Experiment #6: Completing the Microprocessor

Steven Comer

Implementation Due: 11 Dec 2012
Report Due: 11 Dec 2012

Steven Comer

Contents

1.0 Introduction

The purpose of this experiment was to learn about State Machine design with Read Only Memory (ROM) and Random Access Memory (RAM). The experiment made use of components constructed in previous experiments. These included RAM, ROM, Program Counter (PC) and Arithmetic Logic Unit (ALU). New components included an Address Decoder, Tri-state Buffers, State Machine, and Central Processing Unit (CPU). The ROM served as a simulated external device providing instructions for the state machine to execute. The RAM served as a second external device, which could be written to and read from. The experiment required implementation of a control unit to interpret instructions and provide control signals for the state machine. When implemented correctly, the state machine displays the sequence {1, 3, 5, 7, 9, B, D, F, 6, 4, 2, 0} to a seven-segment display.

2.0 Method

2.1 Background Information

2.1.1 Register

Registers are simple storage units. The registers used in this experiment are each four bits wide. If the load signal for a register is high on the rising edge of a clock signal, the register updates its output to the current input. Three distinct registers are used in this experiment. The first is the Instruction Register (IR). The IR stores the instruction currently being executed. The second and third registers are the Operand Register HI (ORH) and the Operand Register LO (ORL). The operand registers work together to provide the program counter with an 8-bit address for branch instructions.

2.1.2 Read Only Memory

Read only memory (ROM) is a type of memory can only be read from, not written to. In the case of this experiment, ROM contains the instructions to be executed. The ROM delivers instructions to the CPU by outputting memory locations onto the data bus, which feeds into the control unit via the instruction register.

2.1.3 Random Access Memory

Random access memory (RAM) may be read from or written to. In a personal computer RAM is high speed memory which increases the number of processes a user may run simultaneously. RAM allows two operations, read and write. The read operation is synchronous and requires a valid address and a high readEnable signal. The value stored at the indicated memory address is then output. The write operation is also synchronous and requires a valid address and a high writeEnable signal. The 2-bit value from input will be stored in the indicated address of memory. When a write operation occurs, the output is set to "00". Simultaneous reading and writing is not allowed. If the user attempts to read and write simultaneously, no operation is performed.

2.1.4 Tri-State Buffer

Tri-state buffers have three possible states adding high impedance to the typical high and low possibilities. Outputting high impedance is useful when dealing with busses which may be written to by multiple components. An output of high impedance essentially outputs nothing. A TSB can be compared to a switch, which can be open or closed. The TSB should output high impedance, like an open switch, when another source is writing a shared bus. When the TSB is enabled, it simply passes the input to the output.

2.1.5 Address Decoder

An address decoder works with the TSB's and other components to control access to a shared busses. The ROM and display each occupy distinct portions of memory. The address decoder supplies read enable signals for the ROM and display based on the value of the address bus. Values on the data bus come from two sources, ALU and ROM. The address decoder ensures mutually exclusive writing to the data bus to prevent a race condition. It outputs enable signals for each of the tri-state buffers, one for ALU and one for ROM. The enable signals depend on the instruction currently being executed.

2.1.6 Program Counter

A Program Counter (PC) is a register that contains the address of the instruction being executed at the current time. After an instruction is fetched, the program counter increments its stored value by one, and points to the next instruction in the sequence. The program counter can be cleared or incremented. The clear, increment, and load signals are all synchronous inputs. There also exists an internal adder module which must produce the sum of the input pins and the X- register. Figure 2.1 lists the possible outputs of the PC.

0 - Program Counter
1 - Sum, X + Inputs
2 - Inputs
3 - X register

Figure 2.1 PC Output Select

2.1.7 Arithmetic Logic Unit

An arithmetic logic unit (ALU) is a synchronous digital circuit used to perform arithmetic and logical operations. A typical ALU loads data from two input registers and performs a specific operation on one or both of the inputs, based on the function-select input. The result is then stored in a register. The ALU optionally sets a flag, Z, when the output is zero. Figure 2.2 lists the operations which can be performed by the ALU on input registers A and B.

0 - ADD	4 - NOT A
1 - SUB (A - B)	5 - Accumulator
2 - AND	6 - NOT B
3 - OR	7 - B Register

Figure 2.2 - ALU Function Select

2.1.8 Control Unit

The control unit provides coordination between all the components of the CPU. It directs the flow of data between the components by providing timing and control signals. It has a defined set of procedures to follow for each specific instruction which exists in the project. Based on the current instruction, the control unit generates specific signals which allow it to execute correctly.

2.1.9 Central Processing Unit

The central processing unit (CPU) is the entity which includes the control unit, IR, ORH, ORL, program counter, and ALU. As an entity, the CPU executes instructions which are input by external devices such as ROM or user programs. Data travels between the CPU and external components by means of busses. In this case, the CPU communicates data via a 4-bit data bus and an 8-bit address bus.

2.2 Procedure

See Appendix A for complete VHDL code listing. Note that only new components, not used in experiments prior to Lab 5, are addressed in this section.

2.2.1 Tri-state Buffer Design

The tri-state buffer has two inputs and a single output. The input include an enable bit that is received from the address decoder. the other input is a 4-bit value. The output is determined by the value of the enable bit it this bit is high then the output is equal to the 4-bit input. If the enable bit is low high impendence is outputted. See Figure 2.3 for diagram.

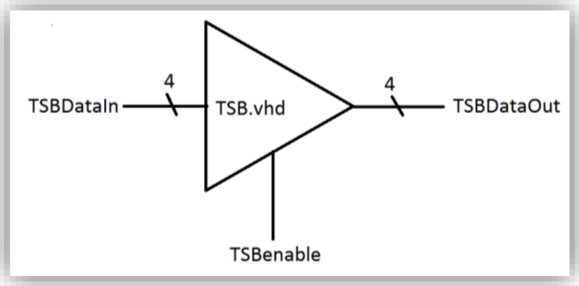


Figure 2.3 - Tri-state Buffer Diagram

2.2.2 Address Decoder Design

The address decoder has four inputs and three outputs. The inputs are an 8-bit address, a read command, a write command, and a clock. The clock is used to determine when the process is run for changing the value of the tri-state buffers. The read and write commands along with the address is used for determining the values of the enable bit sent to the tri-state buffers. The outputs are the enable bits for the tri-state buffers. The first output is the enable ROM bit this will be high when the address is less than 31 and the read command is active with the write command set low. The PORTLED enable bit is high when write is high, read is low, and the address is 96. Then the ALU enable bit is high when neither the read and write command are high. In this case the value of the address is unimportant. See Figure 2.4 for diagram.

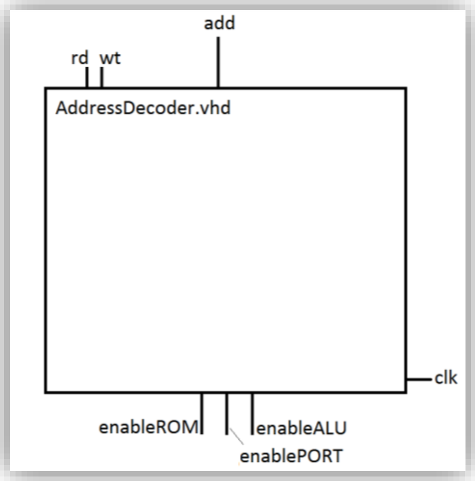


Figure 2.4 - Address Decoder Diagram

2.2.3 ROM Design
The ROM uses two input values and a single output value. The first input value is a clock that determines when the process is run. The second input value is an 8-bit address that determines which value from the ROM is output to the data bus via the tri-state buffer. The output is the value stored at the requested address.

2.2.4 State Machine Design
The state machine uses three inputs and based on these inputs sets the values of sixteen outputs. The first input is the 4-bit instruction code this is used to determine the type instruction being performed. This then helps to determine the values of the outputs. The next input is the clock which determines when the process runs that sets the new step the program is on. The final input is the reset sends the state machine to a safe state where the program can restart. The outputs are the signals needed for all of the other components. These are the signals for the instruction register, the program counter, the operand register low, the operand register high, the ALU, and the read and write commands. Figure 2.6 shows the high-level state transition diagram used to construct the state machine. Red arrows indicate error conditions which branch back to reset. Blue arrows indicate valid transitions. All instructions proceed through the fetch stage, decode stage, and at least one execute stage. Figure 2.5 lists the ending state of each instruction.

The first step of any instruction is the Fetch stage. In this stage, the instruction to be run is loaded into the instruction register from the data bus.

The second step is the Decode. Here the program counter is incremented to point to the next instruction in memory.

The third step is Execute. The length of this step varies depending on the instruction. For a simple instruction such as INCX, the only thing that happens here is setting the INCX signal on the program counter high. For multi-register instructions, the arguments must be loaded in the equivalent of another decode stage.

The fourth step is Memory. In this stage, variants of the store instruction write their results into memory, which is either RAM or the PORTLED for this lab.

Upon completion, all instructions return to first state where the next instruction is fetched, decoded and executed.

