre Decrement
101	xxx	Address Register Indirect with Displacement
110	xxx	Address Register Indirect with Index
111	000	Absolute Short
111	001	Absolute Long
111	010	PC indirect with Displacement
111	011	PC indirect with Index
111	100	Immediate

Table 2.8: Decoding an EA

When using the immediate addressing mode the immediate data is specified in an extension word. When using the displacement or index addressing modes the extension words will contain the displacement that is needed to perform the necessary operation. Figure 2.18 shows how a MOVE instruction is encoded and stored in memory. The MOVE is the only instruction that may have two effective addresses. Other instructions must store the results in destinations specified elsewhere in the instruction.

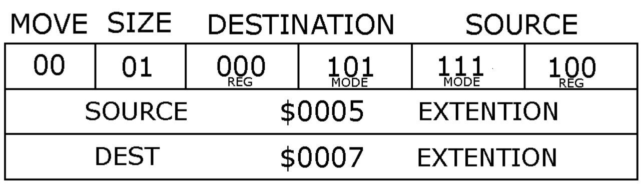


Figure 2.18: MOVE.B #5,(A0) Instruction

Bits 15 down to 6 of a 68000 instruction are used for decoding the instruction type. Bits 15 down to 12 can be used to form a loose grouping of instructions. For example, 0000 in bits 15 down to 12 will indicate that the instruction is either an immediate instruction, such as ANDI, or a bit manipulation instruction such as BTST. As detailed in figure 2.19 it is possible to break up the remainder of the instruction in to well defined pieces in order to work out what instruction is being executed. In fig 2.19 bit 7 differentiates between Bit Manipulation and immediate instructions. Bits 11-8 specify the sub-group of these instructions and the remainder specifies operand size and effective address. For example when 8 is 0 and OPC is 000 we are dealing with an ORI instruction, a member of the immediate instruction functional group. Not all instruction formations are as regular but a similar breakdown can be applied to the majority of instructions.

The form 1111 of the leading 4 bits is reserved for interrupt processing, but the remainder will specify one of several instruction groupings such as the Immediate/Bit-manipulation grouping. The accompanying CD-ROM contains an excel spread sheet which is a guide to the instruction groupings proposed and used in this project.

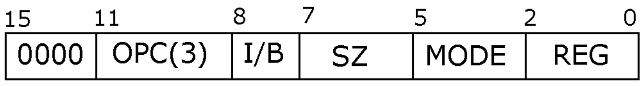


Figure 2.19: Instruction encoding for group 0000

If the instruction is a two operand instruction, bits 11 down to 9 will always specify a register number. How this number is interpreted depends on the instruction. For example, ADD (ea),Dn will interpret this register as the source for the second operand in the addition, and also the destination for result write-back. In this instance the EA will specify the source operand. Size is encoded in one of two ways. Bits 7 and 6 (SZ) contain size information for applicable instructions except in the case of MOVE where bits 13 and 12 (SZ2) dictate operation size.

The Instruction Decoder

A mapping ROM with an address manipulated by combinatorial logic is used to provide the CAR with CROM addresses depending on the operation desired. The select input ISEL controls the operation of the decoder and allows decoding of different parts of the instruction such as the EA and the main op-code. The instruction decoder will provided CAR addresses in one of four ways. Firstly, it will provide the CAR address for the execution of one particular instruction. In this instance the CWORD field ISEL is set to 000. The mapping ROM address is constructed from the instruction word bits 15 down to 6 concatenated with a 0. All mapping ROM addresses that start with 0 are responsible for the decoding of the function of an instruction. For example mapping ROM location 00000001000 contains the CAR address that will run the micro-program dealing with the ANDI instruction.

When the ISEL input is 001 the instruction decoder is performing a decode on an EA and is using this EA to fetch operands. 100—SZ—EA is fed into the mapping ROM giving the location of the required address. For example, location

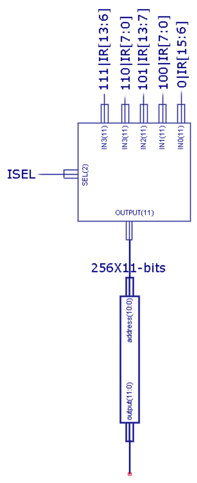


Figure 2.20: Instruction Decoder and mapping ROM

100—00—010000 will contain the CAR address for a byte-operand fetch using the address register indirect addressing mode.

ISEL 010 is identical to the above operation except it performs decoding on bits 11 down to 6 for a move instruction. Locations for such CAR addresses are produced by the word 101&SZ2&EA. 011 and 100 operate in a similar fashion except they are responsible for supplying the CAR with addresses necessary for operand write back when the effective address is a destination.

The locations of thee words in mapping ROM and CROM memory are done in Xilinx through the use of constants. For example, the write micro-routine.

```
-- Source : CROM4096x64.vhd
-- Declaration
constant Read_Byte_Wait       : std_logic_vector(63 downto 0) := ...
constant Read_Byte_Latch      : std_logic_vector(63 downto 0) := ...
constant Read_Byte_Return     : std_logic_vector(63 downto 0) := ...
-- Mapping
when "111111110011" => output <= Read_Byte_Wait;
when "111111110100" => output <= Read_Byte_Latch;
when "111111110101" => output <= Read_Byte_Return;
```

2.5.4 The Control Word

The CROM is a 4096 deep ROM that produces control words of 64-bits in length. It is possible to restrict the size of the word to 64-bits by defining the word three forms shown in figure 2.21. The first form of the word is intended for operation of the system that does not involve jumping or branching. In this instance bit AA is 0 and bits 22 down to 17 influences loading of control path registers.

In order to use the functional unit and the system control registers it is necessary to use this form of the CWORD. The second form of the word sees the inclusion of an 8 bit vector number. This form is only used for the decoding of an 8-bit vector

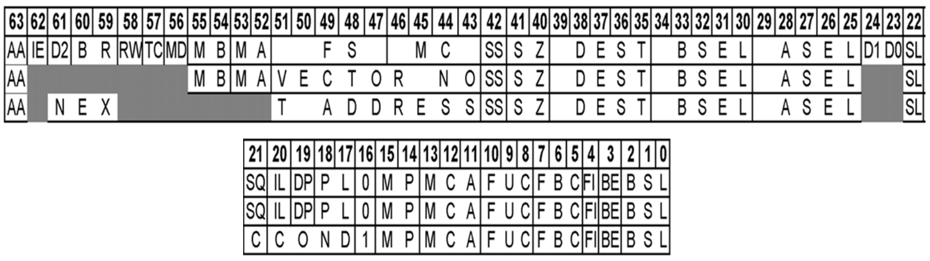

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22
AA IE D2 B R RW TC MD M B M A F S M C SS S Z D E S T B S E L A S E L D1 D0 SL
AA M B M A V E C T O R N O SS S Z D E S T B S E L A S E L SL
AA N E X T A D D R E S S SS S Z D E S T B S E L A S E L SL
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SQ IL DP P L 0 M P M C A F U C F B C FI BE B S L
SQ IL DP P L 0 M P M C A F U C F B C FI BE B S L
C C O N D 1 M P M C A F U C F B C FI BE B S L

Figure 2.21: CWORD Formats

number which is subsequently loaded into the Vector Address register for execution of an exception vector. The third form of the control word sees the inclusion of the CCOND field and the next address field. The CCOND field selects the condition for branching and the next address specifier will give the next address if a branch is taken. The functional unit field have been used in this instance since when functional operation is used in a micro-program it will always occur before the branch and never in the same clock cycle.

