EE 552 High Level Digital ASIC Design Using CAD

PROJECT

CRAM Parallel Processor

Final Project Report

November 27, 2001

Shahid Aslam Khan	Sue Ann Ung	Satneev Bhamra
Shahid1994@yahoo.com	sueann_u@yahoo.com	sbhamra@hotmail.com

DECLARATION OF ORIGINAL CONTENT

The design elements of this project and report are entirely the work of the authors and have not been submitted for credit in any other course except as follows:

- Fig5 is taken from reference [2], fig 1 and 3 are from reference [3]
- A few headings and statements are taken from reference [2] and [3] and are appropriately mentioned with the text as well.
- Fig 4 for the C-RAM is taken from reference [4], *A Tightly Coupled Hybrid SIMD/SISD System.*
- Fig B in the Appendix is obtained by simplifying figure 5 which is from reference [2], *Computational RAM: Implementing Processors in Memory*.
- Figure 6 is taken from references [2, 4].
- Debounce.vhd code is taken from EE 552 course Application notes http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999_w/keypad/debounce.vhd
- Display_7seg.vhd code is reused from EE 552 Lab 4 Part 3

Shahid Aslam Khan   Sue Ann Ung   Satneev Bhamra

ABSTRACT

The following report documents the project specifications to implement C.RAM on an FPGA. C.RAM (computational random access memory) is a fast parallel processor that uses a SIMD architecture. The C.RAM accepts single instruction stream into parallel processing units to perform multiple data operations. The design of the processor is to be implemented on a UP1 board and an Altera FLEX10k20 chip.

In this report a detailed description of the architectural components of C.RAM is provided. Details of the design and operation, requirements for the FPGA and signal and pin-out descriptions are all outlined. A general discussion of our testing method is found later in the document. Finally the complete design documentation of the entire project and completed and working VHDL code is referenced.

This document shows that we were partially successful in implementing C.RAM on an FGPA. We were able to see correct results for the addition instruction and some other simple instructions we had set out to build, however, we did not receive correct results for the multiplication of two four bit numbers. We were successful in combining additional processing elements into an array to have C.RAM work in true SIMD fashion.

C.RAM was designed to run with a 25MHz clock signal, and used 344/1152 (30%) of the AlteraFlex10k20 logic cells.

C-RAM DATA SHEET

C.RAM Features:

- Processing Element Array
  - 256 Instruction ALU : 8 x 1 MUX
  - 3 x 1-bit Registers
- Controller
  - Sequencer
  - Decoder
- Memory
  - 256 x 16 bit instruction ROM initialized by .mif file
  - 256 x 1 bit data RAM initialized by .mif file
- I/O
  - 7 segment display to show results of PE elements and memory
  - LCD display to show instruction set in the ROM.

- Designed to run at 25 MHz clock signal
- C.RAM* used 30 % of Altera Flex10K20 logic cells.

* Total logic cell value is for only one PE

Instruction Set:

The following table outlines the current instruction set used by C-RAM

Instruction	Syntax	Description	Result
Sum	A + B	Addition of two four-bit numbers	Correct
Subtraction	B – A	Subtraction of two four-bit numbers using 2's complement	Incorrect
Multiplication	B x A	Unsigned product of two four-bit numbers	Incorrect

Chip Information:

The C-RAM design takes up 344 logic cells in total. This is with implementing only one PE, if we increase the number of PEs in the array, the number of logic cells used will increase proportionally. Our estimates show that we can safely add up to 30 PEs into our design.

Module	Logic Cells required	Performance
PE (1)	12 (1 %)	113.63 MHz
PE (2)	36 (3 %)	113.63 MHz
Controller	78 (6%)	125.00 MHz
I/O	254 (22%)	22.83 MHz
Total*	344 (30 %)	22.17 MHz

PE*: only one PE

Memory Addressing and Output

C-RAM contains two separate memory blocks, one for storing instructions and one reading and writing data. Each instruction is 16 bits long, and is of two types: a memory address for the PE to receive data from and an operational code for the PE to perform. The MSB of the instruction determines between these two types of instructions. If the flag bit is = ‘1’ then the instruction is an OpCode. If the flag bit = ‘0’, the instruction code is designated as a memory address in the RAM. Data is pre-set into the RAM as one bit on each address line. Data is fed from the designated address in the RAM to the PE one bit at a time.

I/O Signals and Pins

The table below describes each the input and output pins used on the UP-1 board. There are two inputs from push buttons to start the C.RAM and the LCD. There are 3 inputs, one from the clock oscillator on the board, and two input clocks for the C.RAM and LCD. Two seven-segment displays are used for displaying the results in the X, Y and Memory. An LCD is used to display the instruction set stored in the ROM.

IO Signals	Input/Output	Pin Number	Description	Number of Pins
clock_25MHz	In	91	Clock signal provided by on-board oscillator (25.175 MHz)	1
clock_pb	In	28	Clock Signal Button	1
clock_lcd	In	29	clock signal for LCD (2 x clock_pb)	1
				3
display10	Out	6	for segment1 a (result in X or Y)	1
display11	Out	7	for segment2 b (result in X or Y)	1
display12	Out	8	for segment3 c (result in X or Y)	1
display13	Out	9	for segment4 d (result in X or Y)	1
display14	Out	11	for segment5 e (result in X or Y)	1
display15	Out	12	for segment6 f (result in X or Y)	1
display16	Out	13	for segment7 g (result in X or Y)	1
				7
display20	Out	17	for segment1 a (result in M)	1
display21	Out	18	for segment1 b (result in M)	1

IO Signals	**Input/Output**	**Pin Number**	**Description**	**Number of Pins**
display22	Out	19	for segment1 c (result in M)	1
display23	Out	20	for segment1 d (result in M)	1
display24	Out	21	for segment1 e (result in M)	1
display25	Out	23	for segment1 f (result in M)	1
display26	Out	24	for segment1 g (result in M)	1
				7
lcd_data0	Out	65	Output of LCD	1
lcd_data1	Out	66	Output of LCD	1
lcd_data2	Out	67	Output of LCD	1
lcd_data3	Out	68	Output of LCD	1
lcd_data4	Out	70	Output of LCD	1
lcd_data5	Out	71	Output of LCD	1
lcd_data6	Out	72	Output of LCD	1
lcd_data7	Out	73	Output of LCD	1
reset_lcd	Out	40	Reset the LCD screen	1
reg_select	Out	62	Selects register to be written to	1
lcd_rw	Out	63	Read/Write to LCD (set as write)	1
lcd_nenable	Out	64	Enable the LCD	1
				12
				Total = 29

TABLE OF CONTENTS

INTRODUCTION

Computational RAM is a processor-in-memory architecture that makes highly effective use of internal memory bandwidth [3]. The architecture as described in later sections consists of a control unit (sequencer) and many Processing Elements (PE). The host processor generates instructions, which are broadcasted to all Processing Elements. The data in the memory is fetched to the single bit registers (X, Y) and processed. The truth table is sent to the inputs of the 3 to 8 mux. The select lines for the mux are from the registers (X, Y) and one from memory. The 3 to 8 mux can support 256 possible instructions. This SIMD parallel processor architecture is implemented on an FPGA chip.

1.0 Motivation

There is significant interest in processor implementation in Field Programmable Gate Arrays (FPGAs). FPGAs are often used for prototyping new designs as it allows the specification of the function of a system to be simulated and tested before the ASIC is actually fabricated. This saves time and money, and it allows the possibility of quick redesigning and modifications of existing designs.

We are proposing to build a C-RAM controller-processing unit using SIMD architecture. The motivation behind this idea is to “exploit the chips’ wide internal data paths” and “the energy efficiencies that result from better utilization of memory bandwidth and localization of computations on a millimeter scale” [2]. SIMD architecture allows for fast parallel processing of data.

ACHIEVEMENTS

2.1 Top Level

The top level design of C.RAM functioned correctly for certain instruction sets. We found that correct results were obtained for the addition operation of two four-bit numbers. Other simple instructions were also functioning correctly in C.RAM. However, we were unsuccessful in implementing a multiplication algorithm on C.RAM. The output did not result in the correct multiplication of two-four bit numbers.

The top-level design was also successful in implementing an array of PEs. Experiments were done with different array sizes of PEs, and correct results were obtained for the addition operation.

2.2 The Controller

The controller was initially designed as a finite state machine. It contained four main states: idle, read_RAM, write_RAM and Operation. This implementation was found to be faulty. Although it correctly fetched the instructions from the ROM and sent the corresponding OpCode to the PE or the memory address to the RAM, timing issues arose. The synchronous controller was writing to and reading from the RAM in one complete cycle. Although this can be accomplished using an aynchronous design, for our synchrounous machine, this resulted in incorrect results. The controller would not read from the correct address, or instructions, which were labelled as OpCodes were interpreted as addresses.

The controller was redesigned as a simple selector. The finite state machine was abandoned, and the new selector design was implemented. Instructions from the ROM are fetched, and the corresponding signals are set. The instruction format is the same; if the MSB =1 the instruction is deemed an OpCode, if the MSB = 0 the instruction is deemed an address. One further condition is set in the instruction format, if the second MSB is set as a 1, the controller sets this address_write signal, else, the address_read signal is set high.

Currently, the controller is functioning correctly for all test cases. Simulations are as expected.

2.3 PE

The PE design is functioning correctly. Simulation results indicate that for all test cases presented, the PE is working correctly. The PE accepts the OpCode from the controller, and the corresponding enabling registers. These have been sent as inputs to the PE by the test bench. The PE then performs the correct operations set by the OpCode and stores the results in the x, y and memory registers. Note: for the PE component itself, a RAM interface was not used, rather a simple register was used to act as RAM. Data was written and read from this register.

2.4 User Interface

The two 7-segment displays connected to the FLEX10K chip is used to show the contents of the X and Y 1-bit register of the PE. The OpCode going into the PE and the address and enable signals are shown on the LCD display. The two push buttons connected to the FLEX10K are used for manual clocking and reset signals. The push button signals are both connected to debounce circuits. The user interface works as expected.

DESIGN OVERVIEW

Design Hierarchy

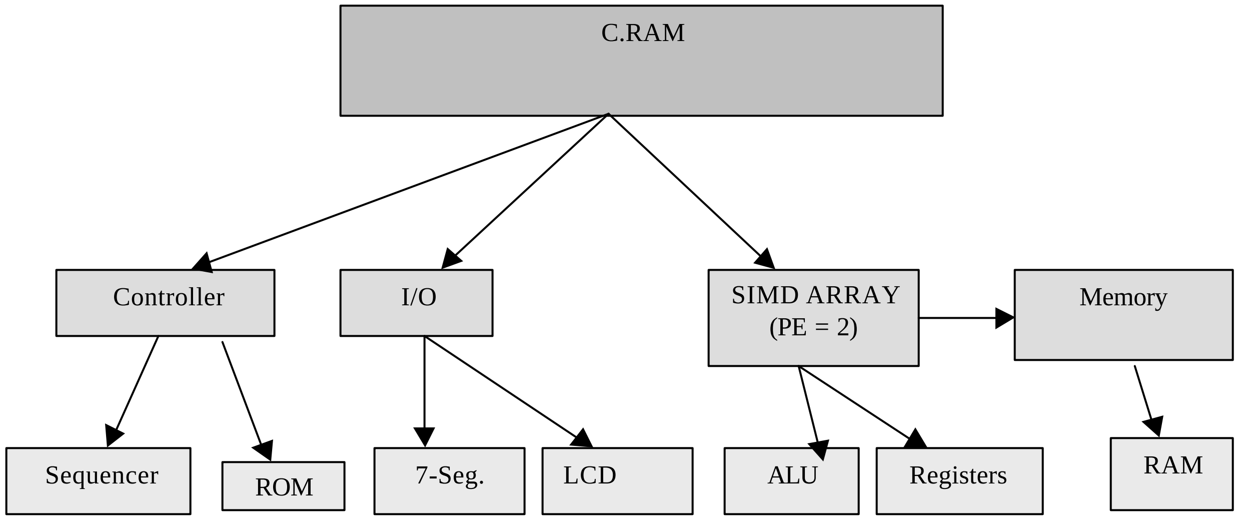


Figure 1: Design Hierarchy

3.0 Description of Operation

The C-RAM is a conventional memory chip that when used with a SIMD computer, can be utilized to run parallel processes in applications such as signal and image processing, computer graphics, databases and CAD [2].

The C-RAM processing module is broken down into four main components: the control unit (CU), processing elements (PE), memory modules (C-RAMs) and I/O interfaces and modules such as video displays and keyboards. The CU contains instructions either locally or fetches then from the host processor and broadcasts each instruction to the PEs. These instructions may include operational codes such as addition or subtraction, and broadcast information. Each PE acts as an independent arithmetic unit (AU) and performs operations on the local memory.