Opcode	Instruction	Final State
0	LDA #	0100
1	LDA mem	1001
2	LDA mem, x	1001
3	STA mem	1001
4	STA mem, x	1001
5	ANDA	0101
6	ORA	0101
7	COMA	0100
8	ADDA	0101
9	SUBA	0101
A	CMPA	0101
B	BRA	1000

C	BEQ	1000
D	BNE	1000
E	CLRX	0011
F	INCX	0011

Figure 2.5 - Instruction Transitions

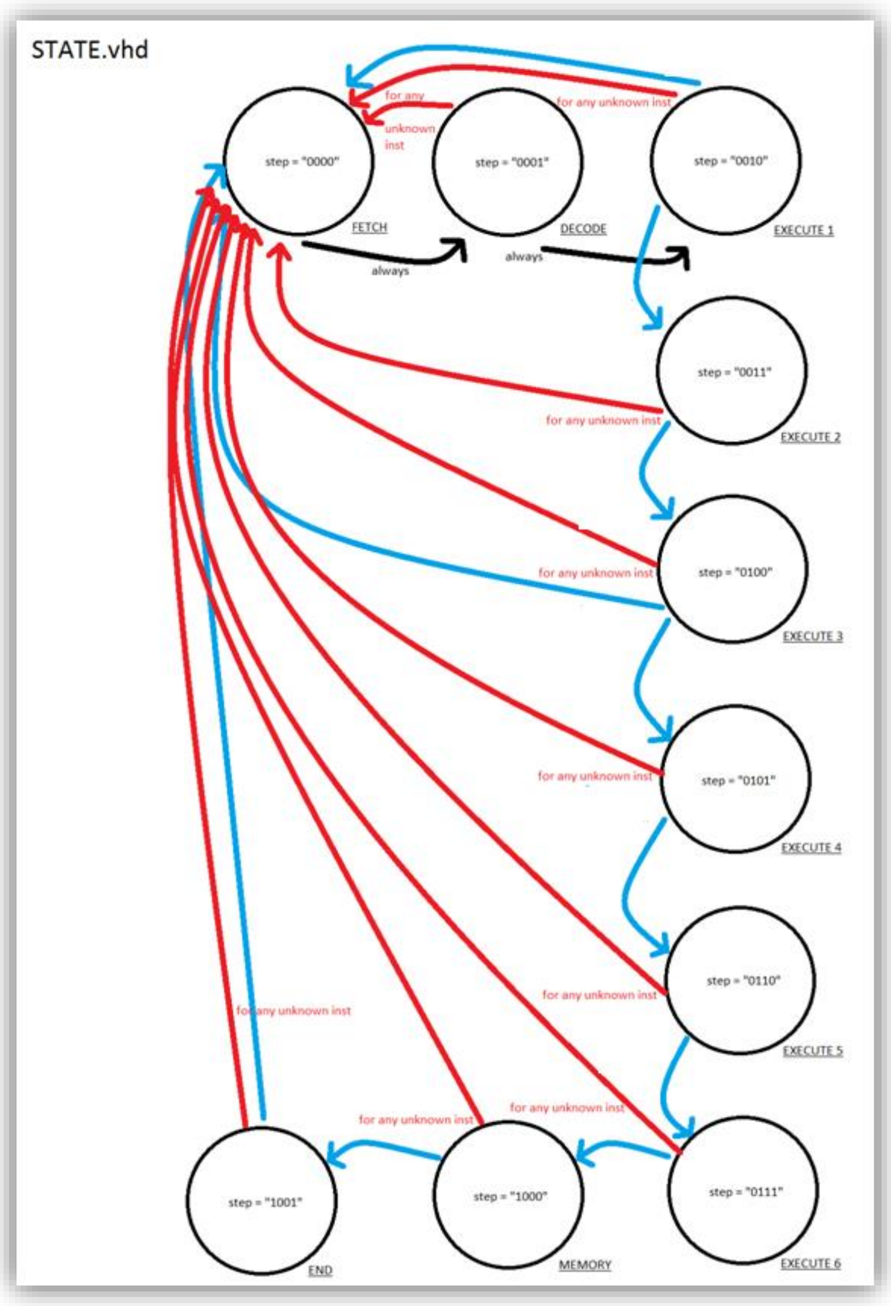


Figure 2.6 - State Transition Diagram

2.2.5 CPU Design

This was the upper level entity. It has two inputs and ten outputs. The inputs would be the clock button and the reset switch. The outputs consist of the seven segment displays and two LEDs. The inputs are given directly by the user through the interface of the Altera board. The outputs are then displayed on the board for the user. Figure 2.7 shows the diagram for the entire CPU including its components.

This paragraph describes the overall interaction of the CPU with its components. The first step is for the ROM to place an instruction on the data bus. Next, the control unit sets the IR load signal to load the instruction into the instruction register. The control unit then decodes the instruction and increments the program counter. If the instruction has multiple registers these values are fetched and decoded as well. If the additional register provide a memory address, the address is sent to the Address Decoder for interpretation. The Address Decoder determines whether to enable the ROM, RAM, or PORTLED, based on the address and type of access. If the access is read, the selected memory component then takes in the address and outputs the value contained at that address. If the access is write, the RAM or PORTLED loads the value from the data bus and stores it at the location given by the address bus. The Address Decoder also provides enable signals for the tri-state buffers, again based on access type and address.

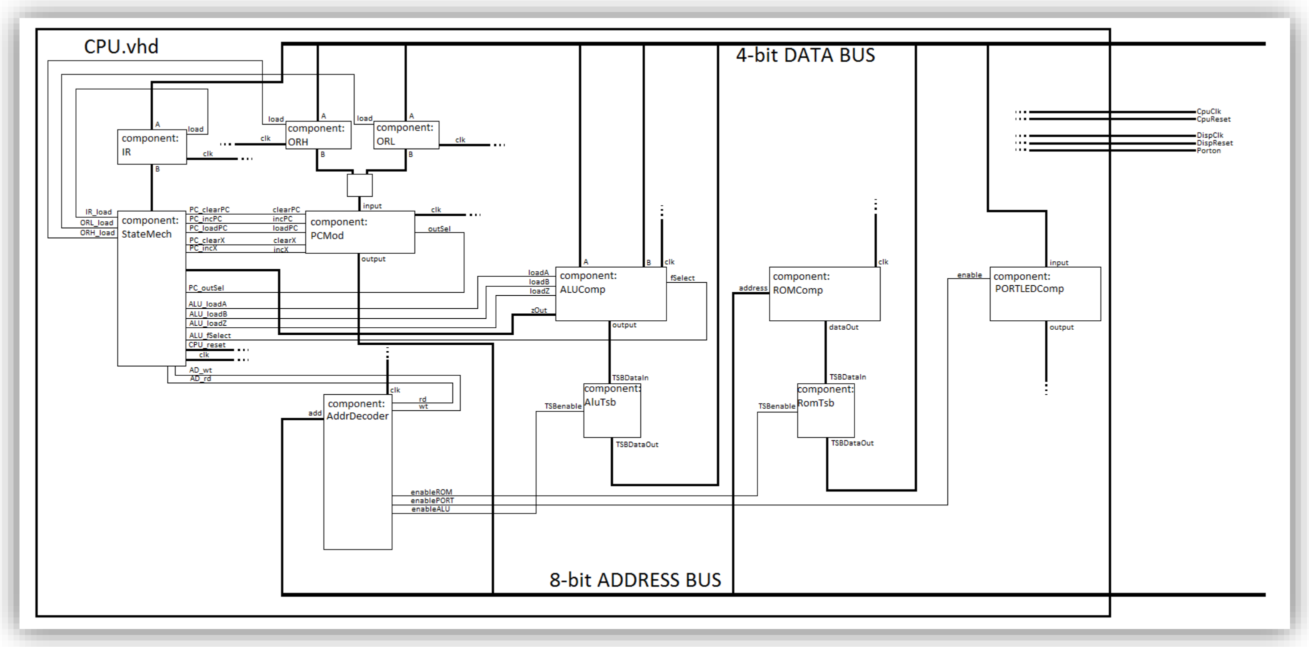


Figure 2.7 - CPU Diagram

2.2.6 Pin Assignments
The clock is determined by a push button inputted to the circuit. Pushing the button then releasing it corresponds to one clock cycle. The button gives a high signal when pressed. The button corresponds to KEY[3] of the Altera board.

The reset is set by a switch inputted to the circuit. When the switch was in the up position then the reset would be high and the program would go to a safe state in the next clock cycle. When the switch is in the down position then the reset would be low and the program would continue. The switch corresponds to SW[0] of the Altera board.

The seven segment displays showed the values of the address in HEX7 and HEX6. It showed the values of the values of the Data bus in HEX5 and the value of the ALU in HEX4. The value of the current instruction was displayed on HEX3 and the current ORH and ORL values were shown on HEX2 and HEX1, respectively. The PORTLED was displayed on HEX0.

The current step is displayed by four LEDs corresponding to LEDR[17]-LEDR[14]. The step was moved to LEDs to free up Hexadecimal display slot.

The remaining outputs were all LEDs. First the reset LED was displayed on a red LED corresponding to LEDR[0]. The clock's LED was shown on LEDG[7] lighting when the clock was pressed. The PORTLED enabled LED light was set to LEDG[0] and lit when the PORTLED could be written to. The Z flag LED was set to LEDG[1] and lit when the Z flag is high.

2.3 Problems and Solutions
Problem: In CPU.vhd, we lost control of control signal and bus updates.
Solution: We made a PROCESS sensitive to "Update," which is a bit that flips on every upward clock edge. In this PROCESS, temporary (next) values for every control signal and bus are stored onto the (current) signals.