Table 2.9 gives a breakdown of the function each field performs. Appendix A gives a full list of the valid values of each field.

FIELD	**BIT NUMBERS**	**FUNCTION**
BSL	2:0	Bus Interface Operation Select
BE	3	Bus Interface Enable
FI	4	Processing a Bus Fault
FBC	7:5	R/File Port Control Select
FUC	10:8	R/File A Port/Destination Control Select
MCA	13:11	MicroSequencer CAR Address Source Select
MP	15:14	MicroStack Pointer Control
ACC	16	Allow Condition Codes to effect CAR
PL	18:17	Program Counter Load Control
DP	19	Displacement Register Load Enable
IL	20	Instruction Register Load Enable
CCOND	21:17	Branch Condition Select
SL	22	Single Branching Register Load Enable
D	61—24:23	Instruction Decoder Function Select
ASEL	29:25	Register File Port A Select
BSEL	34:30	Register File Port B Select
DEST	39:35	Register File Destination select
SZ	41:40	Operation Size Select
SS	42	CWORD/Instruction for Size
MC	46:43	System Register Load Control
FS	51:47	Functional Unit Operation Select
Vector No	51:44	8-bit Vector number
Next Address	61:59—51:43	Next Address Loader
MA	53:52	Mux A Select
MB	5:54	Mux B Select
MD	56	Mux D Select
TC	57	Micro-CCR Load Enable
RW	58	Register File Read Write
BR	60:59	MicroStack Load/Unload Control
IE	62	Generate Internal Exception
AA	63	Allow Next Address field

Table 2.9: CWORD Field Function

Chapter 3

Micro-programming the design

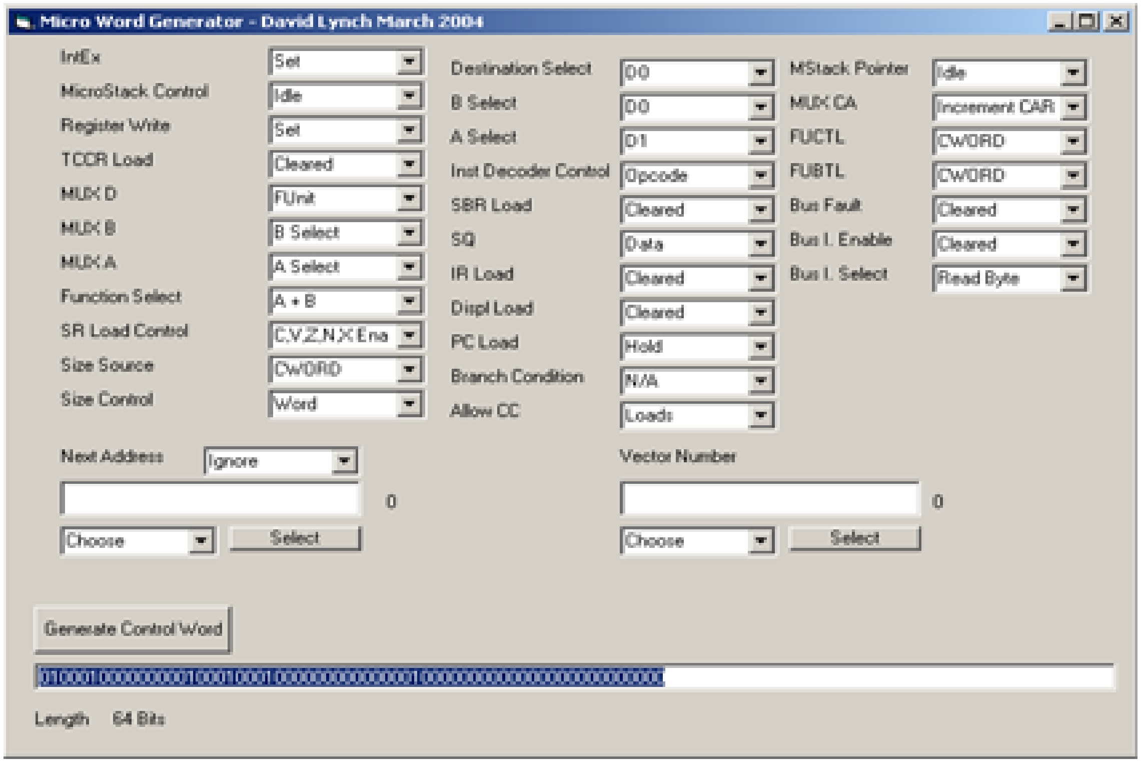
Once all needed hardware was in place and individually tested was necessary to consider micro-coding. This chapter explores the micro-programs necessary to control the system by specifying how micro-program algorithms may be implemented using the hardware in place.

3.1 Control Word Generation

Micro-coding a 64-bit word is a tedious, error prone task when done by hand. To combat this tedium and to decrease the chance of errors during CPU operation a simple Visual Basic application has been developed to automatically generate control words given specified inputs. The generator has been programmed in Visual basic because of its ability to quickly generate forms and automatic generation of essential MFC code.

The application contains a series of drop-down boxes detailing possible values for each control word field. The user specifies the desired value for each field and the application constructs the appropriate CWORD, which then can be copied to the Xilinx application or ROM bit file.

Operation is intuitive for programming a CWORD of the standard form. However, when programming branching and vector addresses there are some necessary omissions. In keeping with the CWORD format, the fields Micro-Stack Control(BR), Function Select, SR Load Control, SQ, IR Load, Displ Load and PC load must all remain blank. Allow CC must be set to conditions for a conditional branch. This field can be ignored when supplying a vector number to the vector address register. Next address locations and vector numbers can be inputted into the provided text-boxes. Although not yet implemented, it is intend that a drop-down list of commonly used next CAR addresses, such as instruction fetch and memory accesses will be able to provide next address values. Similarly, it is intended the complete 68008 memory addressing list be added.


Micro Word Generator - David Lynch March 2004

IntEx	Set	Destination Select	D0	MStack Pointer	Idle
MicroStack Control	Idle	B Select	D0	MUX CA	Increment CAR
Register Write	Set	A Select	D1	FUCTL	CWORD
TCCR Load	Cleared	Inst Decoder Control	Opcode	FUBTL	CWORD
MUX D	FUnit	SBR Load	Cleared	Bus Fault	Cleared
MUX B	B Select	SQ	Data	Bus I. Enable	Cleared
MUX A	A Select	IR Load	Cleared	Bus I. Select	Read Byte
Function Select	A + B	Displ Load	Cleared		
SR Load Control	C,V,Z,N,X Ena	PC Load	Hold		
Size Source	CWORD	Branch Condition	N/A		
Size Control	Word	Allow CC	Loads		

Next Address: Ignore — 0 — Choose — Select

Vector Number: 0 — Choose — Select

Generate Control Word

0100010000000001000100010000000000000100000000000000000000000000

Length 64 Bits

Figure 3.1: Generating a CWORD to Add D0 to D1

3.2 System Control Programming

In order for the system to function correctly a number of system control micro-programs must be implemented. These micro-programs provide a skeleton set of micro-routines on which actual micro-programs can be constructed.