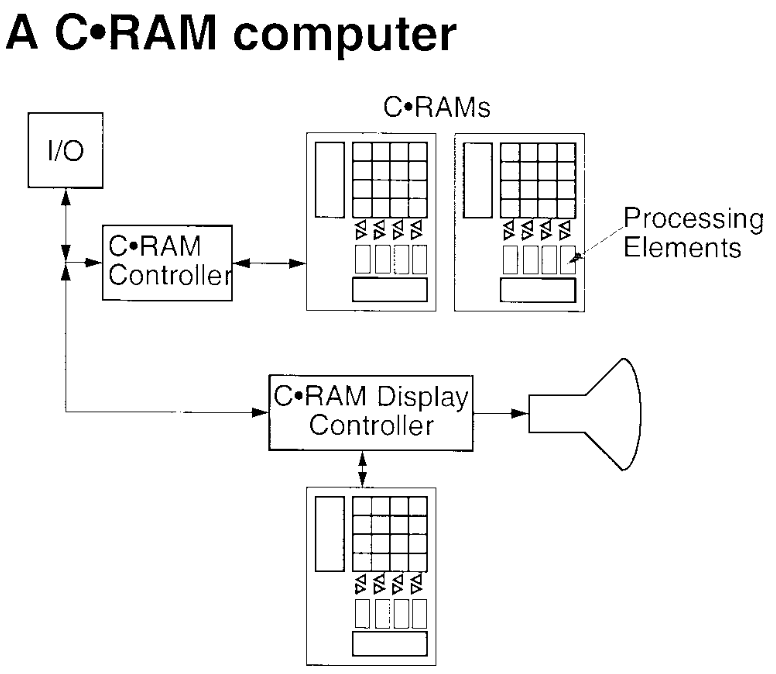


Figure 2: A basic SIMD-CRAM processing module [3]

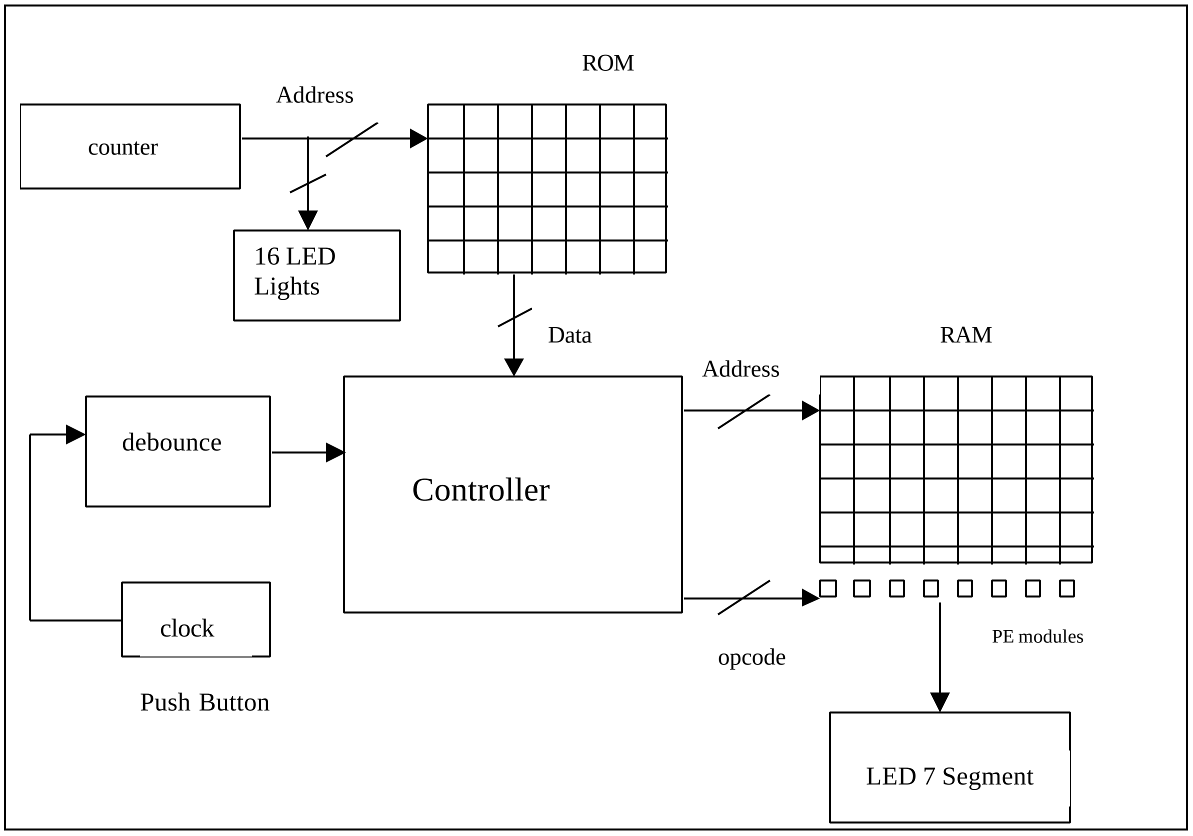


Figure 3: Block Diagram of the C-RAM Parallel Processor System

The instructions and address information are first loaded into the ROM before the start of operation.

The controller will then sequence through the ROM to get instructions and address information for the PE operations. The PE operations and the contents of the X and Y registers and the RAM can be seen on the seven- segment display. Instructions issued by the controller are displayed on the LCD. The clock is input into the system from the push button.

DESIGN DETAILS

4.1 The Control Unit (CU)

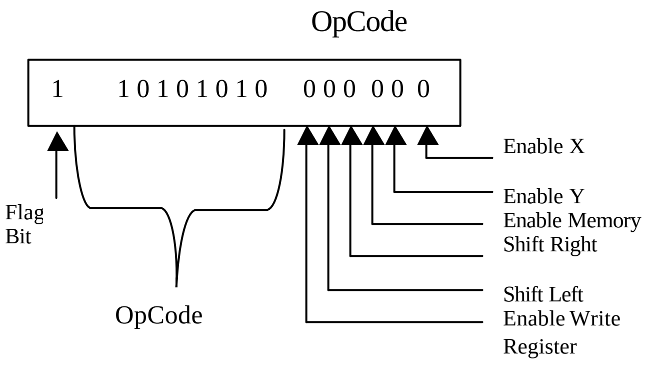
The CU contains two main components: the sequencer and the controller. The sequencer obtains instructions stored in the local storage of the CU and passes them onto the controller. The controller then determines what addresses in the local memory each PE should read from and write to, the related operational codes (OpCode) and broadcast information the PE must perform.

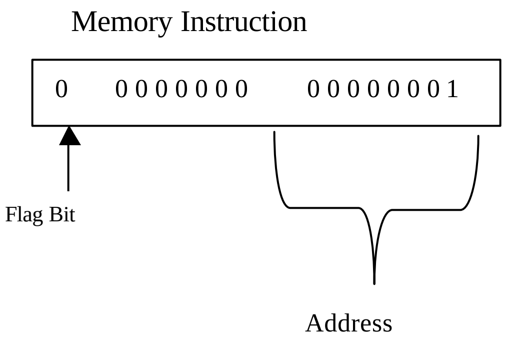
4.1.1 The Sequencer

The C.RAM sequencer provides the instruction stream to each processing element in the C.RAM. These instruction words can take the form of an OpCode or an address in memory.

Initially a minimalist sequencer will be designed and tested within the C.RAM. This sequencer acts a simple program counter to provide instructions to the PE's and requires no ALU or data-path. An instruction storage element can be added to this to store multiple routines for the sequencer. This sequencer style is optimal for many embedded systems such as filtering continuous streams of data in signal processing applications.

The sequencer program counter selects the instruction in the ROM to be fetched. Each instruction is 16 bits long. There are two types of instructions: a memory address where the PE will perform its operation on and the operational code itself. A flag bit is set to differentiate between these two types of instructions. The MSB of the instruction indicates the flag bit; if it is equal to '1' then the instruction is an OpCode. If the flag bit is equal to '0' then the instruction is a memory address in the RAM. Six bits are reserved as register enable bits, for three registers (X, Y and Write-Register), memory write-enable bit, and shift left and shift right bits. Shift left and shift right bits allow for communication between two PEs. This instruction is then passed onto the controller. The following is an example of the format for each instruction:





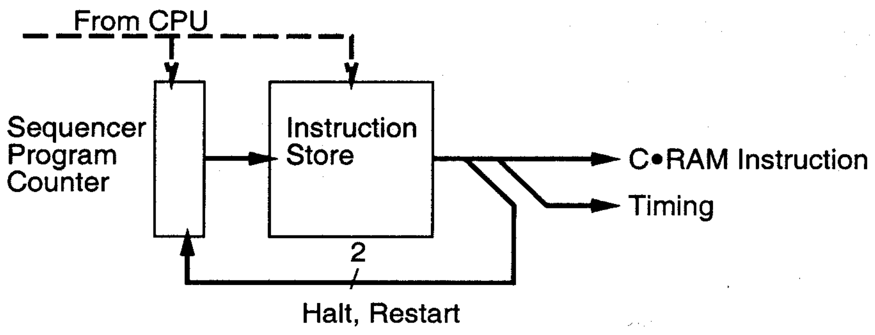


Figure 4: Minimalist microcoded sequencer [3]

4.1.2 The Controller

The Controller is based upon a simple decoder implemented in the code by using a case statement. There are three possible cases based on the two most significant bits of the instruction fetched by the sequencer. The controller either selects the address to read data from the RAM (MSBs are 00) or it selects an address to write data back to RAM (if MSBs are 01) or issues an OpCode and other signals to PE (if MSBs of instruction are none of the above two cases).
So its either the PE if the instruction is an OpCode or to the local RAM if the instruction is a memory address.

The controller interprets the instruction as an address or an OpCode and sets the correct signals corresponding to the instruction. If the flag bit is equal to 1 the controller moves into the operation state, sets the OpCode for the PE and enables the correct registers. If the flag bit is equal to 0, the controller moves to the read state, and sets the address bit to the corresponding address in the instruction and outputs it to the RAM.

In the reset signal is set high, the controller sets all read and write enable flags to zero and hence data is also set to zero. If the flag = ‘0’ then the controller moves to the read state, and issues the instruction to the local RAM. The eight LSBs are sent to the RAM and the rest of the bits are masked off. If the flag = ‘1’ then the controller moves to the operation state. Here, the MSB is masked off, and the next eight bits are sent as the OpCode to the PE. The last six bits of the instruction contain the enabling bits for each of the registers X, Y, and Write-Register. It also contains the memory write enable bit and shift left and shift left bits for communication neighboring PEs.

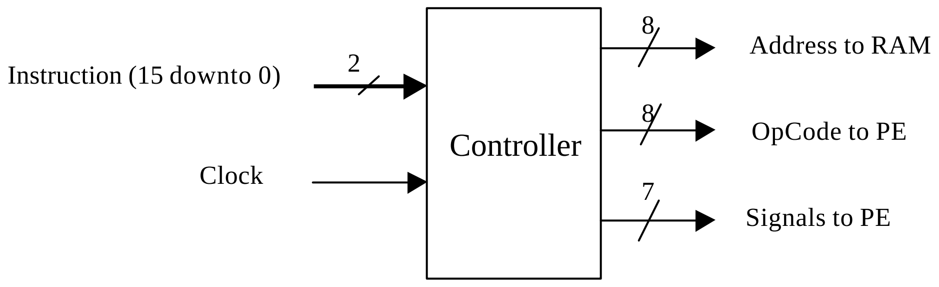


Figure 5: Controller

We will require two different types of memory blocks, ROM and RAM. The memory blocks, which are already present on the Flex10k20 will be sufficient for our initial design of C.RAM. The ROM memory block will contain the operational instructions and memory instructions. These will be fetched by the sequencer and sent to the controller.

RAM memory blocks will be used for reading and writing the data to and from by the PE. Eight bits comprise each address of the memory block, thus a maximum of 256 lines can be used on the Flex10K20.

4.1.3 Memory

We will require two different types of memory blocks, ROM and RAM. The memory blocks, which are already present on the Flex10k20 will be sufficient for our initial design of C.RAM. The ROM memory block will contain the operational instructions and memory instructions. These will be fetched by the sequencer and sent to the controller.

RAM memory blocks will be used for reading and writing the data to and from by the PE. Eight bits comprise each address of the memory block, thus a maximum of 256 lines can be used on the Flex10K20.

4.2 C-RAM Architecture

The C-RAM structure will be implemented using the SRAM memory cells on the Altera board. The sense amplifiers at the bottom of each column may not be necessary if we use SRAM cells instead of the traditional DRAM cells that the C-RAM was implemented on.