Problem: We wanted a convenient way to keep track of read/write/enable controls for everything attached to the CPU.
Solution: Handle all address decoding in a separate component, an Address Decoder, described in AddressDecoder.vhd. We made it so read/write and current address go into the Address Decoder, and enable signals for components attached to the CPU go out.

Problem: In State.vhd, we needed an efficient way to iterate over fetch/decode/execute/memory on clock, considering that the number of execute cycles required varies across all instructions.
Solution: We created a two-tiered iteration system. The step and tstep signals (in State.vhd) keep track of the position of the CPU in the fetch/decode/execute/memory cycle, and case statements in each execute step set tstep according to how far a given instruction needs to go to reach completion. On clock, tstep is stored onto step.

Problem: Values in RAM appear to be random and independent of values written to RAM.
Solution: It turned out that the least significant digit of the address was being stored to RAM instead of the value on the data bus. This was fixed by changing the input signal in CPU.vhd from address to databus.
Problem: TSB for ROM did not allow required writing to the data bus.
Solution: Conditional statement which enabled ROM TSB was coded incorrectly. The conditional statement was changed from SERam to SERom and then it worked correctly.

2.4 Materials and Tools Used

The materials and tools used in this experiment are as follows:

1. Quartus II Software
2. Altera DE 2 Board with Cyclone II – EP2C35F672C6 FPGA

3.0 Results

3.1 Observations

Because this project was composed almost exclusively of programming, observations relate to the output of the FPGA in the form of the results. The project functioned correctly proceeding through the program as described in Section 3.2, Testing Procedure.

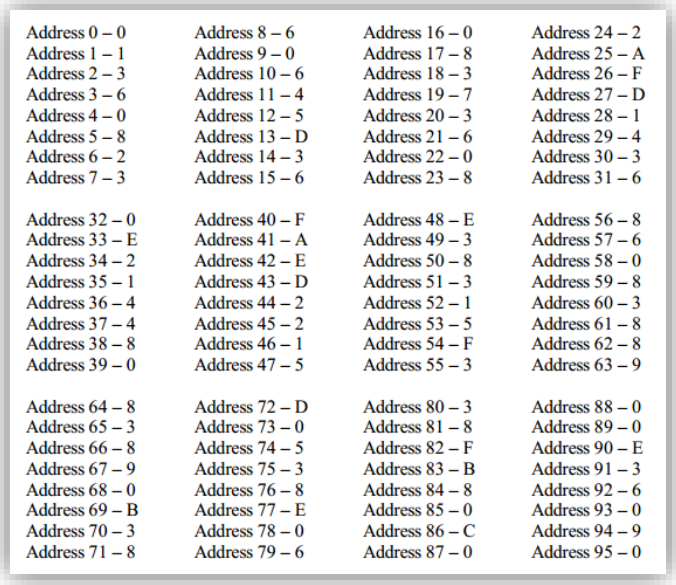
3.2 Testing Procedure

The demonstration of the project involved implementation of a test plan. The plan was mechanically simple: press clock input button until program is finished. The work for the test plan focused on the operation of the program.

The test plan analyzes the program stored in ROM at three different levels: memory values, program flow, and high level output.

3.2.1 Memory Values

This portion describes the contents of the ROM and RAM components. Note that the values included in RAM are the final values, as several addresses in RAM are written to multiple times. See Figures 3.1 and 3.2 for details.



Address 0 – 0	Address 8 – 6	Address 16 – 0	Address 24 – 2
Address 1 – 1	Address 9 – 0	Address 17 – 8	Address 25 – A
Address 2 – 3	Address 10 – 6	Address 18 – 3	Address 26 – F
Address 3 – 6	Address 11 – 4	Address 19 – 7	Address 27 – D
Address 4 – 0	Address 12 – 5	Address 20 – 3	Address 28 – 1
Address 5 – 8	Address 13 – D	Address 21 – 6	Address 29 – 4
Address 6 – 2	Address 14 – 3	Address 22 – 0	Address 30 – 3
Address 7 – 3	Address 15 – 6	Address 23 – 8	Address 31 – 6
Address 32 – 0	Address 40 – F	Address 48 – E	Address 56 – 8
Address 33 – E	Address 41 – A	Address 49 – 3	Address 57 – 6
Address 34 – 2	Address 42 – E	Address 50 – 8	Address 58 – 0
Address 35 – 1	Address 43 – D	Address 51 – 3	Address 59 – 8
Address 36 – 4	Address 44 – 2	Address 52 – 1	Address 60 – 3
Address 37 – 4	Address 45 – 2	Address 53 – 5	Address 61 – 8
Address 38 – 8	Address 46 – 1	Address 54 – F	Address 62 – 8
Address 39 – 0	Address 47 – 5	Address 55 – 3	Address 63 – 9
Address 64 – 8	Address 72 – D	Address 80 – 3	Address 88 – 0
Address 65 – 3	Address 73 – 0	Address 81 – 8	Address 89 – 0
Address 66 – 8	Address 74 – 5	Address 82 – F	Address 90 – E
Address 67 – 9	Address 75 – 3	Address 83 – B	Address 91 – 3
Address 68 – 0	Address 76 – 8	Address 84 – 8	Address 92 – 6
Address 69 – B	Address 77 – E	Address 85 – 0	Address 93 – 0
Address 70 – 3	Address 78 – 0	Address 86 – C	Address 94 – 9
Address 71 – 8	Address 79 – 6	Address 87 – 0	Address 95 – 0

Figure 3.1 - ROM Values

Address	**Value**
0x80	3
0x81	6

0x82	0
0x83	9
0x84	2
0x85	A
0x86	0
0x87	D
0x88	8
0x89	0
0x8A	3
0x8B	6
0x8C	0
0x8D	B
0x8E	5
0x8F	6

Figure 3.2 - Final RAM Values

3.2.2 Program Flow

This portion describes the executed program at the instruction level. There are 127 recognizable instructions which execute, ending with a branch back to address 00. Figure 3.3 lists the instruction, arguments, Accumulator register value, X register value, and PORTLED output for each instruction. Note that @ refers to an address, Acc refers to the Accumulator register, X refers to the X register and Output refers to the PORTLED output.

#	Instruction	Arguments	Acc	X	Output
1	LDA	1	1	0	0
2	STA	@60	1	0	1
3	ADDA	2	3	0	1
4	STA	@60	3	0	3
5	ORA	4	7	0	3
6	ANDA	D	5	0	3
7	STA	@60	5	0	5
8	ADDA	3	8	0	5
9	COMA	N/A	7	0	5
10	STA	@60	7	0	7
11	ADDA	2	9	0	7
12	CMPA	F	9	0	7
13	BNE	@14	9	0	7
14	STA	@60	9	0	9
15	ADDA	2	B	0	9
16	CMPA	F	B	0	9
17	BNE	@14	B	0	9
18	STA	@60	B	0	B
19	ADDA	2	D	0	B
20	CMPA	F	D	0	B
21	BNE	@14	D	0	B
22	STA	@60	D	0	D
23	ADDA	2	F	0	D
24	CMPA	F	F	0	D
25	BNE	@14	F	0	D

26	STA	@60	F	0	F
27	CLRX	N/A	F	0	F
28	LDA	@14	3	0	F
29	STA	@80	3	0	F
30	INCX	N/A	3	1	F
31	CMPA	E	3	1	F
32	BNE	@22	3	1	F
33	LDA	@14	6	1	F
34	STA	@80	6	1	F
35	INCX	N/A	6	2	F
36	CMPA	E	6	2	F
37	BNE	@22	6	2	F
38	LDA	@14	0	2	F
39	STA	@80	0	2	F
40	INCX	N/A	0	3	F
41	CMPA	E	0	3	F
42	BNE	@22	0	3	F
43	LDA	@14	8	3	F
44	STA	@80	8	3	F
45	INCX	N/A	8	4	F
46	CMPA	E	8	4	F
47	BNE	@22	8	4	F
48	LDA	@14	2	4	F
49	STA	@80	2	4	F
50	INCX	N/A	2	5	F
51	CMPA	E	2	5	F
52	BNE	@22	2	5	F
53	LDA	@14	A	5	F
54	STA	@80	A	5	F
55	INCX	N/A	A	6	F
56	CMPA	E	A	6	F
57	BNE	@22	A	6	F
58	LDA	@14	F	6	F
59	STA	@80	F	6	F
60	INCX	N/A	F	7	F
61	CMPA	E	F	7	F
62	BNE	@22	F	7	F
63	LDA	@14	D	7	F
64	STA	@80	D	7	F
65	INCX	N/A	D	8	F
66	CMPA	E	D	8	F
67	BNE	@22	D	8	F
68	LDA	@14	1	8	F
69	STA	@80	1	8	F
70	INCX	N/A	1	9	F
71	CMPA	E	1	9	F
72	BNE	@22	1	9	F
73	LDA	@14	4	9	F
74	STA	@80	4	9	F