3.2.1 Memory Access

Since the first instruction of any program will be held in memory the first micro-program necessary is one that will control memory access. All memory accesses in the system are implemented using the memory access micro-routines. It is intended that these routines be called from micro-programmes that require memory access, such as memory operand fetch or instruction fetch. There are routines for memory write and read of each size of operand.

12-bit Hex Address	Routine
FF3	Read Byte
FF0	Read Word
FC0	Read Long-Word
FC8	Write Byte
FCA	Write Word
FCC	Write Long-Word

Table 3.1: Memory Access CROM Locations

Each of the memory reads takes three micro-instructions while the memory writes take two. The number of clock cycles taken depends on the size of the bus cycle and the number of wait stages inserted by the bus interface.

The reserved memory address register provides the Bus Interface with a location from which to read a memory address. If the operation is a write then the result is placed in the reserved memory data register.

The first micro-instruction sets the bus interface to enable and waits for the PDTACK signal. For example when reading a byte, the BIE field is set to 1 and the BI field is set to 000 for a read. The third CWORD form is used and the branch condition is PDTACK. The next address is fed from the CWORD and is set to FF4 in order to load the next micro-instruction when the data has been received from the peripheral device. MCA is set to next address in this instance. For memory read an address must be provided from the register file. Similarly source data must be provided for a write. MUX A must be set to transfer an address and MUX B must transfer the necessary data to the bus interface

Upon assertion of PDTACK the second micro-instruction is loaded. This instruction is responsible for the reading of data from the bus interface. The memory data register must be selected as the destination register and the register file must be set to CWORD control (FUCTL = 000, FUBTL = 000). Register write must also be enabled. The size field must be set to 00 for byte and the SS field must indicate that size selection is under CWORD control. The last micro-instruction is common to all memory access subroutines and is a return from routine instruction. This instruction simply loads the CAR with the value located in the single branching register, thus returning to the calling micro-program.

3.2.2 RESET Handler

When the 68008 is powered on for the first time, or is reset it must initialise the PC and the system stack pointer. The first long-word of memory, location 0, will contain the initial system stack pointer. The second long-word will contain the initial PC.

In the 68008 VHDL model the PC is reset to 0 by the nRESET signal. It therefore makes sense to use the PC as a pointer to retrieve the initial stack pointer and the initial PC. Branching to the memory micro-routine is not used in this instance as we intend to us the PC as a memory address pointer. Ten instructions are required the first of which is located at CROM location 0.

We must initialise the bus interface and wait for PDTACK as if performing a memory read. On the next micro-instruction we are required to load the SSP. This is done by setting the register read field to write and selecting system stack pointer as the destination. The same must be done for user stack pointer, as, although in theory the same register, they are represented in hardware by two different registers. The PC is then incremented 4 times by setting PL to load for 4 consecutive instructions.

The next bus interface cycle required field is a read-long word for the PC. This accomplished using the initial PC as a source like before. On the next instruction it is necessary to set the PL field to *Load* and MUX D to *External* in order for it to latch data fed from the bus interface. The final instruction simply jumps to the location of the Instruction Fetch micro-routine.

3.2.3 Fetch-Decode-Execute Sequence

All processors need to perform this cycle in order to fetch and execute instructions. Located at CROM location 0x00F the instruction fetch micro-programme operates in exactly the same way as the initial PC fetch routine except a 16-bit word is fetched and the IL field must be set to load instead of PL.

Once the instruction register has been loaded the PC is incremented twice to point to the potential location of the next instruction in memory. It is now desirable to jump to instruction decode. In order to do this the third CWORD form must be used and the location of the ID routine, or 0x808, must be fed to the next-address microsequencer port. MCA is set to Load Next Address, ACC to Conditions and the condition field CCOND must be set to NALOAD. The CAR is now loaded with the ID routine.

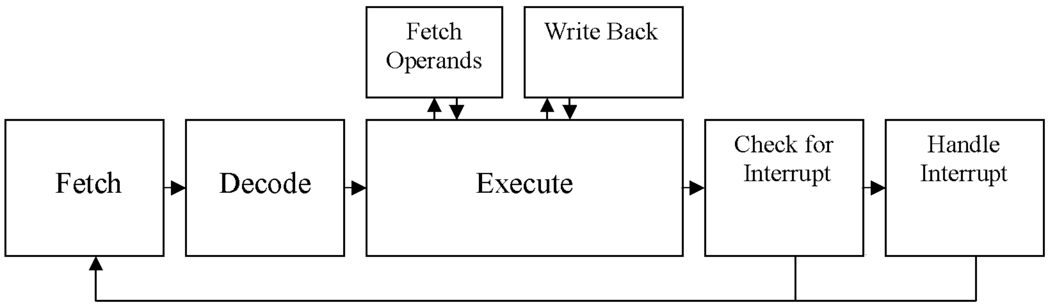


Figure 3.2: The Fetch-Decode-Execute Cycle

The ID routine itself is a simple jump. MCA is set to Instruction Decoder and the ISEL field is set to Opcode. The instruction decoder will decode the instruction and provide a location for its execution which is then fed into the CAR.

The processor is now in the execution phase. Due to the varied nature of the location of operands in the 68008 instruction set, and to the fact that not all instructions may access all addressing modes, each execution micro-program is responsible for the fetching and write back of all operands as well as instruction execution.

Interrupts

Once write back has been completed it is necessary to check for interrupts. Any interrupt requested during execution is processed by the IPL control module. If multiple interrupts are requested during an instruction, the highest level one is considered only.

The CWORD field SQ allows the register file to be loaded with the value emanating from the IPL control. Temporary register T0 is loaded with this value. On the next clock cycle this register becomes an A input to the ALU by setting ASEL to T0 and MUX A to 00. MUX B IPL select will provide the ALU input with the current interrupt level. A subtraction is performed and the MicroCCR is set using the appropriate signals for TC and FS.

If the result of the operation is negative the current interrupt level is higher and the interrupt is rejected. This is accomplished by setting MCA to Next Address and the condition in question set to branch not negative from MicroCCR. If N is set it will cause a jump over the instruction that deals with processor interrupts and back to instruction fetch for the next instruction.

If this jump is not taken then a jump to interrupt processing will take place. The procedure is as follows

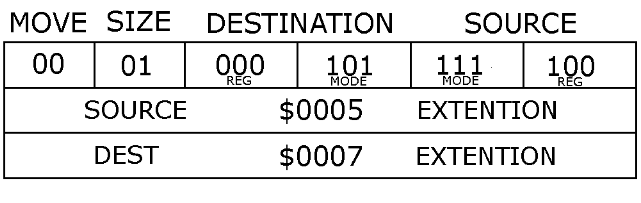
- Save the status register to the system stack.
- Save the PC to the system stack.
- Perform an interrupt acknowledge cycle using the Bus Interface.
- Obtain vector and load to PC.
- Return from interrupt (RTI) is called and original PC and Stack Pointer are restored.

3.3 The MOVE instruction

This section explains in detail how the MOVE instruction is implemented and defines the process by which all micro-programs should be written. It includes details of the addressing mode fetch and write-back micro-routines implemented to date. The MOVE micro-program is located at 0xC20 in the CROM and is 7 words deep.