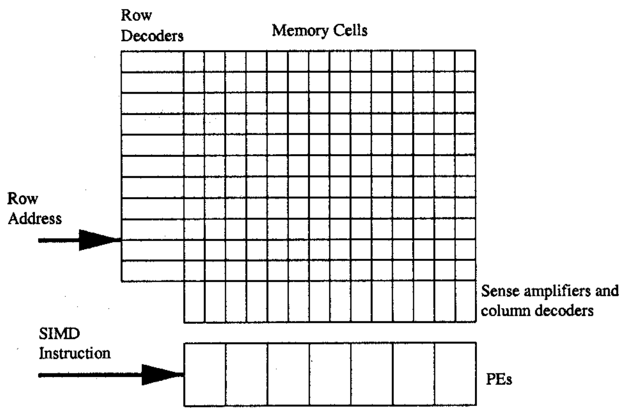


Figure 6: Simplified C-RAM Architecture [4]

Figure 6 shows a simplified C-RAM configuration with the memory cells, sense amplifiers, row and column decoders and processing elements (PEs). SIMD instructions come from a sequencer in the main controller. The instructions are then decoded and routed by the individual PEs according to the common SIMD instructions.

4.3 Processing Element (PE)

Figure 7 shows one PE structure that we can use for the C-RAM. As mentioned before the sense amps may not be necessary when the C-RAM is implemented using SRAM memory cells.

The ALU is a 256 function ALU that is based on an 8-by-1 MUX. The op-code coming from the main controller in the global instruction bus is an 8-bit value that operates on three inputs commonly from the X, Y registers and the PEs data store. The ALU operation is received as a truth table in the op-code to the data inputs of the MUX. The ALU supports bit-serial computation and has left right and wired-AND bussed communication for inter-element (adjacent PEs) operations such as left-right shifting. During the communication between adjacent PEs, the ALU is used to route signals. The output of the ALU goes to the X, Y registers, memory and write-enable register. The controller detects an operation-complete signal from the PEs and sends out the next OpCode or instruction for the next operation (see Figure 7) [2, 4].

The write-enable register is used for conditional operation implementations as a handshaking between the main PE module and its memory store [4].

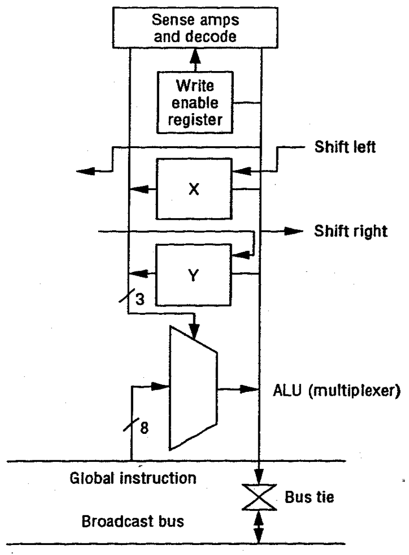


Figure 7: Diagram for PE operation with 256 instructions [2]

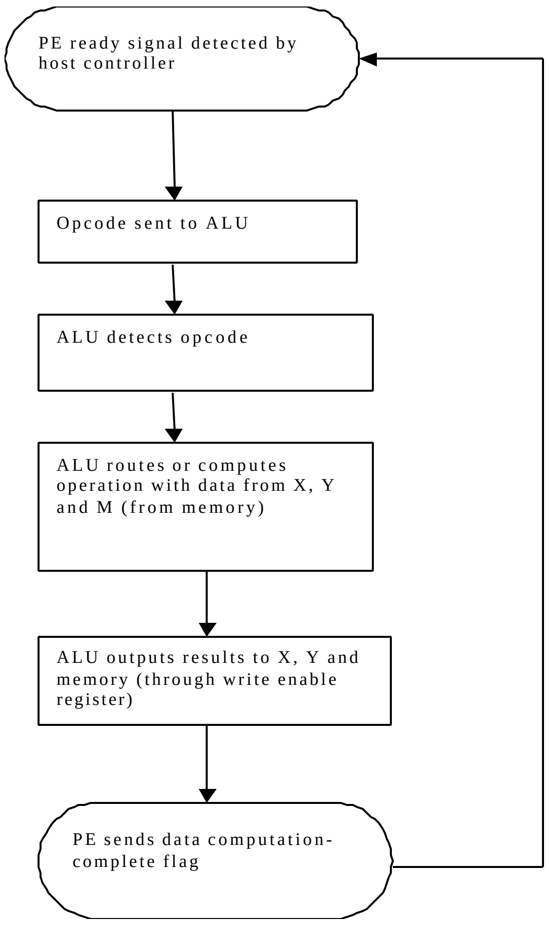

PE ready signal detected by host controller
Opcode sent to ALU
ALU detects opcode
ALU routes or computes operation with data from X, Y and M (from memory)
ALU outputs results to X, Y and memory (through write enable register)
PE sends data computation-complete flag

Figure 8: Flow Diagram for PE operation [2, 4].

4.4 User Interface

Push Button with Debounce
To make the input and output signals slower for the purpose of demonstration of operation, the input clock is controlled from a Push Button. The push button is debounced before being used as input clock to the system.

LED 7 Segment Display
The four 7 segment displays on the board are used to indicate the values that are in the X and Y registers and the ram at a specific clock cycle.

LCD Display

The Optrex DMC 50218 LCD display is used with the two 7-segment LED displays to indicate the type of instructions (OpCode or address) that are being sent to the PEs. As an instruction is read from the ROM and processed, the instruction is also displayed on the LCD. The VHDL code for the LCD display is based on the code from the CDMA group. The code is modified to display the instructions for the C-RAM processor.

The LCD works on the 25 MHz system clock but the push button increments the different screens to be displayed. The progress signal is connected to the push button.

To initialize the LCD display, the following steps were used.

RS	**R/W**	**Data**	**Time Held**	**Purpose**
0	0	00000000	7 cycles (2.8ms)	Allow LCD to warm up
0	0	00110000	15 cycles (6ms)	Set display to 1 line, 8 bit data interface
0	0	00001000	1 cycle (0.4 ms)	Turn display off
0	0	00000001	4 cycles (1.6 ms)	Clear display
0	0	00000110	1 cycle (0.4 ms)	Set display to increment on write, no shift
0	0	00001111	4 cycles (1.6 ms)	Move cursor to home position

EXPERIMENTS

During the course of the project we experimented a few things, which proved to be very important in determining the exact direction of progress.

5.1 Initially we didn't try to use mif files for the instructions and thought about feeding them through keypad but it didn't turn out to be a good idea. After meeting with Dr. Elliott we decided to use mif file to store instructions and ream them from ROM. We also stored the data to be processed in the mif files.

5.2 Another problem we had was with the controller and it proved to be the hardest part to code. Initially we experimented with a state machine, which would go though three states (Idle, read, operation). This idea didn't prove to be too good as it caused timing problems. All components worked correctly separately but when connected together in the top-level file caused problems. We tried using four states (Idle, read, write, operation) but this idea didn't work too well. Finally we used case statements to change states and worked.

5.3 We initially used only one PE for the C-Ram computer and later we used two PE's and two separate RAMs one for each PE. By our calculations we find that we have :

1152 - 344 (total cells with one PE) = 808.

5.2.1.1 / 24 (size of logic cells for one PE) = 33.6

Thus we can safely add up to 30 PEs to our PE array.

5.4 Algorithms:

5.4.1 The multiplication algorithms did not work properly. We tried taking a four bit number, and with the use of a counter, looped it to calculate addition N times.

5.4.2 The subtraction algorithm is also not providing the correct results. Our results show that the 2's complement bits that are read back from the RAM are inverted. This provides an incorrect result.

5.5 Another important feature in this project was working with two clocks, one for reading the instructions from the mif file in ROM. Initially the clock to read from ROM was set to a frequency three times less than system clock but it proved not to be a very good idea. Then we used a clock period of 20ns for the system clock and 40ns for the ROM clock and it worked. The timing problem was the biggest problem we faced and it was hard to solve. The time period for ROM clock was set to high value because for one instruction system clock had to change a couple of states and if clocks with same frequency were used or if only one clock was used it didn't give the correct sequence of operations to get the desired result.

5.6 The instructions mif file was also experimented a lot since as all the operations (arithmetic operations, enable and disable signals) are actually controlled by the instructions form the mif file. The instructions decide the read and write addresses and also the op-code for the PE. We experimented a lot before we finally came with a correct sequence of instructions to carry out the things satisfactorily. The problem was enabling and disabling the registers and memory at correct time.

SCHEMATICS

6.1 Push Buttons

The MAX_PB1 and MAX_PB2 push buttons are used as input clocks to the FPGA. They provide active-low signals and are pulled-up through 10K ohm resistors. Connections to these signals are made by inserting one end of the hook-up wire into the push button female header. The other end of the hook-up wire should be inserted into appropriate female header assigned to the I/O pin of the FPGA device [1].

6.2 **LCDs**

An Optrex DMC 50218 LCD module is used to display the opcodes sent the PE and addresses for read and write.

6.3 **7-Segment Displays**

The following is a schematic connecting the FPGA on the Altera board to three seven segment display. In our project, we will be using three 7 segment display.

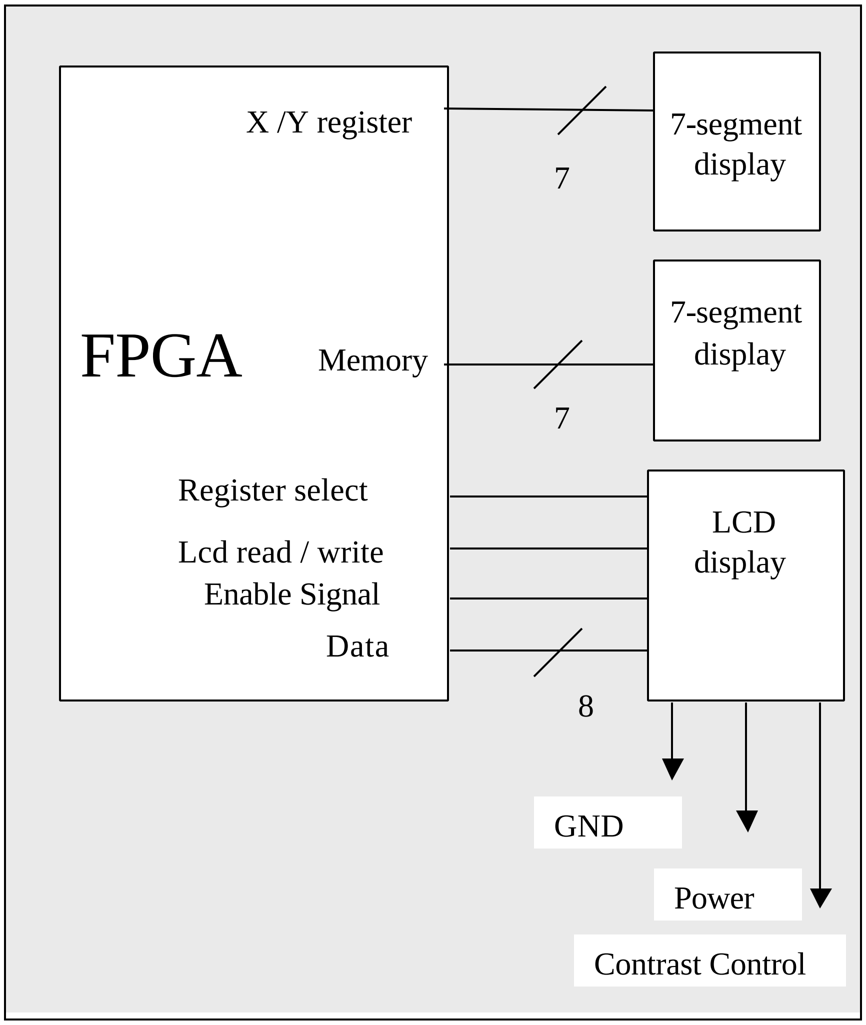


Figure 9: Hardware Schematic of two seven segment and LCD displays to the FPGA

TEST BENCHES

7.1 Controller Test Bench

A simple test bench was created to test the controller. (See Page 33) Inputs of clock, reset and an instruction signals were inputted into the controller and the outputs of OpCode, En_X, En_Y, En_memory, address_rd, address_wr and the states of the controller are verified.