75	INCX	N/A	4	10	F
76	CMPA	E	4	10	F
77	BNE	@22	4	10	F
78	LDA	@14	3	10	F
79	STA	@80	3	10	F
80	INCX	N/A	3	11	F
81	CMPA	E	3	11	F
82	BNE	@22	3	11	F
83	LDA	@14	6	11	F
84	STA	@80	6	11	F
85	INCX	N/A	6	12	F
86	CMPA	E	6	12	F
87	BNE	@22	6	12	F
88	LDA	@14	0	12	F
89	STA	@80	0	12	F
90	INCX	N/A	0	13	F
91	CMPA	E	0	13	F
92	BNE	@22	0	13	F
93	LDA	@14	E	13	F
94	STA	@80	E	13	F
95	INCX	N/A	E	14	F
96	CMPA	E	E	14	F
97	BNE	@22	E	14	F
98	LDA	@5E	9	14	F
99	STA	@83	9	14	F
100	LDA	@5F	0	14	F
101	STA	@86	0	14	F
102	SUBA	8	8	14	F
103	STA	@88	8	14	F
104	SUBA	8	0	14	F
105	STA	@89	0	14	F
106	LDA	B	B	14	F
107	STA	@8D	B	14	F
108	LDA	5	5	14	F
109	STA	@8E	5	14	F
110	LDA	6	6	14	F
111	STA	@8F	6	14	F
112	BRA	@80	6	14	F
113	STA	@60	6	14	6
114	SUBA	2	4	14	6
115	CMPA	0	4	14	6
116	BNE	@80	4	14	6
117	STA	@60	4	14	4
118	SUBA	2	2	14	4
119	CMPA	0	2	14	4
120	BNE	@80	2	14	4
121	STA	@60	2	14	2
122	SUBA	2	0	14	2
123	CMPA	0	0	14	2

124	BNE	@80	0	14	2
125	STA	@60	0	14	0
126	BRA	@56	0	14	0
127	BEQ	@00	0	14	0

Figure 3.3 - Program Flow

3.2.3 High Level Output

Finally, this portion is what the user views. It is the required output of the program.

1. Display 1 on PORTLED.
2. Display 3 on PORTLED.
3. Display 5 on PORTLED.
4. Display 7 on PORTLED.
5. Display 9 on PORTLED.
6. Display B on PORTLED.
7. Display D on PORTLED.
8. Display F on PORTLED.
9. Display 6 on PORTLED.
10. Display 4 on PORTLED.
11. Display 2 on PORTLED.
12. Display 0 on PORTLED.
13. Reset and repeat.

3.3 Experiment Evaluation

The experiment was completely successful. All state transitions work as specified displaying the correct sequence of {1, 3, 5, 7, 9, B, D, F, 6, 4, 2, 0}. The reset condition also worked correctly. It correctly handled the user-input reset signal which can synchronously reset the program at the user's request.

3.4 Warning Explanations

Quartus gives four main types of messages when compiling code: info, warnings, critical warnings, and errors. The same types of warnings occurred in both the RAM and in the Reference Monitor. For this reason, the warnings will be addressed together.

3.4.1 Errors

Errors are the most important type of message. They explicitly describe to the user where in the code or design Quartus found a problem. It is essential that all errors are removed from a project, since the code will not compile correctly until the errors are fixed. It is also important to realize that just because all the errors are removed from a project does not mean the code works as intended.

3.4.2 Critical Warnings

The second most important message type is a critical warning. While these do not explicitly mean that something is wrong with the code, the user should examine these warnings when attempting to debug a functional issue.

<u>Critical Warning 1:</u>
Critical Warning (332012): Synopsys Design Constraints File file not found: 'Lab6_new.sdc'. A Synopsys Design Constraints File is required by the TimeQuest Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design.

Reason:

The project did not supply a timing constraint file, so Quartus was unable to test the timing of the circuit to see if it will work in its environment.

Critical Warning 2:
Critical Warning (332148): Timing requirements not met

Reason:
This critical warning is directly related to critical warning 1. Because a timing constraints file was not supplied, Quartus was unable to do timing analysis, so the timing requirements were not met.

3.4.3 Warnings
Warnings are common in Quartus projects. Due to the fact that Quartus compiles to an industry standard set of specifications, it is concerned with seemingly insignificant details. Quartus displayed six different types of warnings. They are described below.

Type 1: " Warning (10036): Verilog HDL or VHDL warning at CPU.vhd(154): object "SAlu_Z" assigned a value but never read"

Explanation: Zero Flag is never used by State because none of the instructions for this experiment require the Zero Flag. No instructions require this because this experiment uses no conditional branch instructions.

Type 2: " Warning (10492): VHDL Process Statement warning at CPU.vhd(208): signal "TAD_wt" is read inside the Process Statement but isn't in the Process Statement's sensitivity list"

Explanation: For all cases, signals used intentionally inside a given process but not in the sensitivity list of that given process.

Type 3: " Warning (10631): VHDL Process Statement warning at alu.vhd(119): inferring latch(es) for signal or variable "zOut", which holds its previous value in one or more paths through the process"

Explanation: For all cases, we intend these latches because we want values maintained.

Type 4: " Warning (13046): Tri-state node(s) do not directly drive top-level pin(s)"

Explanation: " The specified tri-state buffer only feeds non-tri-state logic, but the chip does not support internal tri-state buffers. Therefore, the behavior of the non-tri-state nodes is undefined when it is driven by High Impedance (Z). As a result, the Quartus II software converts the tri-state buffer to a wire as a "don't care" minimization," according to Altera.com (http://quartushelp.altera.com/11.1/mergedProjects/msgs/msgs/wmls_mls_convert_tri_to_wire.htm).

Type 5: " Warning (306006): Found 52 output pins without output pin load capacitance assignment"

Explanation: This warning states that the reported timing of these pins will be faster than in reality because load capacitance is unknown.

Type 6: " Warning (169174): The Reserve All Unused Pins setting has not been specified, and will default to 'As output driving ground'."

Explanation: The pin voltages are not specified, therefore Quartus has a fail-safe default of setting them to ground. This effect is desired because it prevents the unused pins from sending unintended signals.

3.5 Quartus Flow Summary

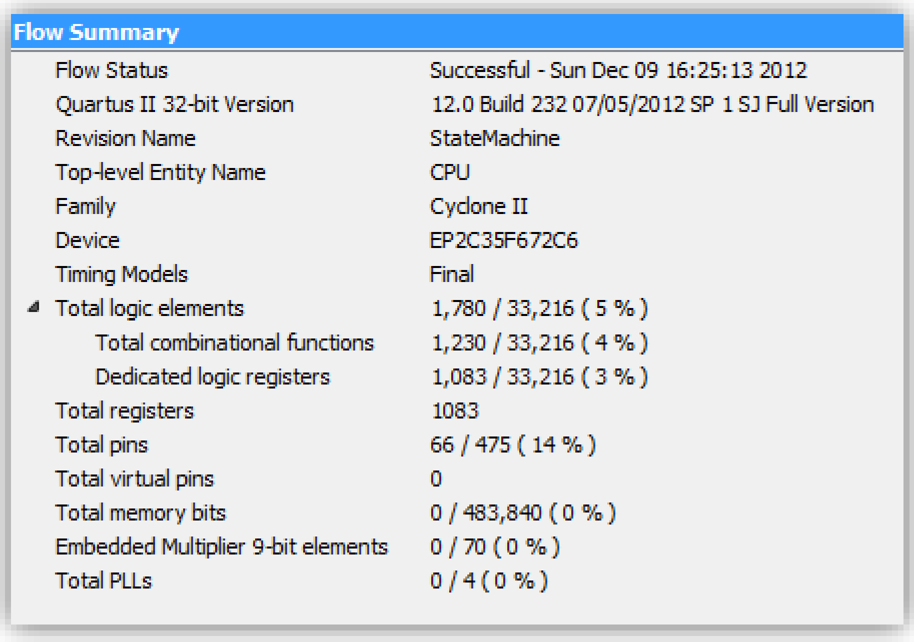
Compared with flow summaries for experiments implemented manually at the gate level in hardware, the Flow Summary here are not as useful because there is no human implementation that the Quartus implementation flow numbers can be compared against. This build includes 1780 total logic elements, 1230 total combinational functions, 1083 dedicated logic registers, and 1083 logic registers. See Figure 3.4 for the full flow summary.

Conditionals generate registers in some cases, according to The Art Of VHDL synthesis (http://www.mrc.uidaho.edu/mrc/people/jff/vhdl_info/Synthesis_Art_2P.pdf). Specifically, registers are used when conditionals in processes output to external signals. Registers are also generated for edge-sensitive conditionals.

There are conditionals (if, elsif, or case statements) throughout the CPU code. Specifically there are 28 conditionals in State.vhd, 1 in PORTLED.vhd, 1 in reg.vhd/reg1.vhd (3 registers used), 2 in ROM.vhd, 3 in ram.vhd,1 in SegDriver.vhd, 1 in TSB.vhd (3 TSB used), 5 in AddressDecoder.vhd, 4 in alu.vhd, 5 in CPU.vhd, 6 in pc.vhd.

For a rough estimate, 28+1+(1*3)+2+3+(1*3)+5+4+5+6 = 60. The rest of the logic elements can be accounted for by the RAM (128*4) and ROM (96*4). This total comes to approximately 1000, which is close to the actual total. Therefore, we expect 47 as the number of dedicated logic registers reported in the flow summary, and the flow summary reported 47 dedicated logic registers.

Similarly, with 1780 total logic elements, Quartus did not make the implementation too inefficient with logic. There is less than 1 % utilization. In conclusion, Quartus did not generate excess logic because less than 5% is minimal.