3.3.1 Decoding

In order to provide the CAR with the correct address for the MOVE instruction it is necessary to place that address in the decoder mapping ROM. Since we are decoding a main instruction, the MSB of the mapping ROM address is 0. The Motorola programmers manual specifies the first two bits of an instruction to be 00. We firstly want to deal with word operands, therefore the next two bits are 11. The MOVE instruction form is shown again in figure 3.3. It is necessary to place 0xC20, or the MOVE micro-program address, at locations 00011xxxxxx of the mapping ROM in order for correct decoding to be performed for all addressing modes for the destination.



MOVE	SIZE	DESTINATION		SOURCE	
00	01	000 REG	101 MODE	111 MODE	100 REG
SOURCE		$0005		EXTENTION	
DEST		$0007		EXTENTION	

Figure 3.3: MOVE Instruction Form

3.3.2 Fetch using Addressing Modes

Once the MOVE instruction micro-program address has been decoded the micro-sequencer will start executing the MOVE micro-program itself. As previously explained, it is necessary to fetch the source operand for the operation. As a result the first 2 instructions of the MOVE routine simply increment the MicroStack pointer by setting MSPI to 01 and push a return address on the MicroStack by setting the BRLoad field to 01. It is necessary to decode the source effective address for a MOVE so 101 is fed into DSEL field.

Five forms of operand fetch addressing modes have been implemented in micro-coding and hardware support exists for the remainder. These are Data Register Direct (CROM location 0x03F), Address Register Direct (0x041), Address Register Indirect (0x043) and Immediate (0xE00).

All addressing mode micro-routines will use at least one of the registers reserved for effective addresses and the memory read and write registers. Taking the Immediate addressing mode as an example it is possible to show how a typical fetch micro-routine will operate.

Immediate operand fetch will move the data encoded in an instruction extension word to the Effective Address Data register. An immediate operand of length 16-bits is located in memory at the address immediately following the core instruction word. The instruction op-code for an immediate fetch is encoded in the lower effective address. Motorola uses 111 for mode and 100 for the register field to indicate an immediate data addressing mode. Instruction decoder mode Fetch for Move is used and therefore the micro-routine address must be located at address 10111111001. When source effective address decoding has taken place the Immediate micro-routine is in execution. A Bus read cycle for a word operation must be performed. As before the DTACK condition is fed into the CCOND field, AA is set to 1, bit 16 is set to 1 and the next address (the location of the next micro-instruction) is fed into the NA field. After the bus cycle the register Effective Data is written. The PC must now be incremented twice following which the micro-routine will return to the MOVE instruction by setting BRLoad to 11 and MCA to 001.

MOVE3 is concerned with setting the correct condition codes. It is necessary to circulate the Effective data Register around the ALU using the Transfer function in order to set the correct condition all condition codes except eXtend are affected. MUX A simply reads the register circulates as the A input of ALU and MC is set to allow all codes except X to be set.

3.3.3 Write Back

Most instructions give the option of using the effective address as a destination instead of a source. The MOVE instruction allows both to be an EA. In either case a write-back micro-routine must be called.

Write back routines have been implemented for the MOVE instruction only. These include Data Register Direct (CROM location 0xE05), Address Register Direct (0xE07), Address Register Indirect (0xE09), Address Register Indirect with Post Increment (0xE11) and Address Register Direct with Pre-Decrement (0xE18).

Decoding is performed by setting ISEL to 100. The address for the write-back micro-routine must be put in its correct mapping ROM place for EA write back on a MOVE instruction. ”111” & SZ2 & EA2 is the address string that will determine this. SZ2 is instruction bits 13 down to 12. EA2 is the destination effective address in a move instruction and consists of bits 11 down to 6. For example, data register direct write back for a MOVE.W instruction must be located at mapping ROM addresses 11111000xxx.

The write back routine will expect to find data to be written back in the Effective data register. If an address is necessary then it is expected to be located in the Effective Address register. All memory-modifiable addressing modes will move this data to the Memory Data and Memory Address register before branching to a Bus-write cycle.

3.4 The MULU instruction

The MULU instruction performs a multiply on two, at maximum, 16-bit unsigned numbers and stores the result. This instructions micro-coded implementation is explored in order to give an example of how one might implement a complex instruction that requires multiple-clock cycles to execute. Operand fetch and write back for MULU are performed in a similar way to that of the MOVE instruction and have not been considered here. The MULU instruction has been located at CROM 0xA0F and requires thirteen instructions six of which are repeated sixteen times for a total of 97 clock cycles. It is noted here that a hardware multiplier can complete such a task in less clock cycles however micro-coding instructions this way is consistent with the CISC design approach and is good way of testing correct function of the control and sequencing logic.

3.4.1 The Shift-and-Add Algorithm

The Shift-and-Add algorithm of binary multiplication is used in this example. This method adds the multiplicand Y to itself X times, where X denotes the multiplier.[15, Zargham CH:2 S:2.5.2].

The first task of shift-and-add multiplication is to set up the temporary registers necessary to perform the operation. The multiplicand is placed in T0 and the multiplier in T1. T3 is used as an accumulator register and stores the results of partial products at every stage of the multiply. A constant value of 16 is placed in register T8. This will count the number of iterations necessary for the multiplication.

MUL4 shifts the multiplier right and allows the MicroCCR condition codes to be set. This is done by choosing logical shift right for the FS field and setting TC to 1. The result is stored back in register T1. MUL5 uses the branching form of the CWORD, sets Branch Carry Clear to be the condition in the CCOND field, bit-16 to be a 1, AA to be 1 and the location of MUL7 to be the next address. If the MicroCCR has shifted a 1 into the carry then MUL6 will be executed. MUL6 simply adds T0 to T3. If carry is clear then MUL6 is not executed and MUL7 is the next address. MUL7 shifts T0 left one. MUL8 subtracts 1 from T8 and sets the MicroCCR again. MUL9 is a branching Micro-Instruction that returns to MUL4 if

T8 is not zero, otherwise proceeds to MUL10 which is responsible for operand write back.

```
-- Source : CROM4096x64.vhd
constant MULU_S ...     -- Clear accumulator register
constant MULU_0 ...     -- Move 8 to Temp 8
constant MULU_1 ...     -- Move source to Temp 0
constant MULU_2 ...     -- Move destination to Temp 1
constant MULU_3 ...     -- Move Temp 0 to Temp 2
constant MULU_4 ...     -- Logical Shift right T1 setting MicroCCR
constant MULU_5 ...     -- Branch Carry Clear to MULU_7
constant MULU_6 ...     -- Add Temp 0 to Temp 3 store in temp 3
constant MULU_7 ...     -- Logical Shift Left Temp 0
constant MULU_8 ...     -- Subtract 1 from Temp 8 set c. codes
constant MULU_9 ...     -- Branch if Temp 8 not Equal to 0 to MULU_4
constant MULU_10 ..     -- write back to dest. Setting codes
```

3.4.2 Other Instructions

Other instructions have been implemented in exactly the same way using the Micro-Word generator and using the following technique.

- Assign a CROM Location.
- Ensure correct decoding of the instruction suggested groupings.
- Use the Micro-Word Generator to generate micro-instructions.

Other implemented instructions to date are JMP, ADD, SUB (for Direct addressing) and BMI (branch on minus).

Chapter 4

Testing and Conclusions

This chapter details what has actually been accomplished during the project by exploring some test results and stepping through a demonstration of a 68000 assembly program. Suggestions for further avenues of work on this project are also explored.