Initially, an address instruction is sent to the controller, from which it is set to read the data. The controller recognizes the instruction as an address to read from, and enables the address_rd signal for the read cycle. Next, an OpCode instruction is sent to the controller. The controller recognizes this instruction as an OpCode, by the MSB of the instruction acting as a flag. The OpCode is sent to the PE and the corresponding registers are enabled in the PE and RAM. Finally, an address instruction to write too, is inputted into the controller. The controller recognizes this instruction as an address to write to by the second MSB of the instruction acting as a read/write flag. The controller sets the address_wr signal to high.

7.2 PE Test Bench

The PE test bench tests the C.RAM PE. (See Page 36). Inputs of clock, enable_x, enable_y, m and OpCode are fed into the test bench and the corresponding outputs of the PE are verified. The PE receives the OpCode and enabling signals, and then performs the computation. Results are stored into registers which are enabled and/or the RAM memory. Note: For the test bench the RAM interface is not being used, rather a simple register is used to act as RAM. Writing and reading to RAM is verified through this register.

7.3 PE Top_Level

The Top_Level test bench, simply inputs a clock and reset signals into the design. (See Page 38). The top-level CRAM component uses instructions stored in the ROM and data stored in the RAM to perform its computations. All the major components: controller and PE are self -contained in the top-level design, and require no external inputs. The only input the system requires is the clock.

INDEX OF SIMULATION WAVEFORM

8.1 Controller

The controller was thoroughly tested with a number of test cases. The controller functioned correctly, as it passed from its initial state of idle to either the operation state or the read state depending on the instruction received. On the Controller Waveform 1 in the Appendix an instruction containing an OpCode and enable bits was sent as an input. The controller received the instruction and moved into the operation state by correctly identifying the instruction as an OpCode from the flag bit (which is set to 1). The controller then set the OpCode signal to the correct OpCode, in this case OpCode = 10101010 and set the correct enable bits, En_X and En_Memory.

In Controller Waveform 2 we have sent varied the instructions from those, which are OpCodes to those, which contain RAM addresses. Again the controller received the instruction in the idle state and then moved into the next state corresponding to the instruction. From 0 to 500ns, and instruction = 1101010100001010 was sent into the controller. The controller again correctly identified this as an OpCode from the flag bit, set the OpCode = 10101010 and initiated the En_X and En_Memory signals to high. The controller then returned back into its starting state of idle to receive the next instruction. Between 500 ns and 1.0 us an instruction = 0000000000000001 is inputted into the controller. The controller identifies this as an RAM address from the flag bit and sets the address signal = 00000001. (As mentioned earlier, only the 8 least significant bits are used as the address, the rest are masked off). Again the controller, then returns to the idle state to await the next instruction.

We can see that the controller is functioning correctly. It interprets the instruction as either an OpCode or an address and sets the correct signals corresponding to the instruction. If the flag bit is equal to 1 the controller moves into the operation state, sets the OpCode for the PE and enables the correct registers. If the flag bit is equal to 0, the controller moves to the read state, and sets the address bit to the corresponding address in the instruction and outputs it to the RAM.

8.2 Controller & ROM

The next phase of testing was to test the controller with the ROM. A mif file was created which contained instructions for the controller and was stored in the ROM. A simple counter was used to access the ROM and pass on the instructions to the controller. To test the controller working with the ROM, we simply verified, the outputs of the controller with the instructions in the mif file.

In Controller + ROM Waveform in the Appendix. The signal "junk" is equal to the instructions in the mif file and is inputted into the controller. By verifying which instruction is sent against what signals are outputted, we can determine if the controller and ROM components are functioning correctly. The first instruction in the mif file is an address of 0000000000000000. We can see this on signal junk. The controller moves from the idle state to the read state and interprets the instruction from the mif file as an RAM address, thus sets the address line = 00000001. The controller then moves back to the initial state of idle. The next instruction in the mif file is an OpCode, junk = 1101010100000010. The controller interprets this instruction as an OpCode, and moves into the operation state. It then sets the OpCode = 10101010, and sets the En_X = 1, as determined by the instruction. The controller returns to the idle state. From 200ns to 275ns, an address instruction = 0000000000000101 is sent to the controller. (This is the third instruction in the mif file). Again we can see the controller moves into the correct state of read and sets the address signal to 0000101. Finally, the next instruction in the mif file is an OpCode, and from 275ns to 350ns, instruction = 1101000000000100 (seen as the junk signal) is sent to the controller. The controller sets the OpCode = 10100000 and sets the En_Y signal = 1.

By verifying the correct outputs of the controller against the instructions in the mif file stored in the ROM we see that the controller is correctly receiving the instructions from the ROM and outputting the correct results to either the PE or the RAM.

8.3 Test cases for the PE

The PE takes in the OpCode from the controller along with some other signals to enable the registers (x, y and write enable register) RAM, buses. The controller also sends in the shift right and shift left signals to the PE. We used internal signals port mapped to outputs to check the contents of the registers and memory.
We haven't done the RAM interface for writing the data back in to RAM so we are using a register as RAM and writing data back to that register.
The Enable signals are set manually in this PE simulation but for the top level File "ram_cont_ram.vhd" all the control signals are generated by the controller.
In the waveform attached in the Appendix we simulated following test cases.

OpCode	**Operation**	**Other signals**	**Expected Result**	**Result Obtained**
10101010	load x or y from data stored in mif file	Enable x is set to high	load the data at first memory location into x	x = 1 as data in RAM is 1
11110000	writing data back to RAM from x register	write enable is set to 1	RAM should go to logic 1 as x is 1	RAM=1
10101010	load x or y form the data stored in mif file	Enable y is set to high	Load data from next memory location into y	y = 1 as next data is 1.
10101010	load x or y form mif file	enable y is set to high	Load data from next location in memory into y	y =0 as next data is 0
10100000	x = x and m performs the and operation	x is enabled to write the result of and operation in y register	Data in register x is 1 and data in next memory location is 0 so and operation gives a 0	x register goes low i.e. 0
01100110	y= x xor m performs an x-or operation	y is enabled	Data in register x is 1 and data in next memory location is 0 so xor operation gives 1	y register goes high

Since we changed the controller from State machine to a selector, the following test cases are simulated to demonstrate that the PE now works better.

CASE1: (M =! Y)
Reads data from a mif file (ram.mif) in the RAM.
The data read from the RAM is inverted and written to register y.
The data in register y is written back to RAM.

The mif files and the simulations for the CASE1 are attached. The mif files are labeled according to the data they have. The simulations for the top-level test cases are so arranged that 1st page shows the whole simulation to see if the result is correct or not. Then in next few pages the simulation is spread over a few pages to see the instructions and op-code and addresses changing clearly.

So for CASE1 data in mif file is read as 0,0,11 and inverted and written to y as 1,1,0,0 and same is written back to RAM as 1,1,0,0.

CASE2: (ADDITION)

Data is read from the RAM (B[0]) and sent to register X.
Data is read from the RAM (A[0]) and its ANDED with data already in X and sent to Y.
Data in X is XOR with data in RAM (A[0]) and written into and X as well as into RAM.
RAM is disabled for Writing.

These steps are repeated three more times with one additional step and we get the final sum of two four bit numbers.

The Data in the mif file(ram_sum.mif) is B= 1001 and A = 1001
The sum being one bit at a time and written back to RAM should be Sum =0010
We get 0010, which is the correct result therefore C-RAM is working properly.
Since we changed the controller from State machine to a selector it seems to work better.

8.4 User Interface (7 Segment LEDs and LCD)

Clock Divider:
The clock divider takes in the 25MHz system clock as input and outputs the required clock speed needed for the circuits. In our system, only the debounce circuit needs to work on a 100Hz clock.

Debounce on the Push Buttons: The output of the debounce changes only when the push button is held for longer than the debounce period (1us).

7 segment LEDs: The LEDs are simulated using the code from Lab 3 where a counter is used to display values onto the LEDs.

LCD
The LCD module takes in the 25MHz system clock and run it through a delay to get the right timing for the LCD to initialize and to display characters from the a mif file.

Index to the Simulation Waveforms	**Page number**
Controller	63
Controller & ROM Simulation	64
C-RAM Simulation (PE = 1) M = !Y	66
C-RAM Simulation (PE = 1) Addition	67
C-RAM Simulation (PE = 2) Addition	68

Index to the Code

Design File	Description	Compile	Function Correctly	Comments	Page number
Display_7seg .vhd	Decoder to display correct outputs on the 7-segment display	Yes	Yes		58
Debounce	Circuit used to debounce the pushbutton input	Yes	Yes		46
LCD	LCD display	Yes	Yes		50
Pe_dp_loop.v hd	This file contains the architecture for the PE	Yes	Yes	Takes the OpCode and the enable control signals as inputs, and outputs the mux result to the corresponding registers or memory	51
Rom_control. vhd	This file contains the architecture for the controller and the ROM	Yes	Yes	Instructions are stored in a mif file located in the ROM and are fed sequentially to the controller. The controller interprets these insructions as either an address or and OpCode.	58
top_level.vhd	This is the top-level file connecting the Rom + Controller + PE + RAM	Yes	Yes	This is the top level file containing all the modules of C.RAM. Data is being read from RAM, operated on by the OpCode, and being written back into RAM.	37

REFERENCES

[1] Altera Corporation (1997). *UP1 Board Documentation*

[2] Elliott, Duncan G., Stumm, Michael, Snelgrove, W. Martin, Cojocaru, Christian, McKenzie, Robert (1999). *Computational RAM: Implementing Processors in Memory*. (p32 – 41) IEEE Design and Test of Computers January-March 1999

[3] Elliott, Duncan G. (1998). *Computational RAM: A Memory – SIMD Hybrid* PhD Thesis, The University of Toronto.

[4] Aklilu, Noah, Elliott, Duncan G., Wickman, Curtis A. *A Tightly Coupled Hybrid SIMD/SISD System.*
MSc Thesis, The University of Alberta.

[5] Luker, Jarrod. *RISC System Implementation in a FPGA* [Online]. Available: vailablehttp://cegt201.bradley.edu/~rekul/msee_project/documents/risc_fpga_pr oposal.pdf

[6] Bensler, Tim, Chan, Eric (1999). *Data Compression Co-processor*. Available :http://www.ee.ualberta.ca/~elliott/ee552/projects/1999_w/dataCompression/ [1991, March 29].

[7] Rivest, Micheal, Lawson, Kelly and Eriksen, Charlene (1999). *uCMK Microprocessor.*Available:http://www.ee.ualberta.ca/~elliott/ee552/projects/1999_w/microprocessor/report.html [1999, March 30].

[8] Benbow, Wendy, Behm, Rob and Joly, Craig (2001). *LLAMA Loadable Logic Arcade Machine Architecture*. Avaliable:
http://www.ee.ualberta.ca/~elliott/ee552/projects/2001_w/llama/ (2001, April 2)

[9] Cheung, Eric, Cheng, Felicia, Li, David and Kwan, Tim. (2000). SRAM Interfacing Basics [Online].
Available:
http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2000_w/interfacing/sram_basics/ Sram.html [2001, February 10].