Flow Summary	
Flow Status	Successful - Sun Dec 09 16:25:13 2012
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Full Version
Revision Name	StateMachine
Top-level Entity Name	CPU
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,780 / 33,216 (5 %)
Total combinational functions	1,230 / 33,216 (4 %)
Dedicated logic registers	1,083 / 33,216 (3 %)
Total registers	1083
Total pins	66 / 475 (14 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 3.4 - Quartus Flow Summary

4.0 Conclusion

The objective of this experiment was to implement a CPU which executes a program from ROM and RAM. This project utilized components from previous experiments, but added new components including ROM, an Address Decoder, Tri-state Buffers, and a state machine. Implementing the individual components proved to be relatively simple. The majority of time and effort for this experiment was spent designing the control unit which enables and controls communication between the previously separate components. It was interesting to see the components of previous labs collaborate to make a larger entity.

A. Appendix A – Code

A.1 **CPU**

```
-------------------------------------------------------------------------------------------------
-- Lab 6
-- CPU
-- Steve Comer

-- Updated 14 Nov 2012
-- SU 397 CPU
-------------------------------------------------------------------------------------------------


LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY CPU IS
PORT(
       CpuClk,CpuReset                     : IN STD_LOGIC;
       DispClk,DispReset,Porton,DispZ      : OUT STD_LOGIC;
       DispStep
       DispData,DispAdd1,DispAdd2,DispPort : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
       DispIns,DispORH,DispORL,DispALU                   : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
);
END CPU;

ARCHITECTURE BEHAVIORAL OF CPU IS

COMPONENT State IS
PORT(
       instruction : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       clk                          : IN STD_LOGIC;
       CPU_reset   : IN STD_LOGIC;
       PC_zflag    : IN STD_LOGIC;
       Ostep                : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       IR_load     : OUT STD_LOGIC;
       PC_clearPC  : OUT STD_LOGIC;
       PC_incPC       : OUT STD_LOGIC;
       PC_loadPC   : OUT STD_LOGIC;
       PC_clearX   : OUT STD_LOGIC;
       PC_incX              : OUT STD_LOGIC;
       PC_outSel   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
       ORL_load             : OUT STD_LOGIC;
       ORH_load             : OUT STD_LOGIC;
       ALU_loadA   : OUT STD_LOGIC;
       ALU_loadB   : OUT STD_LOGIC;
       ALU_loadZ   : OUT STD_LOGIC;
       ALU_fSelect : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
       AD_wt                        : OUT STD_LOGIC;
       AD_rd                        : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT reg IS
PORT(A                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
         clk                          : IN  STD_LOGIC;
         load                : IN  STD_LOGIC;
         B                            : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;

COMPONENT pc IS
PORT(input          : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         clearPC   : IN  STD_LOGIC;
         incPC               : IN  STD_LOGIC;
         loadPC              : IN  STD_LOGIC;
         clearX              : IN  STD_LOGIC;
         incX                : IN  STD_LOGIC;
         outSel             : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
         clk                         : IN STD_LOGIC;
         output             : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

COMPONENT alu IS
PORT(A                      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
         B                           : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
         loadA              : IN  STD_LOGIC;
         loadB              : IN  STD_LOGIC;
         loadZ              : IN  STD_LOGIC;
         fSelect   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
         clk                         : IN  STD_LOGIC;
         zOut               : OUT STD_LOGIC := '0';
         output             : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;

COMPONENT SegDriver IS
PORT(
       switches   : IN STD_LOGIC_VECTOR(3 downto 0);
       segdispout : OUT STD_LOGIC_VECTOR(0 to 6)
);
END COMPONENT;

COMPONENT ram IS
PORT(
       clk          : IN STD_LOGIC;
       readEnable   : IN STD_LOGIC;
       writeEnable  : IN STD_LOGIC;
       address      : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       dataIn                : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       dataOut      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;

COMPONENT ROM IS
PORT(
       clk        : IN STD_LOGIC;
       address    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       dataOut    : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;

COMPONENT PORTLED IS
PORT(
       enable        : IN  STD_LOGIC;
       input         : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);           -- The input value for the
register
       output        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)            -- The output, for display
);
END COMPONENT;

COMPONENT AddressDecoder IS
PORT(
       add        : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       rd, wt     : IN STD_LOGIC;
       clk        : IN STD_LOGIC;
       enableROM  : OUT STD_LOGIC;
       enablePORT : OUT STD_LOGIC;
       enableALU  : OUT STD_LOGIC;
       enableRAM  : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT TSB IS
PORT(
       TSBenable  : IN STD_LOGIC;
       TSBDataIn  : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       TSBDataOut : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
```

```
END COMPONENT;

SIGNAL Update                                                                                    :
STD_LOGIC;
SIGNAL CStep
SIGNAL dataBus,AdataBus,RdataBus                          : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL addressBus                          : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SIR_load, SORH_load, SORL_load           : STD_LOGIC;
SIGNAL SEPort, SEAlu, SERom, SERam              : STD_LOGIC;
SIGNAL SAlu_Z                                                                                    :
STD_LOGIC;
SIGNAL S_DWR
SIGNAL SPC_clearPC, SPC_incPC                                            : STD_LOGIC;
SIGNAL SPC_loadPC, SPC_incX, SPC_clearX         : STD_LOGIC;
SIGNAL SALU_loadA, SALU_loadB, SALU_loadZ : STD_LOGIC;
SIGNAL SAD_wt, SAD_rd                                                                    :
STD_LOGIC;
SIGNAL SPC_outSel                                                                        :
STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SALU_fSelect                                                                              :
STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL Sinst, SAluOut, SRomOut, SRamOut         : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL SPCinput                                                                                  :
STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL portLED_data                                                                              :
STD_LOGIC_VECTOR(3 DOWNTO 0);

SIGNAL TIR_load, TORH_load, TORL_load           : STD_LOGIC;
SIGNAL TPC_clearPC, TPC_incPC                                            : STD_LOGIC;
SIGNAL TPC_loadPC, TPC_incX, TPC_clearX         : STD_LOGIC;
SIGNAL TALU_loadA, TALU_loadB, TALU_loadZ : STD_LOGIC;
SIGNAL TAD_wt, TAD_rd                                                                    :
STD_LOGIC;
SIGNAL TPC_outSel                                                                        :
STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL TALU_fSelect                                                                              :
STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

PROCESS (CpuClk)
BEGIN
       DispClk <= not CpuClk;
END PROCESS;

PROCESS(CpuReset)
BEGIN
       DispReset <= CpuReset;
END PROCESS;

PROCESS(SEPort)
BEGIN
       Porton <= SEPort;
END PROCESS;

PROCESS (CpuClk)
BEGIN
       IF (rising_edge(CpuClk) and CpuClk = '1')THEN
         IF(Update = '0')THEN
                  Update <= '1';
         ELSE
                  Update <= '0';
         END IF;
       END IF;
END PROCESS;

PROCESS(Update)
BEGIN
       SAD_wt               <= TAD_wt;
       SAD_rd               <= TAD_rd;
       SIR_load     <= TIR_load;
```

```
        SORH_load    <= TORH_load;
        SORL_load    <= TORL_load;
        SPC_clearPC  <= TPC_clearPC;
        SPC_incPC    <= TPC_incPC;
        SPC_loadPC   <= TPC_loadPC;
        SPC_incX              <= TPC_incX;
        SPC_clearX   <= TPC_clearX;
        SALU_loadA   <= TALU_loadA;
        SALU_loadB   <= TALU_loadB;
        SALU_loadZ   <= TALU_loadZ;
        SPC_outSel    <= TPC_outSel;
        SALU_fSelect <= TALU_fSelect;
        DispStep      <= CStep;
        DispZ                          <= SAlu_Z;

        IF(SEPort = '1') THEN
          dataBus <= SAluOut;
        ELSIF(SEAlu = '1') THEN
          dataBus <= AdataBus;
        ELSIF(SERam = '1' OR SERom = '1') THEN
          dataBus <= RdataBus;
        ELSE -- nonMutex attempt
          dataBus    <= "ZZZZ";
        END IF;
END PROCESS;

IR                                  : reg PORT MAP(dataBus,CpuClk,SIR_load,Sinst);
ORH                                 : reg PORT MAP(dataBus,CpuClk,SORH_load,SPCinput(7 DOWNTO
4));
ORL                                 : reg PORT MAP(dataBus,CpuClk,SORL_load,SPCinput(3 DOWNTO
0));

AluTsb                     : TSB PORT MAP(SEAlu,SAluOut,AdataBus);
RomTsb                     : TSB PORT MAP(SERom,SRomOut,RdataBus);
RamTsb                     : TSB PORT MAP(SERam,SRamOut,RdataBus);

StateMach                           : State PORT
MAP(Sinst,CpuClk,CpuReset,SAlu_Z,CStep,TIR_load,TPC_clearPC,TPC_incPC,TPC_loadPC,

TPC_clearX,TPC_incX,TPC_outSel,TORL_load,TORH_load,TALU_loadA,TALU_loadB,
                                                  TALU_loadZ,TALU_fSelect,TAD_wt,TAD_rd);
PCMod                               : pc PORT
MAP(SPCinput,SPC_clearPC,SPC_incPC,SPC_loadPC,SPC_clearX,SPC_incX,
                    SPC_outSel,CpuClk,addressBus);
ALUComp                    : alu PORT
MAP(dataBus,dataBus,SALU_loadA,SALU_loadB,SALU_loadZ,SALU_fSelect,
                    CpuClk,SAlu_Z,SAluOut);

RAMComp           : ram PORT MAP(CpuClk,SAD_rd,SAD_wt,addressBus,SAluOut,SRamOut);
ROMComp                    : ROM PORT MAP(CpuClk,addressBus,SRomOut);
PORTLEDComp                         : PORTLED PORT MAP(SEPort,dataBus,portLED_data);
AddrDecoder                         : AddressDecoder PORT
MAP(addressBus,SAD_rd,SAD_wt,CpuClk,SERom,SEPort,SEAlu,SERam);

PORTSEG                    : SegDriver PORT MAP(portLED_data,DispPort);
AddSEG1                    : SegDriver PORT MAP(addressBus(3 DOWNTO 0),DispAdd1);
AddSEG2                    : SegDriver PORT MAP(addressBus(7 DOWNTO 4),DispAdd2);
DataSEG                    : SegDriver PORT MAP(dataBus,DispData);
InsSEG                     : SegDriver        PORT MAP(Sinst,DispIns);
orhSEG                     : SegDriver PORT MAP(SPCinput(7 DOWNTO 4),DispORH);
orlSEG            : SegDriver PORT MAP(SPCinput(3 DOWNTO 0),DispORL);
ALUSEG                     : SegDriver        PORT MAP(SAluOut,DispALU);

END BEHAVIORAL;
```