4.1 Design Testing and Development Progress

All design testing has been performed using ModelSim. A simulation on this platform allows the designer to isolate any wire, register or signal contained within the design. This very powerful tool was a vital component in debugging the operation of the VHDL CPU and

Each individual module has been tested using test-benches designed to verify the functionality on all possible inputs, or when that is not feasible, a set of inputs that practically verifies module operation. Top-level testing was accomplished through strategic implementation of micro-code. As micro-coding the system for the full instruction set is a time consuming task, a focus has been placed on using selected micro-coded applications to test the fully inter-connected system. As a result the micro-code in place to-date will verify the inter-operation of all the modules. For example, MULU will test branching and sequencing, MOVE the addressing mode capability and the bus interface and RESET the handling of exceptions.

Most of the problems encountered throughout this project were results of implementation mistakes. For example, an early implementation of the control path attempted to specify the unit in such a way that the Xilinx synthesis tool would automatically infer control logic. This proved to be an error-prone and unreliable way of implementing control as the design was unnecessarily complex and hard to follow. It was at this early stage that the decision to keep to an RTL level of abstraction was made.

Other problems stemmed from common mistakes such as incorrect behavioural implementation results. In an example the ALU was not generating proper results and condition codes for the subtract function. The ModelSim platform enabled examination of each part of the design to isolate this problem to a programming mistake.

As previously outlined, the project has been developed to a stage where all necessary hardware needed for a VHDL op-code compatible CPU has been put in place. All modules have been tested and confirmed operational under behavioural and post-synthesis simulation. In addition a substantial subset of the micro-coding necessary for full instruction set decoding has been completed and tested under simulation.

4.1.1 Synthesis

A successful synthesis has been performed on the design using the Xilinx synthesis tool targeting the Virtex-II VC1000-256fg FPGA. A synopsis of the synthesis report can be found below.

The design uses 3,131 slices of Virtex-II resources, or approximately 61% of the slice resources available to the FPGA. In comparison, the modified LEON core utilises approximately 62

```
Design Summary:
       Number of errors:                 0
       Number of warnings:               4
Number of Slices:                        3,131 of 5,120 (61%)
       Number of Slices containing
        unrelated logic:                 0 of 3,131 (0%)
Total Number Slice Registers:            1,267 of 10,240 (12%)
       Number used as Flip Flops:        1,233
      Number used as Latches:            34
Total Number 4 input LUTs:               5,547 of 10,240 (54%)
       Number used as LUTs:              5,495
       Number used as a route-thru:      52
       Number of bonded IOBs:            105 of 172 (61%)
       Number of Tbufs:                  20 of 2,560 (1%)
       Number of GCLKs:                  1 of 16 (6%)
Total equivalent gate count for design: 47,675
Additional JTAG gate count for IOBs:     5,040
Mapping completed.
```

The gate count for the designs differ dramatically however, with the LEON core using almost three times the number of gates used by the 68008 design. This is most probably due to the more complex pipelined logic of the LEON core, which has hardware such as a floating point unit which are not found in the 68008 model.

Implementation timings are shown below. The maximum clock period attainable by the current design is 11.718Mhz. The FPGA project board has an 8Mhz clock port, therefore the design is more than capable of handing this clock speed. It is noted that it may be possible to increase this clock speed by examining the design for critical paths and re-organising the design to suit, however, as previously outlined, the nature of the project requires the 68008 model to be functional rather that quick and work emphasis has been placed on this functionality to suit.

```
        Minimum period: 85.342ns (Max Frequency: 11.718MHz)
        Minimum input arrival time before clock: 13.712ns
        Maximum output required time after clock: 88.281ns
        Maximum combinational path delay: 10.146ns
```

4.1.2 A 68008 demonstration

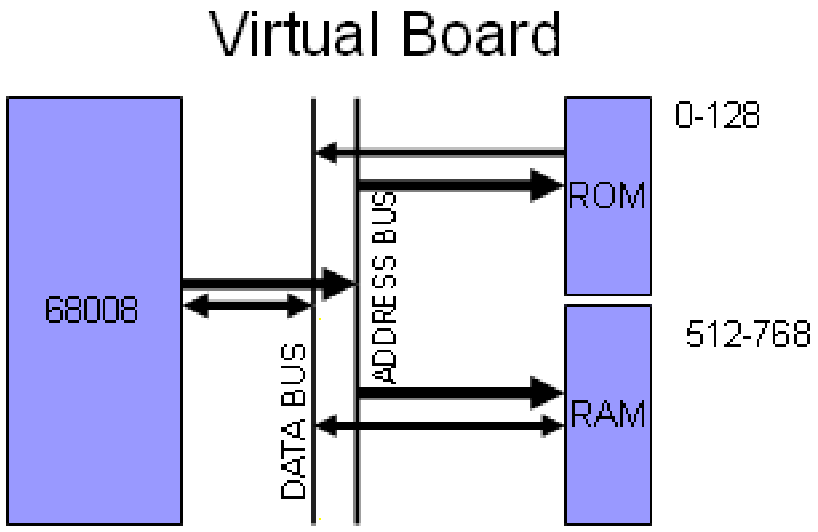


Figure 4.1: The Virtual Board Simulation Setup

For demonstration purposes a 68008 project style environment has been modelled in VHDL. A 68008 has been instantiated within this model and a ROM and RAM compatible with the 68008 bus interface have both been memory mapped to the specified locations through proper generation of chip enable signals as would be done by a student in the Architecture project outlined in the introduction. The ROM has been loaded with an initial stack pointer, an initial PC and a demonstration program.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 20 \\ 43 & 50 \end{bmatrix}$$

Figure 4.2: Multiplication of 2D Matrices

A simple 2D matrix multiplication program has been written in 68000 assembly language and assembled using the 68kasm cross assembler. The first couple of instructions are placed to demonstrate branching at instruction level and the correct generation of condition codes under a subtraction. 2 is subtracted from 1 and a branch is taken if the CCR correctly identifies the result is a negative number. Next, 512, or the first location in RAM is placed in an address register with the view of using it as a stack pointer. The matrix multiplication routine will take the matrices outlined in figure 4.2 and multiply them together yielding a result. When each result is generated it is pushed onto the stack by using the move instruction which utilises the indirect-with-post increment addressing mode to ensure correct stack operation. The following listing shows the program as implemented in assembler on the right and shows the raw machine code as generated by the assembler on the left.

```
                       1
00000008                       2       org $08
                       3
                       4
                       5  BEGIN
00000008  303C 0001    6       move.w   #1,d0
0000000C  323C 0002    7       move.w   #2,d1
00000010  343C 0003    8       move.w   #3,d2
00000014  307C 0200    9       move.w   #512,a0
00000018  9041         11      sub.w    d1,d0
0000001A  6B00 0002    12      bmi      MATRIX
0000001E  303C 0001    14  MATRIX: move.w       #1,d0
00000022  323C 0005    15      move.w   #5,d1
00000026  C2C0         16      mulu     d0,d1
00000028  343C 0002    17      move.w   #2,d2
0000002C  363C 0007    18      move.w   #7,d3
00000030  C6C2         19      mulu     d2,d3
00000032  D641         20      add.w    d1,d3
00000034  30C3         21      move.w   d3,(a0)+
00000036  303C 0001    23      move.w   #1,d0
0000003A  323C 0006    24      move.w   #6,d1
0000003E  C2C0         25      mulu     d0,d1
00000040  343C 0002    26      move.w   #2,d2
00000044  363C 0008    27      move.w   #8,d3
00000048  C6C2         28      mulu     d2,d3
0000004A  D641         29      add.w    d1,d3
0000004C  30C3         30      move.w   d3,(a0)+
0000004E  303C 0003    32      move.w   #3,d0
00000052  323C 0005    33      move.w   #5,d1
00000056  C2C0         34      mulu     d0,d1
00000058  343C 0004    35      move.w   #4,d2
0000005C  363C 0007    36      move.w   #7,d3
00000060  C6C2         37      mulu     d2,d3
00000062  D641         38      add.w    d1,d3
00000064  30C3         39      move.w   d3,(a0)+
00000066  303C 0003    41      move.w   #3,d0
0000006A  323C 0006    42      move.w   #6,d1
0000006E  C2C0         43      mulu     d0,d1
00000070  343C 0004    44      move.w   #4,d2
00000074  363C 0008    45      move.w   #8,d3
00000078  C6C2         46      mulu     d2,d3
0000007A  D641         47      add.w    d1,d3
0000007C  30C3         48      move.w   d3,(a0)+
0000007E               55  END
No errors detected
No warnings generated
```