APPENDIX

VHDL Codes

```
-------------------------------------------------------------------
-- Filename: tester.vhd
-- Descripton: Testbench for controller
-- By: CRAM Group (referenced from EE 552 Class Notes)
-- Date: November 12, 2001
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity tester is

end tester;


architecture test of tester is

-- send instructions to controller and verify results

component control2
       generic(
              total : positive := 15;
             size : positive := 7);

       port (
                clock, reset : in std_logic;
                instruction : in std_logic_vector (total downto 0);
                address_rd,address_wr : out std_logic_vector (size downto 0);
                opcode : out std_logic_vector (size downto 0);
                En_X, En_Y, En_memory : out std_logic;
          select1, dout1 : out std_logic_vector(1 downto 0);
                En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out std_logic
              );
end component;


signal clock, reset : std_logic;
signal instruction : std_logic_vector(total downto 0);

begin

control_check : control2
                           port map(
                           clock => clock,
                           reset => reset,
                           instruction => instruction
                           );

clock_period : process
begin
       clock <= '0';
       wait for 40 ns;
       clock <= '1';
       wait for 40 ns;
end process clock_period;

instruction_test : process

begin
    -- instructions for addition of two 4 bit numbers.
```

```
       wait for 120 ns;
       instruction <= "0000000000000000"; -- test for address
       wait for 120 ns;
       instruction <= "1101010100000010"; -- test for opcode
       wait for 120 ns;
       instruction <= "0000000000000100"; --test for address
       wait for 120 ns;
       instruction <= "1101000000000100"; -- test for opcode
    wait for 120 ns;
       instruction <= "1010110100001010"; -- test for opcode
       wait for 120 ns;
       instruction <= "1000000000000000"; -- test for opcode
       wait for 120 ns;

       instruction <= "0000000000000001"; --test for address
       wait for 120 ns;
       instruction <= "1011001100000010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1100010000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "00000000000000101"; -- test for address
       wait for 120 ns;
    instruction <= "1111011000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "1010110100001010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1000000000000000"; --test for opcode
       wait for 120 ns;

       instruction <= "0000000000000010"; --test for address
       wait for 120 ns;
       instruction <= "1011001100000010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1100010000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "0000000000000110"; -- test for address
       wait for 120 ns;
    instruction <= "1111011000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "1010110100001010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1000000000000000"; --test for opcode
       wait for 120 ns;

    instruction <= "0000000000000011"; --test for address
       wait for 120 ns;
       instruction <= "1011001100000010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1100010000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "0000000000000111"; -- test for address
       wait for 120 ns;
    instruction <= "1111011000000100"; --test for opcode
       wait for 120 ns;
       instruction <= "1010110100001010"; -- test for opcode
       wait for 120 ns;
    instruction <= "1000000000000000"; --test for opcode
       wait for 120 ns;


end process instruction_test;
end test;
```

```
------------------------------------------------------------------
-- Filename: tester2.vhd
-- Descripton: Testbench for PE
-- By: CRAM Group (referenced from EE 552 Class Notes)
-- Date: November 12, 2001
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity tester2 is
end tester2;


architecture test2 of tester2 is

-- send instructions to controller and verify results

component pe_dp_loop
       port (clock,enable_x,enable_y, write_enable: in std_logic;
          bus_enable,shift_right,shift_left : in std_logic;
          opcode : in std_logic_vector(7 downto 0);

          m_test : out std_logic_vector(0 downto 0);
          m: in std_logic_vector(0 downto 0);

          x_int,y_int,mux_d :  out std_logic_vector (0 downto 0)
          --data_written_back_toRAM
             );

end component;


signal clock, enable_x, enable_y, bus_enable, write_enable : std_logic;
signal shift_right, shift_left, m : std_logic;
signal opcode : std_logic_vector(7 downto 0);

begin

pe_check : pe_dp_loop
                           port map(
                           clock <= clock,
                           enable_x <= enable_x,
                           enable_y <= enable_y,
                           bus_enable <= bus_enable,
                           shift_right <= shift_right,
                           shift_left <= shift_left,
                           m <= m,
                           opcode <= opcode
                           );


clock_period : process
begin
       clock <= '0';
       wait for 40 ns;
       clock <= '1';
       wait for 40 ns;
end process clock_period;

opcode_test : process
```

```
begin
       wait for 150 ns;
       opcode <= "10101010"; -- test for opcode
       enable_x <= '1';
    m <= '1';

       wait for 150 ns;
       opcode <= "1111000"; -- test for opcode
       write_enable <= '1';

       wait for 150 ns;
       opcode <= "10101010";
       enable_y <= '1';
       m <= '1';

       wait for 150 ns;
       opcode <= "10100000";
       enable_x <= '1';
       m <= '0';

       wait for 150 ns;
       opcode <= "01100110";
       enable_y <= '1';
       m <= '0';



end process opcode_test;

end test2;
```

```
-----------------------------------------------------------------
-- Filename: tester3.vhd
-- Descripton: Testbench for top-level design of CRAM
-- By: CRAM Group (referenced from EE 552 Class Notes)
-- Date: November 12, 2001
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity tester3 is
end tester3;


architecture test3 of tester3 is

-- send instructions to controller and verify results

component top_level is
       port (clock,reset: in std_logic;
              Instruction : out std_logic_vector(15 downto 0);
             display1,display2,display3: out std_logic_vector(6 downto 0);
              RAM_value_read : out std_logic_vector (0 downto 0);
             RAM_value_written : out std_logic_vector (0 downto 0);

             xreg,yreg,muxout,muxin : out std_logic_vector(0 downto 0);
             write_ad,read_ad : out std_logic_vector(7 downto 0);
             enx,eny, RAM_we, Bus_Enable : out std_logic
                            );
end component top_level;


signal clock, reset : std_ulogic;

begin

top_level_check : top_level
                           port map(
                           clock => clock,
                           reset => reset
                           );

clock_period : process
begin
       clock <= '0';
       wait for 50 ns;
       clock <= '1';
       wait for 50 ns;

end process clock_period;

--- reset process not used currently.
--reset_test : process
--begin
       --reset <= '1';
--     reset <= '0'; after 5 ns;
--     wait for 1 ms;
--end process


end test3;
```

```
-- Filename:   top_level.vhd
-----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library lpm;
use lpm.lpm_components.all;

entity top_level is
       generic(
               zero : positive := 0;
               size : positive := 7);


              port ( clock,reset: in std_logic;
              --memory_value : out std_logic_vector (zero downto 0);
                      --opcode : out std_logic_vector(size downto 0);
                      Instruction : out std_logic_vector(15 downto 0);
                    --display1,display2,display3: out std_logic_vector(size - 1
downto 0);
                      RAM_value_read : out std_logic_vector (zero downto 0);
                    RAM_value_written : out std_logic_vector (zero downto 0);

                    -- result : out std_logic_vector(zero downto 0);
                    xreg,yreg,muxout,muxin : out std_logic_vector(zero downto
0);
                    write_ad,read_ad,opcode_out : out std_logic_vector(size
downto 0);
                    enx,eny, RAM_we, Bus_Enable : out std_logic
                      -- En_X, En_Y, junk_we : out std_logic;
                      -- En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out
std_logic
                     );
end top_level;

architecture structural of top_level is
       signal internal_add_rd,internal_add_wr,Int_opcode :
std_logic_vector (size downto 0);
       signal internal_enable_memory : std_logic;
      signal Int_En_Write_Reg, Int_Shift_R, Int_Shift_L : std_logic;
      signal Int_Bus_Enable,Int_En_X,Int_En_Y : std_logic;
      signal memory_value : std_logic_vector (zero downto 0);
      signal xreg1,yreg1,muxout1 :std_logic_vector(zero downto 0);
      signal rom_data : std_logic_vector(zero downto 0);
      signal RAM_value_int : std_logic_vector(zero downto 0);
       --signal Int_RAM_data_in : std_logic;
       --signal Int_Write_Enable_Register;


       component rom_control is
              generic(
               zero : positive := 0;
               size : positive := 7);
              port(
                     clock, reset : in std_logic;
                     address_rd,address_wr : out std_logic_vector (size downto
0);
                   opcode : out std_logic_vector (size downto 0);
```

```
                    En_X, En_Y, En_memory : out std_logic;
                     En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out std_logic;
                     junk : out std_logic_vector(15 downto 0)
                     );
end component rom_control;


component rom2 is
       generic (rom_data_width : positive := 1; -- width of data in rom
                 rom_address_width : positive := 8;-- width of address in rom
                 rom_data_size : positive := 13);     -- number of values in rom
       PORT(  clock : in std_logic;
             address1 : in std_logic_vector(7 downto 0);
              rom_data: out std_logic_vector(rom_data_width -1 downto 0));
              -- data in address indicated by "counter"
end component rom2;

component ram2 is
       generic (rom_data_width : positive := 1; -- width of data in ram
       rom_address_width : positive := 8;      -- width of address in ram
       rom_data_size : positive := 13); -- number of values in ram
       PORT(  clock:  in std_logic;                   --system clock
              memory_address : in std_logic_vector(7 downto 0);
             write_enable : in std_logic;
             data_to_ram : in std_logic_vector (rom_data_width -1 downto 0);
              ram_out : out std_logic_vector( rom_data_width -1 downto 0));
       -- output data
end component ram2;

--component dqff is
--port (clock,enable : in std_logic;
--    D: in std_logic_vector(0 downto 0);
--  Q: out std_logic_vector(0 downto 0)
--);
-- end component dqff;

component pe_dp_loop
 generic(
               zero : positive := 0;
               size : positive := 7);
       port (clock,enable_x,enable_y: in std_logic;
          bus_enable,shift_right,shift_left : in std_logic;
          opcode : in std_logic_vector(size downto 0);
          --x,y: in std_logic;
          m_test : out std_logic_vector(zero downto 0);
          m: in std_logic_vector(zero downto 0);
          --y: buffer std_logic;
          x_int,y_int,mux_d : out std_logic_vector(zero downto 0)
              );
end component pe_dp_loop;

--component display_7seg is
--    generic(
--            width :positive := 5
--      );
--      port(
--       clk : in std_logic;
--    input : in  std_logic_vector( 3 downto 0);
--    output: out std_logic_vector( width + 1 downto 0)
--    );
--   end component display_7seg;
```

```
--component display7seg2 is
--generic(width :positive := 4;
          zero : positive := 0);

-- port(
-- clk: in std_logic;
-- input1: in std_logic_vector(zero downto 0);
-- input2: in std_logic_vector(zero downto 0);
-- output : out std_logic_vector(width + 2 downto 0)
-- );
--end component display7seg2;


begin

 RC_Component : rom_control
   port map(
              reset => reset,
              Clock => clock,
           --clock_rom => clock_rom,
              address_rd => internal_add_rd,
             address_wr => internal_add_wr,
              opcode => Int_opcode,
              En_X => Int_En_X,
              En_Y => Int_En_Y,
       En_memory => internal_enable_memory,
              En_Write_Reg => Int_En_Write_Reg,
              Shift_R => Int_Shift_R,
              Shift_L => Int_Shift_L,
              Bus_Enable => Int_Bus_Enable,
              junk => Instruction
       );

    Rom_with_data : rom2
      port map(
             clock=>clock,
             address1 => internal_add_rd,
               rom_data=>memory_value
                            );
    R2_Component : ram2
       port map(
              clock => clock,
              memory_address => internal_add_wr,
              write_enable => internal_enable_memory,
              data_to_ram => muxout1,
              ram_out => RAM_value_int
                          );

       --Data_write_back_RAM: dqff
      --               port map (clock=>clock,
      --          enable=>internal_enable_memory,
       --         D=>muxout1,
       --       Q=>RAM_value_int
       --     );


    PE : pe_dp_loop
      port map(
           clock => clock,
           enable_x => Int_En_X,
           enable_y => Int_En_Y,
           bus_enable => Int_Bus_Enable,
           shift_right => Int_Shift_R,
           shift_left => Int_Shift_L,
           x_int=>xreg1,
           y_int=>yreg1,
           mux_d=>muxout1,

          opcode     => Int_opcode,
           --wr_en_reg_output => Int_Write_Enable_Register,
           --write_reg_en => Int_En_Write_Reg,
          m => memory_value
              );

       --display_op_3to1: display_7seg
       --              port map (
       --          clk => clock,
       --            input =>Int_opcode(width-1 downto 0)  ,
       --            output =>display1
       --            );
       -- display_op_7to4: display_7seg
       --      port map (
       --      clk => clock,
       --      input => Int_opcode(size downto width) ,
       --      output =>display2
        --    );
   --  display_x_y: display7seg2
       --      port map (
       --       clk => clock,
       --         input1 =>xreg1,
       --          input2 => yreg1,
       --          output => display3
       --      );

--p1: process(clock)
   -- begin
     --mux_d<=a;
     --m_test<=m;
     xreg<=xreg1;
     yreg<=yreg1;
     muxout<=muxout1;
     enx<=Int_En_X;
     eny<=Int_En_Y;
     RAM_value_read <= memory_value;
     Bus_Enable <= Int_Bus_Enable;
     RAM_value_written <= Ram_value_int ;
     muxin <= memory_value;
     write_ad<=internal_add_wr;
     read_ad<=internal_add_rd;
     opcode_out<=Int_opcode;
     -- end process p1;
RAM_we <= internal_enable_memory;
end structural;
```

```
-- Filename:   control2.vhd
-----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

package controller_pkg is

component control2 is
       generic(
               size : positive := 7);

       port (
                clock, reset: in std_logic;
                instruction : in std_logic_vector (15 downto 0);
                address_rd,address_wr : out std_logic_vector (size downto 0);
                opcode : out std_logic_vector (size downto 0);
                En_X, En_Y, En_memory : out std_logic;
          En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out std_logic
              );

end component control2;

end package controller_pkg;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.controller_pkg.all;

library lpm;
use lpm.lpm_components.all;

entity control2 is
       generic(

               size : positive := 7);

       port (
                clock, reset : in std_logic;
                instruction : in std_logic_vector (15 downto 0);
                address_rd,address_wr : out std_logic_vector (size downto 0);
                opcode : out std_logic_vector (size downto 0);
                En_X, En_Y, En_memory : out std_logic;
               select1, dout1 : out std_logic_vector(size-6 downto 0);
                En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out std_logic
              );
end entity control2;

architecture mixed of control2 is

    signal sel : std_logic_vector(1 downto 0);
    signal dout : std_logic_vector(1 downto 0);

begin
```

```
sel <= instruction(15 downto 14);

process (reset,clock)
   begin
        if reset = '1' then
                    opcode <= "00000000";
                    Bus_Enable <= '0';
                    En_X <= '0';
                    En_Y <= '0';
                    En_memory <= '0';
                    Shift_L <= '0';
                    Shift_R <= '0';
                 En_Write_Reg<='0';

             elsif (clock'event and clock = '1') then

      case sel is
         when "00"=> dout <= "00";
                     address_rd <= instruction(size downto 0);

         when "01"=> dout <= "01";
                     address_wr <= instruction(size downto 0);

         when others => dout <= "10";
                     opcode <= instruction(14 downto size);
                          Bus_Enable <= instruction(0);
                          En_X <= instruction(1);
                        En_Y <= instruction(2);
                          En_memory <= instruction(3);
                        Shift_L <= instruction(4);
                          Shift_R <= instruction(5);
                          En_Write_Reg <= instruction(6);
     end case;
end if;
   end process;

select1<=sel;
dout1<=dout;

end mixed;
```

```
-- Filename: clk_div.vhd
-- Description: clock divider
------------------------------------------------------------------------------------------------------------------

LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY clk_div IS

        PORT
        (
                clock_25Mhz                             : IN        STD_LOGIC;
                clock_1MHz                              : OUT       STD_LOGIC;
                clock_100KHz                    : OUT     STD_LOGIC;
                clock_10KHz                             : OUT       STD_LOGIC;