A.2 State Machine

```
-----------------------------------------------------------------------------------------
-- Lab 6
-- State
-- Steve Comer
-- Updated 8 Dec 2012
-- State Machine
----------------------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;


ENTITY State IS
PORT(
         instruction : IN STD_LOGIC_VECTOR(3 DOWNTO 0);  -- The 4-bit instruction Opcode
         clk                   : IN STD_LOGIC;
         CPU_reset : IN STD_LOGIC;                                                          --
Reset the CPU
         PC_zflag          : IN STD_LOGIC;
         Ostep             : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
         IR_load     : OUT STD_LOGIC;                                                       --
Load flag for the Instruction Register
         PC_clearPC  : OUT STD_LOGIC;                                                       --
Clear input for counter
         PC_incPC     : OUT STD_LOGIC;                                                      --
Increment input for counter
         PC_loadPC   : OUT STD_LOGIC;                                                       --
Load input for counter
         PC_clearX : OUT STD_LOGIC;                                                         --
Clear input for x
         PC_incX           : OUT STD_LOGIC;
         PC_outSel : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);   -- Output select value
         ORL_load          : OUT STD_LOGIC;
         ORH_load          : OUT STD_LOGIC;
         ALU_loadA : OUT STD_LOGIC;                                                         --
Load flag for the ALU A register
         ALU_loadB : OUT STD_LOGIC;                                                         --
Load flag for the ALU B register
         ALU_loadZ : OUT STD_LOGIC;                                                         --
Load flag for the ALU Z flag
         ALU_fSelect       : OUT STD_LOGIC_VECTOR(2 DOWNTO 0); -- ALU function selector
         AD_wt                      : OUT STD_LOGIC;
         AD_rd                      : OUT STD_LOGIC
);
END State;

ARCHITECTURE BEHAVIORAL OF State IS

SIGNAL step  : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Holds the step for the operation
SIGNAL tstep : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Holds the temp step for the operation

BEGIN

PROCESS(clk)
--
BEGIN
         IF(rising_edge(clk))THEN
                  IF (CPU_reset = '1')THEN
                           step <= "1111";
                           Ostep <= "1111";
                  ELSE
                           step <= tstep;
                           Ostep <= tstep;
                  END IF;
         END IF;
END PROCESS;

PROCESS(step)
-- int vars
VARIABLE intInstr : INTEGER;
```

```
-- Fetch, Decode, Execute, Memory
BEGIN
         -- set int vars
         intInstr := conv_integer(instruction);

         -- reset all signals
         IR_load     <= '0';
         PC_incPC    <= '0';
         PC_clearPC  <= '0';
         PC_loadPC   <= '0';
         PC_clearX   <= '0';
         PC_incX     <= '0';
         PC_outSel   <= "00";
         ORL_load    <= '0';
         ORH_load    <= '0';
         ALU_loadA   <= '0';
         ALU_loadB   <= '0';
         ALU_loadZ   <= '0';
         ALU_fSelect <= "101";
         AD_wt       <= '0';
         AD_rd       <= '0';

         -- Put ROM onto data bus
         IF(step = "0000")THEN
                  AD_rd       <= '1';
                  tstep       <= "0001";

         --Fetch
         ELSIF(step = "0001")THEN
                  IR_load     <= '1';
                  AD_rd       <= '1';
                  tstep        <= "0010";

         --Decode
         ELSIF(step = "0010")THEN
                  PC_incPC    <= '1';
                  AD_rd       <= '1';
                  tstep        <= "0011";

         --Execute 1
         ELSIF(step = "0011")THEN
                  CASE intInstr IS
                           -- LDA #, LDA mem, LDA mem+x, STA mem, STA mem+x
                           -- put regval 2 from ROM onto data bus
                           WHEN 0 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 12 | 13 =>
                                    AD_rd       <= '1';
                                    tstep        <= "0100";
                           -- COMA
                           WHEN 7 =>
                                    ALU_loadZ   <= '1';
                                    ALU_fSelect <= "100";
                                    tstep                       <= "0100";
                           -- CLEARX
                           -- perform clear x operation
                           -- put next instr on data bus from ROM
                           WHEN 14 =>
                                    PC_clearX   <= '1';
                                    AD_rd       <= '1';
                                    tstep        <= "0001";
                           -- INCX
                           -- perform inc x operation
                           -- put next instr on data bus from ROM
                           WHEN 15 =>
                                    PC_incX     <= '1';
                                    AD_rd       <= '1';
                                    tstep        <= "0001";
                           -- Error
                           WHEN OTHERS =>
                                    PC_clearPC  <= '1';
                                    PC_clearX   <= '1';
                                    AD_wt       <= '1';
```

```
                                      AD_rd       <= '1';
                                      tstep        <= "0000";
                        END CASE;

              --Execute 2
              ELSIF(step = "0100")THEN
                        CASE intInstr IS
                                -- LDA #
                                -- load regval 2 from data bus into A
                                -- enable Z flag
                                WHEN 0 =>
                                      PC_incPC    <= '1';
                                      ALU_loadA   <= '1';
                                      --ALU_loadZ   <= '1';
                                      tstep        <= "0000";
                                -- LDA mem, LDA mem+x, STA mem, STA mem+x
                                -- BRA, BEQ, BNE
                                -- load regval 2 into ORH
                                WHEN 1 | 2 | 3| 4 | 11 | 12 | 13 =>
                                      PC_incPC    <= '1';
                                      ORH_load    <= '1';
                                      AD_rd       <= '1';
                                      tstep        <= "0101";
                                -- COMA
                                -- put ~A onto data bus and load A
                                -- inc PC
                                WHEN 7 =>
                                      ALU_fSelect <= "100";
                                      ALU_loadZ   <= '1';
                                      ALU_loadA   <= '1';
                                      tstep                        <= "0000";
                                -- ANDA
                                -- load B, output A AND B, check Z
                                WHEN 5 =>
                                      ALU_fSelect <= "010";
                                      ALU_loadB   <= '1';
                                      ALU_loadZ   <= '1';
                                      tstep                        <= "0101";
                                -- ORA
                                -- load B, output A OR B, check Z
                                WHEN 6 =>
                                      ALU_fSelect <= "011";
                                      ALU_loadB   <= '1';
                                      ALU_loadZ   <= '1';
                                      tstep                        <= "0101";
                                -- ADDA
                                -- load B, output A + B, check Z
                                WHEN 8 =>
                                      ALU_fSelect <= "000";
                                      ALU_loadB   <= '1';
                                      ALU_loadZ   <= '1';
                                      tstep                        <= "0101";
                                -- SUBA, CMPA
                                -- load B, output A - B, check Z
                                -- might be a problem if CMPA writes to data bus?
                                -- does it anyway
                                WHEN 9 | 10 =>
                                      ALU_fSelect <= "001";
                                      ALU_loadB   <= '1';
                                      ALU_loadZ   <= '1';
                                      tstep                        <= "0101";
                                -- Error
                                WHEN OTHERS =>
                                      PC_clearPC  <= '1';
                                      PC_clearX   <= '1';
                                      AD_wt       <= '1';
                                      AD_rd       <= '1';
                                      tstep        <= "0000";
                        END CASE;

              --Execute 3
```

```
        ELSIF(step = "0101")THEN
                CASE intInstr IS
                        -- LDA mem, LDA mem+x, STA mem, STA mem,x
                        -- BRA, BEQ, BNE
                        -- ORH is now "in"
                        -- put regval 3 onto data bus
                        WHEN 1 | 2 | 3 | 4 | 11 | 12 | 13 =>
                                AD_rd       <= '1';
                                tstep        <= "0110";
                        -- ANDA
                        -- put A AND B on data bus and load A
                        -- inc PC
                        WHEN 5 =>
                                ALU_fSelect <= "010";
                                ALU_loadA   <= '1';
                                ALU_loadZ   <= '1';
                                PC_incPC           <= '1';
                                tstep                       <= "0000";
                        -- ORA
                        -- put A OR B on data bus and load A
                        -- inc PC
                        WHEN 6 =>
                                ALU_fSelect <= "011";
                                ALU_loadA   <= '1';
                                ALU_loadZ   <= '1';
                                PC_incPC           <= '1';
                                tstep                       <= "0000";
                        -- ADDA
                        -- put A + B on data bus and load A
                        -- inc PC
                        WHEN 8 =>
                                ALU_fSelect <= "000";
                                ALU_loadA   <= '1';
                                ALU_loadZ   <= '1';
                                PC_incPC           <= '1';
                                tstep                       <= "0000";
                        -- SUBA
                        -- put A - B on data bus and load A
                        -- inc PC
                        WHEN 9 =>
                                ALU_fSelect <= "001";
                                ALU_loadA   <= '1';
                                ALU_loadZ   <= '1';
                                PC_incPC           <= '1';
                                tstep                       <= "0000";
                        -- CMPA
                        -- keep outputting A-B, might be a problem?
                        -- inc PC
                        WHEN 10 =>
                                ALU_fSelect <= "001";
                                ALU_loadZ   <= '1';
                                PC_incPC    <= '1';
                                tstep                       <= "0000";
                        -- Error
                        WHEN OTHERS =>
                                PC_clearPC  <= '1';
                                PC_clearX   <= '1';
                                AD_wt       <= '1';
                                AD_rd       <= '1';
                                tstep        <= "0000";
                END CASE;

        --Execute 4
        ELSIF(step = "0110")THEN
                CASE intInstr IS
                        -- LDA mem, LDA mem+x, STA mem, STA mem+x
                        -- BRA, BEQ, BNE
                        -- load regval 3 from data bus into ORL
                        WHEN 1 | 2 | 3 | 4 | 11 | 12 | 13 =>
                                ORL_load    <= '1';
                                AD_rd       <= '1';
```

```
                                        tstep        <= "0111";
                              -- Error
                              WHEN OTHERS =>
                                        PC_clearPC  <= '1';
                                        PC_clearX   <= '1';
                                        AD_wt       <= '1';
                                        AD_rd       <= '1';
                                        tstep        <= "0000";
                    END CASE;

          --Execute 5
          ELSIF(step = "0111")THEN
                    CASE intInstr IS
                              -- LDA mem, LDA mem+x, STA mem, STA mem+x
                              -- BRA, BEQ, BNE
                              -- ORL is now "in"
                              WHEN 1 | 2 | 3 | 4 | 11 | 12 | 13 =>
                                        AD_rd       <= '1';
                                        tstep        <= "1000";
                              --Error
                              WHEN OTHERS =>
                                        PC_clearPC  <= '1';
                                        PC_clearX   <= '1';
                                        AD_wt       <= '1';
                                        AD_rd       <= '1';
                                        tstep        <= "0000";
                    END CASE;

          --Execute 6
          ELSIF(step = "1000")THEN
                    CASE intInstr IS
                              -- LDA mem
                              WHEN 1 =>
                                        PC_outSel   <= "10";
                                        AD_rd       <= '1';
                                        tstep        <= "1001";
                              -- LDA mem,x
                              WHEN 2 =>
                                        PC_outSel   <= "01";
                                        AD_rd       <= '1';
                                        tstep        <= "1001";
                              -- STA mem
                              WHEN 3 =>
                                        PC_outSel   <= "10";
                                        AD_wt       <= '1';
                                        tstep        <= "1001";
                              -- STA mem,x
                              WHEN 4 =>
                                        PC_outSel   <= "01";
                                        AD_wt       <= '1';
                                        tstep        <= "1001";
                              -- BRA
                              WHEN 11 =>
                                        PC_loadPC          <= '1';
                                        tstep              <= "0000";
                              -- BEQ
                              WHEN 12 =>
                                        IF (PC_zflag = '1') THEN
                                                  PC_loadPC <= '1';
                                        ELSE
                                                  PC_incPC <= '1';
                                        END IF;
                                        tstep                         <= "0000";
                              -- BNE
                              WHEN 13 =>
                                        IF (PC_zflag = '0') THEN
                                                  PC_loadPC <= '1';
                                        ELSE
                                                  PC_incPC <= '1';
                                        END IF;
                                        tstep                         <= "0000";
```

```
                            -- Error
                            WHEN OTHERS =>
                                     PC_clearPC  <= '1';
                                     PC_clearX   <= '1';
                                     AD_wt       <= '1';
                                     AD_rd       <= '1';
                                     tstep        <= "0000";
                   END CASE;

          -- Memory
          ELSIF(step = "1001")THEN
                   CASE intInstr IS
                            -- LDA mem
                            WHEN 1 =>
                                     PC_incPC    <= '1';
                                     PC_outSel   <= "10";
                                     ALU_loadA   <= '1';
                                     ALU_loadZ   <= '1';
                                     AD_rd       <= '1';
                                     tstep        <= "0000";
                            -- LDA mem,x
                            WHEN 2 =>
                                     PC_incPC    <= '1';
                                     PC_outSel   <= "01";
                                     ALU_loadA   <= '1';
                                     ALU_loadZ   <= '1';
                                     AD_rd       <= '1';
                                     tstep        <= "0000";
                            -- STA mem, STA mem+x
                            WHEN 3 | 4 =>
                                     PC_incPC    <= '1';
                                     PC_outSel   <= "10";
                                     AD_wt       <= '1';
                                     AD_rd       <= '1';
                                     tstep        <= "0000";
                            -- Error
                            WHEN OTHERS =>
                                     PC_clearPC  <= '1';
                                     PC_clearX   <= '1';
                                     AD_wt       <= '1';
                                     AD_rd       <= '1';
                                     tstep        <= "0000";
                            END CASE;

          -- ERRORS
          ELSE
                   PC_clearPC  <= '1';
                   PC_clearX   <= '1';
                   AD_wt       <= '1';
                   AD_rd       <= '1';
                   tstep        <= "0000";
          END IF;
END PROCESS;

END BEHAVIORAL;
```