This program executed successfully. The timing diagram below details the first few microseconds of the program operation. Cycle 1 shown in the diagram is the microprogram run to fetch the initial SP and PC from the ROM. Note how the CAR will stay at location 001 until the bus interface has received and processed the second DTACK signal from the ROM. The data-bus and address bus show the data being transferred from the ROM is pulled low. The initial SSP is 768 (the top of the RAM). The initial PC is the next thing to be fetched and this is set to 8 or the location of the first instruction.

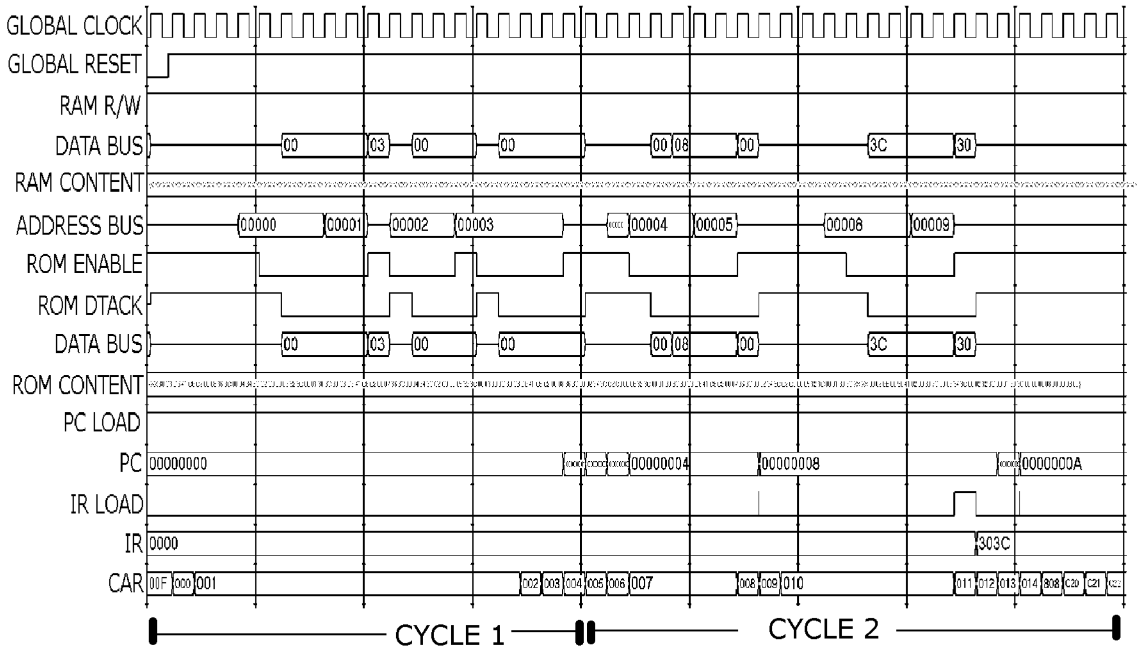


Figure 4.3: A timing diagram for the first 9ns of operation

The second cycle highlighted sees the CAR perform the Fetch and decode stage of execution. At CROM location 0x009 the instruction fetch micro-program resides. The diagram demonstrates a word-sized bus interface read cycle. Once this has been performed the IR is loaded with the first instruction, 303C is the first MOVE instruction, and the CAR loaded with 0x808, the instruction decode micro-program. Next C20 is decoded and loaded to the CAR and the MOVE micro-program begins execution.

Figure 4.4 shows the final write back instruction performing a write-word bus cycle. The stack pointer, A0, is also shown.

Figure 4.5 shows the final contents of the first few locations of RAM after the program has completed. The correct results have been placed in their correct locations at this stage.