                clock_1KHz                              : OUT       STD_LOGIC;
                clock_100Hz                             : OUT       STD_LOGIC;
                clock_10Hz                              : OUT       STD_LOGIC;
                clock_1Hz                               : OUT       STD_LOGIC);

END clk_div;

ARCHITECTURE a OF clk_div IS

        SIGNAL count_1Mhz: STD_LOGIC_VECTOR(4 DOWNTO 0);
        SIGNAL count_100Khz, count_10Khz, count_1Khz : STD_LOGIC_VECTOR(2 DOWNTO 0);
        SIGNAL count_100hz, count_10hz, count_1hz : STD_LOGIC_VECTOR(2 DOWNTO 0);
        SIGNAL  clock_1Mhz_int, clock_100Khz_int, clock_10Khz_int, clock_1Khz_int: STD_LOGIC;
        SIGNAL clock_100hz_int, clock_10Hz_int, clock_1Hz_int : STD_LOGIC;
BEGIN
        PROCESS
        BEGIN
-- Divide by 25
                WAIT UNTIL clock_25Mhz'EVENT and clock_25Mhz = '1';
                        IF count_1Mhz < 24 THEN
                                count_1Mhz <= count_1Mhz + 1;
                        ELSE
                                count_1Mhz <= "00000";
                        END IF;
                        IF count_1Mhz < 12 THEN
                                clock_1Mhz_int <= '0';
                        ELSE
                                clock_1Mhz_int <= '1';
                        END IF;

                -- Ripple clocks are used in this code to save prescalar hardware
                -- Sync all clock prescalar outputs back to master clock signal
                        clock_1Mhz <= clock_1Mhz_int;
                        clock_100Khz <= clock_100Khz_int;
                        clock_10Khz <= clock_10Khz_int;
                        clock_1Khz <= clock_1Khz_int;
                        clock_100hz <= clock_100hz_int;
                        clock_10hz <= clock_10hz_int;
                        clock_1hz <= clock_1hz_int;
        END PROCESS;

                -- Divide by 10
        PROCESS
```

```
BEGIN
        WAIT UNTIL clock_1Mhz_int'EVENT and clock_1Mhz_int = '1';
                IF count_100Khz /= 4 THEN
                        count_100Khz <= count_100Khz + 1;
                ELSE
                        count_100khz <= "000";
                        clock_100Khz_int <= NOT clock_100Khz_int;
                END IF;
END PROCESS;

        -- Divide by 10
PROCESS
BEGIN
        WAIT UNTIL clock_100Khz_int'EVENT and clock_100Khz_int = '1';
                IF count_10Khz /= 4 THEN
                        count_10Khz <= count_10Khz + 1;
                ELSE
                        count_10khz <= "000";
                        clock_10Khz_int <= NOT clock_10Khz_int;
                END IF;
END PROCESS;

        -- Divide by 10
PROCESS
BEGIN
        WAIT UNTIL clock_10Khz_int'EVENT and clock_10Khz_int = '1';
                IF count_1Khz /= 4 THEN
                        count_1Khz <= count_1Khz + 1;
                ELSE
                        count_1khz <= "000";
                        clock_1Khz_int <= NOT clock_1Khz_int;
                END IF;
END PROCESS;

        -- Divide by 10
PROCESS
BEGIN
        WAIT UNTIL clock_1Khz_int'EVENT and clock_1Khz_int = '1';
                IF count_100hz /= 4 THEN
                        count_100hz <= count_100hz + 1;
                ELSE
                        count_100hz <= "000";
                        clock_100hz_int <= NOT clock_100hz_int;
                END IF;
END PROCESS;

        -- Divide by 10
PROCESS
BEGIN
        WAIT UNTIL clock_100hz_int'EVENT and clock_100hz_int = '1';
                IF count_10hz /= 4 THEN
                        count_10hz <= count_10hz + 1;
                ELSE
                        count_10hz <= "000";
                        clock_10hz_int <= NOT clock_10hz_int;
                END IF;
END PROCESS;

        -- Divide by 10
PROCESS
BEGIN
        WAIT UNTIL clock_10hz_int'EVENT and clock_10hz_int = '1';
```

```
                        IF count_1hz /= 4 THEN
                                count_1hz <= count_1hz + 1;
                        ELSE
                                count_1hz <= "000";
                                clock_1hz_int <= NOT clock_1hz_int;
                        END IF;
        END PROCESS;

END a;
```

```
-- Filename:   debounce.vhd
-- Description: debounce circuit
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

        -- Debounce Pushbutton: Filters out mechanical switch bounce for around 40Ms.
ENTITY debounce IS
        PORT(pb, clock_100Hz        : IN        STD_LOGIC;
                pb_debounced                   : OUT      STD_LOGIC);
END debounce;

ARCHITECTURE a OF debounce IS
        SIGNAL SHIFT_PB                        : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

        -- Debounce clock should be approximately 10ms or 100Hz
        PROCESS
        BEGIN
                WAIT UNTIL (clock_100Hz'EVENT) AND (clock_100Hz = '1');
                -- Use a shift register to filter switch contact bounce
                SHIFT_PB(2 DOWNTO 0) <= SHIFT_PB(3 DOWNTO 1);
                SHIFT_PB(3) <= NOT PB;
                IF SHIFT_PB(3 DOWNTO 0)="0000" THEN
                        PB_DEBOUNCED <= '0';
                ELSE
                        PB_DEBOUNCED <= '1';
                END IF;
        END PROCESS;
END a;
```

```
-- Filename:     bin_to_led.vhd
-- Description:  binary to 7 segment led converter
--
-- Modified by:  Sue Ann Ung
-- Date:         September 28, 2001
-----------------------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity bin_to_led is
        port (input: in std_logic_vector (3 downto 0);
        output: out std_logic_vector (6 downto 0));
end entity bin_to_led;

architecture behavioral of bin_to_led is
        begin
                with input select
                        output <=
                                "1000000" when "0000",
                                "1111001" when "0001",
                                "0100100" when "0010",
                                "0110000" when "0011",
                                "0011001" when "0100",
                                "0010010" when "0101",
                                "0000010" when "0110",
                                "1111000" when "0111",
                                "0000000" when "1000",
                                "0010000" when "1001",
                                "0001000" when "1010",
                                "0000011" when "1011",
                                "1000110" when "1100",
                                "0100001" when "1101",
                                "0000110" when "1110",
                                "0001110" when "1111",
                                "1000000" when others;
end behavioral;
```

```
-- Filename:    lcd_package.vhd
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package lcd_package is

component lcd_out is
        generic(
                data_out_width: positive := 8;
                data_in_width: positive := 9
        );
        port(
                lcd_data_in : in std_logic_vector(data_in_width-1 downto 0);
                lcd_data_out : out std_logic_vector(data_out_width-1 downto 0);
                lcd_register_select, lcd_nenable : out std_logic;

                clock, reset : in std_logic;
                extra_delay, lcd_latch : in std_logic;
                lcd_complete : out std_logic
        );

end component lcd_out;

component segdisplay is
        generic(countwidth: positive := 4; segment_width : positive := 7);
   port(count: in std_logic_vector(countwidth-1 downto 0);
        display: out std_logic_vector(segment_width-1 downto 0)
   );
end component segdisplay;

component lcd is
        generic(
                data_in_width : positive := 16;
                data_out_width : positive := 8;
                mode_width : positive := 4
        );

        port(
--              ** external device ports **
                lcd_data_out : out std_logic_vector(data_out_width-1 downto 0);
                lcd_register_select : out std_logic;
                lcd_nenable : out std_logic;
                lcd_rw : out std_logic;

--              ** internal device ports **
                clock : in std_logic;
                areset : in std_logic;
--              lcd_data_in : in std_logic_vector(data_in_width-1 downto 0);
                lcd_mode : in std_logic_vector(3 downto 0);
                lcd_mode_chg : in std_logic;
                lcd_done: buffer std_logic;
                segment1, segment2: out std_logic_vector(6 downto 0)
        );
end component lcd;

end lcd_package;
```

```
-- Lab 3
-- Filename:    count4bit.vhd
-- Description: 4 bit counter with asynchronous reset
--              modified from count2bit.vhd
-- Modified by: Sue Ann Ung
-- Date:        September 28, 2001

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity count4bit is
        generic ( counterwidth: positive := 4;
       segments: positive := 7);

        port (    clock_25MHz :     in std_logic;
                   manual_clock:  in std_logic;
             reset :                    in std_logic;
                   led:                          out std_logic_vector (segments-1 downto 0));
end count4bit;

architecture behavior of count4bit is
        signal bin          :     std_logic_vector (counterwidth-1 downto 0);
        signal clock_100Hz  :         std_logic;
        signal system_clock :         std_logic;


        component bin_to_led
                port (input:          in std_logic_vector (counterwidth-1 downto 0);
                   output:            out std_logic_vector (segments-1 downto 0));
        end component bin_to_led;


        component debounce
                port (pb              :     in std_logic;
                         clock_100Hz :      in std_logic;
                   pb_debounced: out std_logic
                );
        end component debounce;

        component clk_div
                port (clock_25Mhz                         : IN      STD_LOGIC;
                         clock_1MHz                             : OUT     STD_LOGIC;
                         clock_100KHz                           : OUT     STD_LOGIC;
                         clock_10KHz                            : OUT     STD_LOGIC;
                         clock_1KHz                             : OUT     STD_LOGIC;
                         clock_100Hz                            : OUT     STD_LOGIC;
                         clock_10Hz                             : OUT     STD_LOGIC;
                         clock_1Hz                                      : OUT     STD_LOGIC);

        end component clk_div;


        begin
                counter: process (system_clock, reset)
                        begin
                                --if rising_edge (system_clock) then
                                        --bin <= bin + '1';
                --elsif reset = '0' then
                        --bin <= (others => '0');
                --end if;

                if reset = '0' then
                        bin <= (others => '0');
                elsif rising_edge (system_clock) then
                        bin <= bin + '1';
                end if;
        end process counter;


        whatever : clk_div
                port map (
                        clock_25Mhz => clock_25Mhz,
                        clock_100Hz => clock_100Hz);

        decode: bin_to_led
                port map (
                        input => bin,
                        output => led);

        bounce : debounce
                port map (
                        pb => manual_clock,
                        clock_100Hz => clock_100Hz,
                        pb_debounced => system_clock);


end behavior;
```

```
-- Filename:  lcd.vhd
-- Description: LCD Controller Interface
------------------------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library lpm;
use lpm.lpm_components.all;
library work;
use work.lcd_package.all;

entity lcd is
        generic(
                data_in_width : positive := 16;
                data_out_width : positive := 8;
                mode_width : positive := 4
        );

        port(
--              ** external device ports **
                lcd_data_out : out std_logic_vector(data_out_width-1 downto 0);
                lcd_register_select : out std_logic;
                lcd_nenable : out std_logic;
                lcd_rw : out std_logic;

--              ** internal device ports **
                clock : in std_logic;
                areset : in std_logic;
--              lcd_data_in : in std_logic_vector(data_in_width-1 downto 0);
                lcd_mode : in std_logic_vector(3 downto 0);
                lcd_mode_chg : in std_logic;
                lcd_done: buffer std_logic;
                segment1, segment2: out std_logic_vector(6 downto 0)
        );
end entity lcd;

-- architecture for lcd entity
architecture lcd_arch of lcd is

type state_type is (mpoweron, initialization, initialization_wait, wait_for_mode, wait_address,
                                load_address, get_char, display_char, inc_address);

signal state : state_type;

constant clr_screen : std_logic_vector(11 downto 0) := X"001";          -- clear the lcd
signal address, mode_address: std_logic_vector(7 downto 0);
signal ch : std_logic_vector(11 downto 0);
signal dsp_done, screen_refresh : std_logic;
signal latch_add, display_enable : std_logic;
signal name_dis, increment : std_logic;
signal lcd_complete, char_valid, delay: std_logic;
signal setup, init_int: std_logic;
signal clock_count, init_time: std_logic_vector(19 downto 0);
signal latch_address, init: std_logic;

begin

        lcd_rw <= '0';
        init_time <= X"5B8D8"; -- used to create a 15ms delay 0x5B8D8
        LCD_1:  lcd_out
        generic map(data_out_width => data_out_width,
                                data_in_width => 9)
        port map(lcd_data_in => ch(8 downto 0),
                        lcd_data_out => lcd_data_out,
                        lcd_register_select => lcd_register_select,
                        lcd_nenable => lcd_nenable,
                        clock => clock,
                        reset => areset,
                        extra_delay => delay,
                        lcd_latch => char_valid,
                        lcd_complete => lcd_complete
        );

        seg1: segdisplay
                port map(count => ch(7 downto 4),
                                display => segment1
                );

        seg2: segdisplay
                port map(count => ch(3 downto 0),
                                display => segment2
                );


        startrom: lpm_rom
                generic map(
                        LPM_WIDTH => 12,
                        LPM_WIDTHAD => 8,
                        LPM_FILE => "lcd.mif"
                )
                port map(
                        inclock => clock,
                        outclock => clock,
                        address => address,
                        q => ch
                );

        with lcd_mode select
                mode_address <=
                "00000101" when "0000",     -- poweron
                "00010001" when "0001",     -- command
                "00101001" when "0010",     -- from ?
                "00111100" when "0011",     -- waiting for tx
                "01100011" when "0100",     -- downloading
                "01001100" when "0101",     -- streaming
                "01011000" when "0110",     -- complete msg
                "10100011" when "0111",     -- blank screen
                "10000011" when "1000", -- play icon
                "10001001" when "1001", -- pause icon
                "10000110" when "1010", -- stop icon
                "10001100" when "1011", -- delete? 1=OK
                "10011011" when "1100", -- deleted
                -- when "0110",      -- default song
                -- when "0111",      -- default name
--              mplay when "1000",          -- play icon
--              mpause when "1001",         -- pause icon
--              mstop when "1010",          -- stop icon
--              mdel_what when "1011",      -- delete ?
--              mdel_done when "1100",      -- deleted
                "01110001" when others;     -- anything else
```

```
with state select
        latch_add <=
        '1' when initialization_wait,
        '1' when load_address,
        '1' when wait_address,
        '0' when others;

with state select
        lcd_done <=
        '1' when wait_for_mode,
        '0' when others;

with state select
        char_valid <=
        '1' when display_char,
        '0' when others;

with state select
        name_dis <=
        '0' when initialization,
        '0' when others;

with state select
        init_int <=
        '1' when initialization_wait,
        '1' when initialization,
        '0' when others;

with state select
        increment <=
        '1' when inc_address,
        '0' when others;

with ch select
        delay <=
                '1' when clr_screen,
                '0' when others;

comb_logic: process (clock, areset)
begin
        if areset = '1' then
                state <= mpoweron;
        elsif rising_edge(clock) then
                case state is
                        when mpoweron =>
                                state <= initialization_wait;
                        when initialization_wait =>
                                if setup = '1' then
                                        state <= initialization;
                                else
                                        state <= initialization_wait;
                                end if;
                        when initialization =>
                                state <= get_char;
                        when wait_for_mode =>
                                if lcd_mode_chg = '1' then
                                        state <= load_address;
                                else
                                        state <= wait_for_mode;
                                end if;
                        when load_address =>
```

```
                                        state <= wait_address;
                                when wait_address =>
                                        if mode_address = address then
                                                state <= get_char;
                                        end if;
                                when get_char =>
                                        state <= display_char;
                                when display_char =>
                                        if ch = X"100" then