A.3 ROM

```
--------------------------------------------------------------------------------------------
-- Lab 5
-- ROM
-- Steve Comer
-- Edward Hazeldine
-- Michial Stikkel
-- Updated 29 Oct 2012
--       Saves the Values in ROM
--------------------------------------------------------------------------------------------

-- Import the necessary libraries.
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

---------------------------------------------------------------------------------------------

ENTITY ROM IS

PORT(
         clk     : IN STD_LOGIC;
         address : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         dataOut : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);

END ROM;

ARCHITECTURE Behavioral OF ROM IS

         TYPE tROM IS ARRAY (0 TO 95) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
         CONSTANT cROM : tROM := (

                   0  => "0000",
                   1  => "0001",
                   2  => "0011",
                   3  => "0110",
                   4  => "0000",
                   5  => "1000",
                   6  => "0010",
                   7  => "0011",
                   8  => "0110",
                   9  => "0000",
                   10 => "0110",
                   11 => "0100",
                   12 => "0101",
                   13 => "1101",
                   14 => "0011",
                   15 => "0110",
                   16 => "0000",
                   17 => "1000",
                   18 => "0011",
                   19 => "0111",
                   20 => "0011",
                   21 => "0110",
                   22 => "0000",
                   23 => "1000",
                   24 => "0010",
                   25 => "1010",
                   26 => "1111",
                   27 => "1101",
                   28 => "0001",
                   29 => "0100",
                   30 => "0011",
                   31 => "0110",
------------------------------------
                   32 => "0000",
                   33 => "1110",
                   34 => "0010",
                   35 => "0001",
                   36 => "0100",
                   37 => "0100",
                   38 => "1000",
                   39 => "0000",
                   40 => "1111",
                   41 => "1010",
                   42 => "1110",
                   43 => "1101",
                   44 => "0010",
                   45 => "0010",
                   46 => "0001",
                   47 => "0101",
                   48 => "1110",
                   49 => "0011",
                   50 => "1000",
                   51 => "0011",
                   52 => "0001",
                   53 => "0101",
                   54 => "1111",
                   55 => "0011",
                   56 => "1000",
                   57 => "0110",
                   58 => "0000",
                   59 => "1000",
                   60 => "0011",
                   61 => "1000",
                   62 => "1000",
                   63 => "1001",
------------------------------------
                   64 => "1000",
                   65 => "0011",
                   66 => "1000",
                   67 => "1001",
                   68 => "0000",
                   69 => "1011",
                   70 => "0011",
                   71 => "1000",
                   72 => "1101",
                   73 => "0000",
                   74 => "0101",
                   75 => "0011",
                   76 => "1000",
                   77 => "1110",
                   78 => "0000",
                   79 => "0110",
                   80 => "0011",
                   81 => "1000",
                   82 => "1111",
                   83 => "1011",
                   84 => "1000",
                   85 => "0000",
                   86 => "1100",
                   87 => "0000",
                   88 => "0000",
                   89 => "0000",
                   90 => "1110",
                   91 => "0011",
                   92 => "0110",
                   93 => "0000",
                   94 => "1001",
                   95 => "0000");

BEGIN

         Read_Process : PROCESS(clk)
         BEGIN
                   IF rising_edge(clk) THEN
                            dataOut <= cROM(conv_integer(address));
                   END IF;
         END PROCESS Read_Process;


END Behavioral;
```