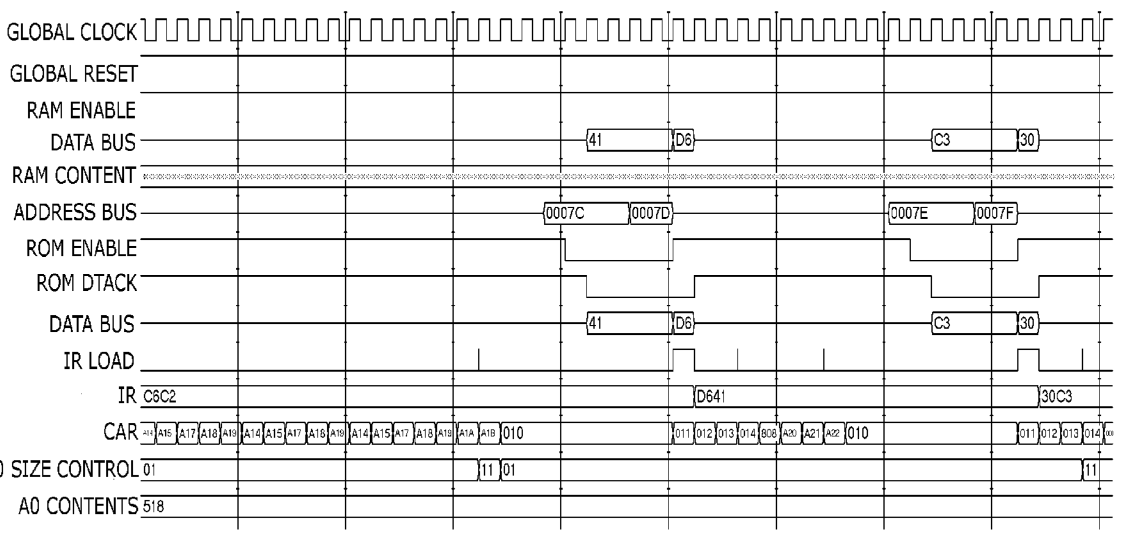


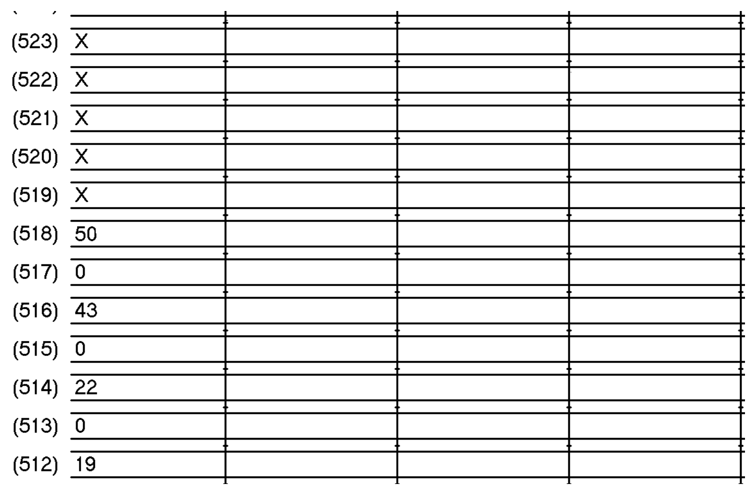
Figure 4.4: The final write back

4.2 Conclusions

The first motivation behind this project was to generate a Motorola 68008 CPU using VHDL. The 68008 package is in place ready for instantiation in any VHDL design. All necessary hardware is in place and while full micro-coding has not been completed enough instructions exist to verify the validity and functionality of this hardware.

A requirement of the design was to be capable of download to the FPGA project board. The design has been demonstrated as synthesizable and downloadable to the FPGA. As required a compatible bus interface has been successfully implemented in VHDL. The 68008 CPU model, as demonstrated, is capable of executing code as generated by the 68k cross assembler. While only a subset of the instruction set is implemented the instructions that have been demonstrate there is a solid foundation on which to add the remainder.

A goal of the work currently being put into the FPGA board design is to have two co-existing CPU architectures on the one FPGA. When considering downloadability of the design it is necessary to take into account the fact that when the two CPU designs are put side by side they take up more than the available resources for the Virtex-II FPGA in use. It is noted here that a re-arrangement of hardware in the 68008 design, perhaps omitting hardware duplicated for ease of operation e.g. the Displacement Register or the Vector Decoder could reduce the resources used by the 68008. It may be also possible to exploit common hardware features of both CPU's e.g. by having a common functional unit and only differing control paths. The results of both of these operations would remove replicated hardware across both CPU models and decrease the resources taken up by both CPUs thus allowing the designs to co-exist. Further reduction in the use of the most valuable resource, FPGA slices, may be accomplished by taking better advantage of the FPGA SelectRAM resources. The SelectRAM resources are dedicated on board RAM blocks



(523)	X			
(522)	X			
(521)	X			
(520)	X			
(519)	X			
(518)	50			
(517)	0			
(516)	43			
(515)	0			
(514)	22			
(513)	0			
(512)	19			

Figure 4.5: RAM contents after the program has finished

designed for the implementation of memory structures such as ROM. Modification of the ROMs used in the 68008 may free up some slice resources and further reduce the size of the design.

The second motivation behind this design was to create a readable, well documented design model with a view to teaching Computer Architecture students how such a design may be implemented in practical terms. In completing the project the design constraints outlined in the introduction have been adhered to and a modular, understandable design produced. Sufficient documentation exists for a student to understand and expand on the current design. Also, a basic software tool has been provided to ease the future micro-coding process.

To conclude, the main objectives of the project i.e. a functional, readable VHDL 68008-opcode compatible CISC implementation have been achieved to an extent allowable given the time constraints of the project.

4.2.1 Skills Acquired and Lessons Learned

The main skills acquired in implementing this project are proficiency in VHDL programming, knowledge of the HDL design flow including design specification and simulation for FPGA. Also a more detailed knowledge of processor design has been acquired together with a more in-depth insight into how a processor, such as the 68008, is implemented in real terms.

There have been several lessons taken from the completion of this project. Firstly, I have learned that time management skills are important when balancing a large project with course-based study. Secondly, when building a substantial project using an incremental prototype-functional-enhanced design technique is key to generating an all-round functional design.

4.2.2 Further Work

Opportunities exist for further work both in the completion of the current design and the expansion of the current design. The first goal has to be the completion of micro-coding. Some suggestions for such a project include.

- Re-design of the MicroWord Generator program to include generation of comments, reverse-encoding of a control word and full inclusion of branching locations.
- Design of an efficient memory map for the CROM
- Implementation of the full set of decoding.
- Efficient implementation of all complex instructions such as divide signed.

On the hardware side of the project further work suggested stems from making the implementation operable on the FPGA project board. Suggestions include

- Downloading of the current VHDL unit to the project board.
- Implement the supervisor/user programming model as defined by Motorola.
- Examine design efficiency and how it may be improved. For example, reduce FPGA slice resource use by using FPGA SelectRAM resources for CROM or an external dedicated ROM.
- Attempt to enable the co-existence of the LEON and the 68008 on the board through optimisation of both designs and removal of common hardware.
- Attempt to exploit the complex hardware already built into the LEON unit, for example using the LEON floating point unit to add floating point functionality to the 68008 model.
- Devise a more efficient, user friendly manner of programming the CROM i.e. other than the use of constants. For example, exploring SelectRAM generation and initial value specification using the CoreGEN synthesis tool.

Bibliography

[1] Antonakos, J. *The 68000 Microprocessor, Hardware and Software Principles and Applications*, forth ed. Prentice Hall, 1999.

[2] Brennan, R. The design and evaluation of an fpga based microprocessor project board. Tech. rep., Department of Computer Science, Trinity College Dublin, 2003.

[3] Brennan, R., and Manzke, M. On the introduction of reconfigurable hardware into computer education. *Workshop on Computer Architecture Education* (2003).

[4] fpgacpu.org. Fpga-cpu open source group. Online, http://www.fpgacpu.org.

[5] Gaisler, J. Leon/amba vhdl model description. Tech. rep., European Space Agency: Control, Data and Power Division, 2000.

[6] Glaurt, W. Vhdl tutorial. http://www.vhdl-online.de.

[7] Hennessy, J., and Patterson, D. *Computer Architecture, A Quantitative Approach*, third ed. Morgan Kaufmann, 2002.

[8] Mano, M. *Digital Design*, second ed. Prentice Hall, 1991.

[9] Mano, M., and Kime, C. *Logic and Computer Design Fundamentals*, second updated ed. Prentice Hall, 2001.

[10] Motorola Inc. *Motorola M68000 Family Programmers Reference Manual.* Available Online, http://www.motorola.com.

[11] Motorola Inc. *Motorola M68000 Users Manual*, ninth ed. Prentice Hall PTR, ISBN: 0-13-566695-3 / 0135666953.

[12] Sparc. *The SPARC V8 Architecture Manual*, v80 ed. Available online, http://www.sparc.com/standards/V8.pdf.

[13] Xilinx. Xilinx foundation tool product guide. Available online, http://www.xilinx.com.

[14] Yalamanchili, S. *Introductory VHDL, From Simulation to Synthesis*, first ed. Prentice Hall, 2001.

[15] Zargham, M. *Computer Architecture, Single and Parallel Systems*, first ed. Prentice Hall, 1998.