                                                state <= wait_for_mode;
                                        elsif lcd_complete = '1' then
                                                state <= inc_address;
                                        else
                                                state <= display_char;
                                        end if;
                                when inc_address =>
                                        state <= get_char;
                        end case;
                end if;
        end process comb_logic;


        regs2: process(clock, areset)
        begin
                if areset = '1' then
                        address <= (others => '0');
                elsif rising_edge(clock) then
                        if latch_address = '1' and init = '1' then
                                address <= (others => '0');
                        elsif latch_address = '1' and name_dis = '0' then
                                address <= mode_address;
                        elsif latch_address = '1' and name_dis = '1' then
                                address <= mode_address;
                        elsif increment = '1' then
                                address <= address + '1';
                        end if;
                end if;
        end process regs2;

        init_count: process(clock, areset)
        begin
                if areset = '1' then
                        latch_address <= '0';
                        setup <= '0';
                        clock_count <= (others => '0');
                elsif rising_edge(clock) then
                        latch_address <= latch_add;
                        init <= init_int;
                        if init = '1' then
                                if clock_count = init_time then
                                        setup <= '1';
                                else
                                        clock_count <= clock_count + '1';
                                end if;
                        end if;
                end if;
        end process init_count;
end architecture lcd_arch;
```

```
-- Filename:   lcd_out.vhd
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity lcd_out is
        generic(
                data_out_width: positive := 8;
                data_in_width: positive := 9
        );
        port(
                lcd_data_in : in std_logic_vector(data_in_width-1 downto 0);
                lcd_data_out : out std_logic_vector(data_out_width-1 downto 0);
                lcd_register_select, lcd_nenable : out std_logic;

                clock, reset : in std_logic;
                extra_delay, lcd_latch : in std_logic;
                lcd_complete : out std_logic
        );

end entity lcd_out;

architecture lcd_display of lcd_out is

type state_type is (wait_for_char, load_data, set_count, wait_time);

signal state : state_type;
signal load_char, clear_count, count_down : std_logic;
signal count, clock_count, time_count : std_logic_vector(15 downto 0);
signal nenable : std_logic;

begin

        with extra_delay select
                count <=
                        X"FFFF" when '1',  -- used to create 1.5ms delay 0x927C
                        X"0FFF" when '0',  -- used to create 40us delay 0x03E8
                        X"0FFF" when others;
--                      X"002F" when '1',  -- used to create 1.5ms delay 0x927C
--                      X"000A" when '0',  -- used to create 40us delay 0x03E8
--                      X"000A" when others;

        with state select
                nenable <=
                        '0' when wait_time, -- used to trigger enable (active low) LCD
                        '0' when wait_for_char,
                        '1' when others;

        with state select
                lcd_complete <=
                        '1' when wait_for_char,
                        '0' when others;

        with state select
                count_down <=
                        '1' when wait_time,
                        '0' when others;
        with state select
                clear_count <=
                        '1' when set_count, -- clear the counters and set the time delay
                        '0' when others;

        with state select
                load_char <=
                        '1' when load_data,  -- used to latch lcd_data_in to lcd_data_out
                        '0' when others;

        -- purpose to step through the states of execution
        comb_logic : process(clock, reset)
        begin
                if reset = '1' then     -- active high asynchronous reset
                        state <= wait_for_char;
                elsif rising_edge(clock) then
                        case state is
                                when wait_for_char =>
                                        if lcd_latch = '1' then
                                                state <= load_data;
                                        else
                                                state <= wait_for_char;
                                        end if;
                                when load_data =>
                                        state <= set_count;
                                when set_count =>
                                        state <= wait_time;
                                when wait_time =>
                                        if time_count = clock_count then
                                                state <= wait_for_char;
                                        else
                                                state <= wait_time;
                                        end if;
                        end case;
                end if;
        end process comb_logic;

        process (clock, reset)
        begin
                if reset = '1' then
                        time_count <= (others => '0');
                elsif rising_edge(clock) then
                        lcd_nenable <= nenable;
                        if load_char = '1' then
                                lcd_data_out <= lcd_data_in(7 downto 0);
                                lcd_register_select <= lcd_data_in(8);
                        elsif clear_count = '1' then
                                time_count <= (others => '0');
                                clock_count <= count;
                        elsif count_down = '1' then
                                time_count <= time_count + '1';
                        end if;
                end if;
        end process;

end lcd_display;
```

```
-- Filename: lcd.mif
-- Description: MIF file for LCD Displays
--------------------------------------------------------------------------------

WIDTH = 12;
DEPTH = 17;
ADDRESS_RADIX = DEC;
DATA_RADIX = HEX;
CONTENT BEGIN

-- initialize --
0: 001;         -- clear screen
1: 03C;    -- two-lines set function set
2: 00C;         -- display-on
3: 006;         -- set auto increment cursor or use 014 for cursor/display shift
4: 100;         -- null

5: 001;         -- clear screen
6: 143;         -- C
7: 12D;         -- hyphen
8: 152;         -- R
9: 141;         -- A
10: 14D;  -- M
11: 120;  -- space
12: 147;  -- G
13: 152;  -- R
14: 14F;  -- O
15: 155;  -- U
16: 150;  -- P
17: 0C0;   -- next line
18:     153;         -- S
19:     148;         -- H
20:     141;         -- A
21:     148;         -- H
22:     149;         -- I
23:     144;         -- D
24:     120;         -- space
25:     153;         -- S
26:     155;         -- U
27:     145;         -- E
28:     120;         -- space
29:     153;         -- S
30:     141;         -- A
31: 154;        -- T
32:     14E;         -- N
33:     145;         -- E
34:     145;         -- E
35:     156;         -- V
36:     100;         -- null
END;
```

```
----------------------------------
-- top-level file using two PEs
-- top_level1.vhd
-- C-RAM group
----------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library lpm;
use lpm.lpm_components.all;

entity top_level1 is
        port ( clock,reset,clock_rom: in std_logic;
             Instruction : out std_logic_vector(15 downto 0);
             display1,display2,display3: out std_logic_vector(6 downto 0);
             RAM_value_read,RAM_value_read2 : out std_logic_vector (0 downto 0);
            RAM_value_written,RAM_value_written2 : out std_logic_vector (0 downto 0);
            xreg,yreg,muxout,muxin : out std_logic_vector(0 downto 0);
            write_ad,read_ad,opcode1 : out std_logic_vector(7 downto 0);
            enx,eny, RAM_we, Bus_Enable : out std_logic
        );
end top_level1;

architecture structural of top_level1 is

  signal internal_add_rd,internal_add_wr,Int_opcode : std_logic_vector (7 downto 0);
  signal internal_enable_memory : std_logic;
  signal Int_En_Write_Reg, Int_Shift_R, Int_Shift_L : std_logic;
  signal Int_Bus_Enable,Int_En_X,Int_En_Y : std_logic;
  signal memory_value,memory_value2 : std_logic_vector (0 downto 0);
  signal xreg1,yreg1,muxout1,xreg2,yreg2,muxout2 :std_logic_vector(0 downto 0);
  signal rom_data,rom_data2 : std_logic_vector(0 downto 0);
  signal RAM_value_int,RAM_value_int2 : std_logic_vector(0 downto 0);

        component rom_control is
                port(
                        clock,reset,clock_rom : in std_logic;
                        address_rd,address_wr : out std_logic_vector (7 downto 0);
                        opcode : out std_logic_vector (7 downto 0);
                        En_X, En_Y, En_memory : out std_logic;
                        En_Write_Reg, Shift_R, Shift_L, Bus_Enable : out std_logic;
                        junk : out std_logic_vector(15 downto 0)
                        );
        end component rom_control;

   component ramPE1 is
        generic ( rom_data_width : positive := 1;                -- width of data in rom
                  rom_address_width : positive := 8;    -- width of address in rom
                  rom_data_size : positive := 13);      -- number of values in rom

        PORT (    clock : in std_logic;
                  address1 : in std_logic_vector(7 downto 0);
            rom_data: out std_logic_vector(rom_data_width -1 downto 0)); -- data in address indicated by "counter"
  end component ramPE1;
  component ramPE2 is
        generic (  rom_data_width : positive := 1;                    -- width of data in rom
                   rom_address_width : positive := 8;        -- width of address in rom
                   rom_data_size : positive := 13);          -- number of values in rom

        PORT(    clock : in std_logic;
                  address1 : in std_logic_vector(7 downto 0);
        rom_data: out std_logic_vector(rom_data_width -1 downto 0)); -- data in address indicated by "counter"
 end component ramPE2;

  component ram2PE1 is
                          generic (  rom_data_width : positive := 1;                    -- width of data in ram
                          rom_address_width : positive := 8;        -- width of address in ram
                          rom_data_size : positive := 13);          -- number of values in ram
        PORT(    clock:  in std_logic;                    --system clock
                          memory_address : in std_logic_vector(7 downto 0);
                          write_enable : in std_logic;
                          data_to_ram : in std_logic_vector (rom_data_width -1 downto 0);
                          ram_out : out std_logic_vector( rom_data_width -1 downto 0));         -- output data
        end component ram2PE1;

  component ram2PE2 is
                   generic (  rom_data_width : positive := 1;                    -- width of data in ram
                          rom_address_width : positive := 8;        -- width of address in ram
                          rom_data_size : positive := 13);          -- number of values in ram
        PORT(    clock:  in std_logic;                    --system clock
                   memory_address : in std_logic_vector(7 downto 0);
                  write_enable : in std_logic;
                  data_to_ram : in std_logic_vector (rom_data_width -1 downto 0);
                   ram_out : out std_logic_vector( rom_data_width -1 downto 0));         -- output data
        end component ram2PE2;


 component pe_dp_loop
              port (enable_x,enable_y,clock: in std_logic;
                    bus_enable,shift_right,shift_left : in std_logic;
                    opcode : in std_logic_vector(7 downto 0);
                    m_test : out std_logic_vector(0 downto 0);
                    m: in std_logic_vector(0 downto 0);
                    x_int,y_int,mux_d : out std_logic_vector(0 downto 0)
                   );
  end component pe_dp_loop;

  component display_7seg is
                   generic(
                   width :positive := 5
                          );
                     port(
                       clk : in std_logic;
                       input : in  std_logic_vector( 3 downto 0);
                       output: out std_logic_vector( 6 downto 0)
                          );
  end component display_7seg;

 component display7seg2 is
   --generic(width :positive := 4);

 port(
   clk: in std_logic;
   input1: in std_logic_vector(0 downto 0);
   input2: in std_logic_vector(0 downto 0);
   output : out std_logic_vector(6 downto 0)
  );
end component display7seg2;


        begin

        RC_Component : rom_control
                port map(
                        reset => reset,
                        Clock => clock,
                        clock_rom => clock_rom,
                        address_rd => internal_add_rd,
                        address_wr => internal_add_wr,
                        opcode => Int_opcode,
                        En_X => Int_En_X,
                        En_Y => Int_En_Y,
                        En_memory => internal_enable_memory,
                        En_Write_Reg => Int_En_Write_Reg,
                        Shift_R => Int_Shift_R,
                        Shift_L => Int_Shift_L,
                        Bus_Enable => Int_Bus_Enable,
                        junk => Instruction
                                                );
        Rom_with_data1 : ramPE1
            port map
                    (
                        clock=>clock,
                        address1 => internal_add_rd,
                        rom_data=>memory_value
                    );

       Rom_with_data2 : ramPE2
            port map
                    (
                        clock=>clock,
                        address1 => internal_add_rd,
                        rom_data=>memory_value2
                    );

        R2_Component1 : ram2PE1
                    port map(
                        clock => clock,
                        memory_address => internal_add_wr,
                        write_enable => internal_enable_memory,
                        data_to_ram => muxout1,
                        ram_out => RAM_value_int
                      );
 R2_Component2 : ram2PE2
                    port map(
                        clock => clock,
                        memory_address => internal_add_wr,
                        write_enable => internal_enable_memory,
                        data_to_ram => muxout2,
                        ram_out => RAM_value_int2
                      );

  PE1 : pe_dp_loop
                    port map(
                        clock => clock,
                        enable_x => Int_En_X,
                        enable_y => Int_En_Y,
                        bus_enable => Int_Bus_Enable,
                                        shift_right => Int_Shift_R,
                                        shift_left => Int_Shift_L,
                                        x_int=>xreg1,
                                        y_int=>yreg1,
                                        mux_d=>muxout1,
                                        opcode => Int_opcode,
                                        m => memory_value
                                                                );
   PE2 : pe_dp_loop
                                port map(
                                        clock => clock,
                                        enable_x => Int_En_X,
                                        enable_y => Int_En_Y,
                                        bus_enable => Int_Bus_Enable,
                                        shift_right => Int_Shift_R,
                                        shift_left => Int_Shift_L,
                                        x_int=>xreg2,
                                        y_int=>yreg2,
                                        mux_d=>muxout2,
                                        opcode => Int_opcode,
                                        m => memory_value2
                                        );

   display_op_3to1: display_7seg
                               port map (
                                        clk => clock,
                                        input =>Int_opcode(3 downto 0)  ,
                                        output =>display1
                                        );
    display_op_7to4: display_7seg
                                port map (
                                        clk => clock,
                                        input => Int_opcode(7 downto 4) ,
                                        output =>display2
                                        );

   display_x_y: display7seg2
                                 port map (
                                        clk => clock,
                                        input1 =>xreg1,
                                        input2 => yreg1,
                                        output => display3
                                        );


                                xreg<=xreg1;
                                yreg<=yreg1;
                                muxout<=muxout1;
                                enx<=Int_En_X;
                                eny<=Int_En_Y;
                                RAM_value_read <= memory_value;
                                RAM_value_read2 <= memory_value2;
                                Bus_Enable <= Int_Bus_Enable;
                                RAM_value_written <= Ram_value_int ;
                                RAM_value_written2 <= Ram_value_int2 ;
                                muxin <= memory_value;
                                write_ad<=internal_add_wr;
                                read_ad<=internal_add_rd;
                                opcode1<=Int_opcode;
                                RAM_we <= internal_enable_memory;

    end structural;
```

Simulation Waveforms

Controller Simulation

The controller was thoroughly tested with a number of test cases. The controller functioned correctly, as it passed from its initial state of idle to either the operation state or the read state depending on the instruction received. On the Controller Waveform 1 in the Appendix an instruction containing an OpCode and enable bits was sent as an input. The controller received the instruction and moved into the operation state by correctly identifying the instruction as an OpCode from the flag bit (which is set to 1). The controller then set the OpCode signal to the correct OpCode, in this case OpCode = 10101010 and set the correct enable bits, En_X and En_Memory.

In Controller Waveform 2 we have sent varied the instructions from those, which are OpCodes to those, which contain RAM addresses. Again the controller received the instruction in the idle state and then moved into the next state corresponding to the instruction. From 0 to 500ns, and instruction = 1101010100001010 was sent into the controller. The controller again correctly identified this as an OpCode from the flag bit, set the OpCode = 10101010 and initiated the En_X and En_Memory signals to high. The controller then returned back into its starting state of idle to receive the next instruction. Between 500 ns and 1.0 us an instruction = 0000000000000000 is inputted into the controller. The controller identifies this as an RAM address from the flag bit and sets the address signal = 00000001. (As mentioned earlier, only the 8 least significant bits are used as the address, the rest are masked off). Again the controller, then returns to the idle state to await the next instruction.

We can see that the controller is functioning correctly. It interprets the instruction as either an OpCode or an address and sets the correct signals corresponding to the instruction. If the flag bit is equal to 1 the controller moves into the operation state, sets the OpCode for the PE and enables the correct registers. If the flag bit is equal to 0, the controller moves to the read state, and sets the address bit to the corresponding address in the instruction and outputs it to the RAM.

Controller & ROM Simulation

The next phase of testing was to test the controller with the ROM. A mif file was created which contained instructions for the controller and was stored in the ROM. A simple counter was used to access the ROM and pass on the instructions to the controller. To test the controller working with the ROM, we simply verified, the outputs of the controller with the instructions in the mif file.

In Controller + ROM Waveform in the Appendix. The signal "junk" is equal to the instructions in the mif file and is inputted into the controller. By verifying which instruction is sent against what signals are outputted, we can determine if the controller and ROM components are functioning correctly. The first instruction in the mif file is an address of 0000000000000000. We can see this on signal junk. The controller moves from the idle state to the read state and interprets the instruction from the mif file as an RAM address, thus sets the address line = 00000000. The controller then moves back to the initial state of idle. The next instruction in the mif file is an OpCode, junk = 1101010100000010. The controller interprets this instruction as an OpCode, and moves into the operation state. It then sets the OpCode = 10101010, and sets the En_X = 1, as determined by the instruction. The controller returns to the idle state. From 200ns to 275ns, an address instruction = 0000000000000101 is sent to the controller. (This is the third instruction in the mif file). Again we can see the controller moves into the correct state of read and sets the address signal to 0000101. Finally, the next instruction in the mif file is an OpCode, and from 275ns to 350ns, instruction = 1101000000000100 (seen as the junk signal) is sent to the controller. The controller sets the OpCode = 10100000 and sets the En_Y signal = 1.

By verifying the correct outputs of the controller against the instructions in the mif file stored in the ROM we see that the controller is correctly receiving the instructions from the ROM and outputting the correct results to either the PE or the RAM.

Test cases for the PE Simulation

The PE takes in the OpCode from the controller along with some other signals to enable the registers (x, y and write enable register) RAM, buses. The controller also sends in the shift right and shift left signals to the PE. We used internal signals port mapped to outputs to check the contents of the registers and memory.
We haven't done the RAM interface for writing the data back in to RAM so we are using a register as RAM and writing data back to that register.
The Enable signals are set manually in this PE simulation but for the top level File "ram_cont_ram.vhd" all the control signals are generated by the controller.
In the waveform attached in the Appendix we simulated following test cases.

OpCode	**Operation**	**Other signals**	**Expected Result**	**Result Obtained**
10101010	load x or y from data stored in mif file	Enable x is set to high	load the data at first memory location into x	x = 1 as data in RAM is 1
11110000	writing data back to RAM from x register	write enable is set to 1	RAM should go to logic 1 as x is 1	RAM=1
10101010	load x or y form the data stored in mif file	Enable y is set to high	Load data from next memory location into y	y = 1 as next data is 1.
10101010	load x or y form mif file	enable y is set to high	Load data from next location in memory into y	y =0 as next data is 0
10100000	x = x and m performs the and operation	x is enabled to write the result of and operation in y register	Data in register x is 1 and data in next memory location is 0 so and operation gives a 0	x register goes low i.e. 0
01100110	y= x xor m performs an x-or operation	y is enabled	Data in register x is 1 and data in next memory location is 0 so xor operation gives 1	y register goes high

Since we changed the controller from State machine to a selector, the following test cases are simulated to demonstrate that the PE now works better.

C.RAM Simulation (PE = 1)

CASE1: (M =! Y)
Reads data from a mif file (ram.mif) in the RAM.
The data read from the RAM is inverted and written to register y.
The data in register y is written back to RAM.

The mif files and the simulations for the CASE1 are attached. The mif files are labeled according to the data they have. The simulations for the top-level test cases are so arranged that 1st page shows the whole simulation to see if the result is correct or not. Then in next few pages the simulation is spread over a few pages to see the instructions and op-code and addresses changing clearly.

So for CASE1 data in mif file is read as 0,0,11 and inverted and written to y as 1,1,0,0 and same is written back to RAM as 1,1,0,0.

C.RAM Simulation (PE = 1)

ADDITION

Data is read from the RAM (B[0]) and sent to register X.
Data is read from the RAM (A[0]) and its ANDED with data already in X and sent to Y.
Data in X is XOR with data in RAM (A[0]) and written into and X as well as into RAM.
RAM is disabled for Writing.

These steps are repeated three more times with one additional step and we get the final sum of two four bit numbers.

The Data in the mif file(ram_sum.mif) is B= 1001 and A = 1001
The sum being one bit at a time and written back to RAM should be Sum =0010
We get 0010, which is the correct result therefore C-RAM is working properly.
Since we changed the controller from State machine to a selector it seems to work better.

C.RAM Simulation (PE = 2)

ADDITION

For both PEs the following instruction set is the same:

Data is read from the RAM (B[0]) and sent to register X.
Data is read from the RAM (A[0]) and its ANDED with data already in X and sent to Y.
Data in X is XOR with data in RAM (A[0]) and written into and X as well as into RAM.
RAM is disabled for Writing.

These steps are repeated three more times with one additional step and we get the final sum of two four bit numbers.

The Data in the mif file of PE #1 is B= 1111 and A = 1111
The Data in the mif file of PE #2 is B = 1010 and A = 0101

The sum being one bit at a time and written back to RAM should be Sum #1 = 1110
and sum for PE #2 is: Sum #2 = 1111.

The simulations show that we get a result of 1110 for PE #1 and 1111 for PE #2, which is the correct result therefore C-RAM is working properly.