Appendix A

Control Word Field Guide

Field	Signal	Function
BI	000	Read Byte Cycle
BI	001	Read Word Cycle
BI	010	Read Long-Word Cycle
BI	011	Write Byte Cycle
BI	100	Write Word Cycle
BI	101	Write Long-Word Cycle
BI	110	Interrupt Acknowledge Cycle
BIE	1/0	Bus Interface Enable
FAULTIN	1/0	Bus Fault processing flag
MCA	000	Increment CAR
MCA	001	Load CAR from top of MicroStack
MCA	010	Load CAR from single branching register
MCA	011	Load CAR from instruction decoder
MSPI	00	Hold MicroStack pointer value
MSPI	01	Increment MicroStack
MSPI	10	Increment MicroStack
MSPI	11	Decrement MicroStack
SL	1/0	Load the Single Branching Register with CAR + 1
PL	00	Hold PC Value
PL	01	Increment PC
PL	10	Load PC
PL	11	Unused
DP	1/0	Load Displacement register
IL	1/0	Load Instruction register
REGISTER	00xxx	D0 D7
REGISTER	01xxx	A0 A7
REGISTER	10000	System Stack Pointer
REGISTER	10001	EA Address Register
REGISTER	10010	EA Data Register
REGISTER	10011	Memory Address Register
REGISTER	10100	Memory Data Register
REGISTER	10101 11101	Temporary Registers T0-T8
REGISTER	11110	Vector Address Register
REGISTER	11111	Zero Provider
MC	0000	No Condition Code Write

MC	0001	Set C,V,N,Z
MC	0010	Set X,C,V,N,Z
MC	0010	Allow Z to be set only
MC	0011	Allow N to be set only
MC	0100	Set the IPL bits from internal bus
MC	0101	Set Trace Bit
MC	0110	Clear Trace Bit
MC	0111	Set Supervisor Bit
MC	1000	Clear Supervisor Bit
MC	1111	Load SR
FUC	000	CWORD Destination / A Port Control
FUC	001	Instruction[11:9] as Data register for A Port / Dest.
FUC	010	Instruction[3:0] as Data Register for A Port / Dest.
FUC	011	Instruction[11:9] as Addr. Register for A Port / Dest.
FUC	100	Instruction[3:0] as Addr. Register for A Port / Dest.
FBC	000	CWORD B-Port ontrol
FBC	001	Instruction[11:9] as Data register for B Port
FBC	010	Instruction[3:0] as Data Register for B Port
FBC	011	Instruction[11:9] as Addr. Register for B Port
FBC	100	Instruction[3:0] as Addr. Register for B Port
MA	00	Register File A Select
MA	01	PC to Address Bus
MA	10	Vector Address from Decoder
MA	11	Register File B Select
MB	00	Register File B Select
MB	01	SR[IPL]
MB	10	System Register
MB	11	Displacement Register
TC	1/0	MicroCCR Load
Next Address	12-bits	Next CAR Address
Vector No	8-bits	Vector Number from CWORD
SQ	1/0	Load destination port with IPL control
AA	1/0	Allow next address to be specified
ACC	1/0	Allow condition codes to be specified
IE	1/0	Generate Internal Exception
CCOND	00000	B. Carry Clear (MicroCCR)
CCOND	00001	B. V Clear (MicroCCR)
CCOND	00010	B. Not Zero (MicroCCR)
CCOND	00011	B. Positive (MicroCCR)
CCOND	00100	B. Carry Set (MicroCCR)
CCOND	00101	B. V. Set (MicroCCR)
CCOND	00110	B. Zero (Micro CCR)
CCOND	00111	B. Negative (Micro CCR)
CCOND	01000	BCC
CCOND	01001	BVC
CCOND	01010	BNE
CCOND	01011	BPL
CCOND	01100	BCS
CCOND	01101	BVS

CCOND	01110	BEQ
CCOND	01111	BMI
CCOND	10011	Wait on DTACK
CCOND	10110	BHI
CCOND	10111	BLS
CCOND	11000	BGE
CCOND	11001	BLT
CCOND	11010	BGT
CCOND	11011	BLE
CCOND	11100	Next Address Load
SS	1/0	Set Size from Instruction / CWORD
SZ	00	Byte
SZ	01	Word
SZ	10	Long
DS	000	Decode Opcode
DS	001	Decode EA for Fetch
DS	010	Decode EA for Write Back
DS	011	Decode EA for Fetch (MOVE)
DS	100	Decode EA for Write Back (MOVE)
MD	1/0	Load External Data / Data From ALU
RW	1/0	Register Write / Read
BR	00	MicroStack Idle
BR	01	Push
BR	10	Push
BR	11	Pop
FS	00000	Transfer A
FS	00001	Increment A
FS	00010	A + B
FS	00011	A + B + 1
FS	00100	Negate B
FS	00101	Decrement A
FS	00110	A B (A + (/B + 1))
FS	00111	Transfer A
FS	0100x	A and B
FS	0101x	A or B
FS	0110x	A xor B
FS	0111x	Negate A
FS	1x000	Arithmetic Shift Right
FS	1x001	Arithmetic Shift Left
FS	1x010	Rotate Right
FS	1x011	Rotate Left
FS	1x100	Rotate Right with extend (ROXR)
FS	1x101	Rotate Left with extend (ROXL)
FS	1x110	Logical Shift Right
FS	1x111	Logical Shift Left

Appendix B

Design Hierarchy

For demonstration and simulation purposes a Virtual Board has been constructed consisting of an instantiation of the 68008 VHDL CPU, a ROM and a RAM. The ROM is memory mapped from memory locations 0-128 while the RAM is mapped from location 512-768.

The following outline shows the full hierarchy of the design from board level to register level. The accompanying CD-ROM contains a Xilinx 4.1 project in the X_6800-VHDL folder and a Xilinx 5.1 project in the 68000-VHDL folder.

```
\- 68008 [MC68000.VHD]
 \- Bus Interface [BusInterface.VHD]
  \-Register_3bit [Register_3.vhd]
 \- Control Path [Control.vhd]
  \- CROM4096x64 [CROM4096x64.vhd]
  \- Instruction Decoder [Instdecoder.vhd]
   \- ROM256x11 [BlockROM.vhd]
  \- MicroSequencer [Microsequencer.vhd]
   \- Register_12bit [S_Reg_12.vhd]
   \- Register_CAR (CAR) [SReg_12CAR.vhd]
   \- Register_12bit_NegEdge [S_Reg_12neg.vhd]
 \- Register_32bit_0 (PC) [R32.vhd]
 \- Register_32bit_0_NegEdge (PC + 1) [R32neg.vhd]
 \- Register_16bit (IR / DISPL) [Register_16.vhd]
 \- Register_3bit (IPL Pending) [Register_3.vhd]
  \- Vector Decoder [VectorDecoder.vhd]
 \- Data Path [Datapath.vhd]
  \- Functional Unit [Functional_Unit.vhd]
   \- 32Bit ALU [ALU32.vhd]
    \- 1 bit Full Adder [FullAdder.vhd]
   \- Shifter [Shifter.vhd]
 \- Register_4bit      [Register_5.vhd] (MicroCCR)
 \- Register_16bit [RegisteR_16.vhd] (SR)
 \- Register File [RegisterFile.vhd]
  \- Register_32bit_Motorola [Register32.vhd]
 \- Tri-State Buffer [tristatebus.vhd]
\- RAM [RAMC.vhd]
\- ROM [spblockrom.vhd]
```