A.4 Address Decoder

```
-- Lab 5
-- AddressDecoder                                                                                    --
-- Steve Comer
-- Edward Hazeldine
-- Michael Stikkel                                                                                   --
-- Updated 29 Oct 2012                                                                               --
-- Allows the external devices to Read/Write --
-- Lab 6
-- AddressDecoder                                                                                           --
-- Steve Comer
-- Updated 14 Nov 2012                                                                                      --
-- Allows the external devices to Read/Write --

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

ENTITY AddressDecoder IS
PORT(
         add        : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         rd, wt     : IN STD_LOGIC;
         clk        : IN STD_LOGIC;
         enableROM  : OUT STD_LOGIC;
         enablePORT : OUT STD_LOGIC;
         enableALU  : OUT STD_LOGIC;
         enableRAM  : OUT STD_LOGIC
);
END AddressDecoder;

ARCHITECTURE Behavioral OF AddressDecoder IS

BEGIN

PROCESS(clk)
--Controls the Tri-State Buffers

VARIABLE enALU_var  : STD_LOGIC;
VARIABLE enPORT_var : STD_LOGIC;
VARIABLE enROM_var  : STD_LOGIC;
VARIABLE enRAM_var  : STD_LOGIC;
VARIABLE intAddr    : INTEGER;

BEGIN

IF(rising_edge (clk))THEN
         enALU_var  := '0';
         enPORT_var := '0';
         enROM_var  := '0';
         enRAM_var  := '0';

         intAddr := conv_integer(add);

                  IF(wt = '0' AND rd = '0')THEN
                           enALU_var  := '1';
                  ELSIF(wt = '0' AND rd = '1')THEN
                           IF(intAddr <= 31)THEN
                                    enROM_var  := '1';
                           ELSIF(intAddr >= 128 AND intAddr <= 255) THEN
                              enRAM_var  := '1';
                           END IF;
                  ELSIF(wt = '1' AND rd = '0')THEN
                           IF(intAddr = 96)THEN
                                    enPORT_var := '1';
                           END IF;
                  END IF;

         enableALU  <= enALU_var;
         enablePORT <= enPORT_var;
         enableROM  <= enROM_var;
         enableRAM  <= enRAM_var;
END IF;

END PROCESS;

END Behavioral;
```

A.5 PORTLED

```
--------------------------------------------------------------------------------------------
-- Lab 5
-- PORTLED
-- Steve Comer
-- Edward Hazeldine
-- Michial Stikkel
-- Updated 29 Oct 2012

--------------------------------------------------------------------------------------------

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

ENTITY PORTLED IS

PORT(
           enable    : IN  STD_LOGIC;
           input     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);           -- The input value for the
register
           output             : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)            -- The output, for
display
);

END PORTLED;

ARCHITECTURE BEHAVIORAL OF PORTLED IS
BEGIN

-- This is the synchronous load command. When the clock is high, then the value
--                 of the register is updated
PROCESS (enable)
BEGIN
         IF (enable = '1') THEN                -- If the clock is on its rising edge
                  output <= input;                      --                 then store the value
of A into B
         END IF;
END PROCESS;

END BEHAVIORAL;
```

A.6 TSB

```
-- Lab 5 --
-- TSB --
-- Steve Comer --
-- Edward Hazeldine --
-- Michael Stikkel --
-- Updated 29 Oct 2012 --
-- Protects the BUS from unexpected writes.

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


ENTITY TSB IS
PORT(
         TSBenable:         IN STD_LOGIC;
         TSBDataIn:         IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         TSBDataOut:        OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END TSB;
```

```
ARCHITECTURE Behavioral OF TSB IS

BEGIN

PROCESS(TSBDataIn, TSBenable)
--display input switches
BEGIN
         IF TSBenable = '1' THEN
                  TSBDataOut <= TSBDataIn;
         ELSE
                  TSBDataOut <= "ZZZZ";
         END IF;
END PROCESS;

END Behavioral;
```

A.7 PC

```
-- Lab 3
-- ALU
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012

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

ENTITY pc IS

PORT(input         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);             -- input value
           clearPC : IN  STD_LOGIC;
           incPC            : IN  STD_LOGIC;
           loadPC           : IN  STD_LOGIC;
           clearX           : IN  STD_LOGIC;
           incX             : IN  STD_LOGIC;
           outSel           : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);             -- output select
value
           clk                       : IN STD_LOGIC;
           output           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  -- output value of the PC

END pc;

ARCHITECTURE Behavioral OF pc IS

         -- Create two registers for both the PC and X
         SIGNAL pcREG : STD_LOGIC_VECTOR(7 DOWNTO 0) REGISTER;
         SIGNAL xREG : STD_LOGIC_VECTOR(7 DOWNTO 0) REGISTER;

BEGIN

         -- Change values on a high clock edge
         PROCESS(clk)
         BEGIN

                  IF (RISING_EDGE(clk)) THEN                      -- check fro rising edge of
clock

                           IF clearPC = '1' THEN                           -- update the PC,
priority is:
                                    pcREG <= "00000000";                           --
         clear, load, incrememnt
                           ELSIF loadPC = '1' THEN
                                    pcREG <= input;
                           ELSIF incPC = '1' THEN
                                    pcREG <= pcREG + '1';
                           END IF;

                           IF clearX = '1' THEN                                    -- update
```

```
the X register, priority is:
                                        xREG <= "00000000";                   --       clear,
increment
                              ELSIF incX = '1' THEN
                                        xREG <= xREG + '1';
                              END IF;

                    END IF;

          END PROCESS;

          -- Update the output
          PROCESS(outSel)
          BEGIN
                    CASE (outSel) IS
                              WHEN "00" =>
                                        output <= pcREG;
                              WHEN "01" =>
                                        output <= xREG + input;
                              WHEN "10" =>
                                        output <= input;
                              WHEN "11" =>
                                        output <= xREG;
                              WHEN OTHERS => NULL;                                              --
safe case, do nothing
                    END CASE;
          END PROCESS;

END Behavioral;
```

A.8 ALU

```
----------------------------------------------------------------------------------------------
-- Lab 3
-- ALU
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012
----------------------------------------------------------------------------------------------

-- Import the necessary libraries
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------------

-- Create the ALU entity which loads values and does calculations based on those values
ENTITY alu IS

PORT(A                    : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);             -- The A input, or
accumulator
            B                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);   -- The B input
            loadA            : IN  STD_LOGIC;


            loadB            : IN  STD_LOGIC;


            loadZ            : IN  STD_LOGIC;


            fSelect : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);            -- The function select input
is defined
```

```
           clk                         : IN  STD_LOGIC;

           zOut              : OUT STD_LOGIC := '0';
           output            : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)              -- The 4 bit result
from the desired

);

END alu;

----------------------------------------------------------------------------------------------

ARCHITECTURE Behavioral OF alu IS

         -- This is a simple four bit register which updates is value if a clock input is given
         --                  while load = '1'
         COMPONENT reg
                   PORT(A                      : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);             --
the register's input
                              clk                        : IN  STD_LOGIC;
                              load             : IN  STD_LOGIC;
                              B                          : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
                   );
         END COMPONENT;

         -- This is a one bit register with a clock and clear signal. If a clock input is given
         --                  the register will load the input. If the clear signal changes then
the register
         --                  will reset the register back to '0'
         COMPONENT bitreg IS
                   PORT(A                      : IN  STD_LOGIC;
                              clk                        : IN  STD_LOGIC;
                              clear            : IN  STD_LOGIC;
                              B                          : OUT STD_LOGIC
                   );
         END COMPONENT;

         -- Create wires for processes to communicate with other
         SIGNAL regA    : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- holds the value of registerA
         SIGNAL regB    : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- holds the value of registerB
         SIGNAL regZ        : STD_LOGIC;
         SIGNAL outFlag : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- holds the result of the ALU operation
         SIGNAL clear       : STD_LOGIC := '0';                                   -- used to
clear registerZ

----------------------------------------------------------------------------------------------

BEGIN

                   -- Create the three registers to store A, B, and Z
                   registerA : reg PORT MAP(A, clk, loadA, regA);
                   registerB : reg PORT MAP(B, clk, loadB, regB);
                   registerZ : bitreg PORT MAP(loadZ, clk, clear, regZ);

                   -- This process calculates the desired ALU operation using the stored values
in
                   --                  registerA and registerB
                   PROCESS(fSelect, regA, regB)
                   BEGIN
                            CASE fSelect IS
                                     WHEN "000" =>
                                              outFlag <= regA + regB;
                                     WHEN "001" =>
                                              outFlag <= regA - regB;
                                     WHEN "010" =>
                                              outFlag <= regA and regB;
                                     WHEN "011" =>
```

```
                                        outFlag <= regA or regB;
                                WHEN "100" =>
                                        outFlag <= not regA;
                                WHEN "101" =>
                                        outFlag <= regA;
                                WHEN "110" =>
                                        outFlag <= not regB;
                                WHEN "111" =>
                                        outFlag <= regB;
                                WHEN OTHERS =>
                                        outFlag <= "0000";
                        END CASE;
                END PROCESS;

                -- This process updates the calculated result to the output
                PROCESS (outFlag)
                BEGIN
                        output <= outFlag;                                      --
set the output to the ALU's result
                        IF (regZ = '1') THEN
                                IF outFlag = "0000" THEN                        --
                                        zout <= '1';
                                ELSE
                                        zout <= '0';
                                END IF;
                        END IF;
                END PROCESS;

                -- Whenever an operation is performed then the z flag does not need to be
updated anymore
                PROCESS (fSelect)
                BEGIN
                        IF regZ = '1' THEN                      -- If registerZ was set when a
function was performed then
                                clear <= not clear;             --                 clear the
value of registerZ after the function
                        END IF;                                                 --
otherwise disregard registerZ
                END PROCESS;


END Behavioral;
```

A.9 Bitreg

```
---------------------------------------------------------------------------------------------
-- Lab 3
-- reg1
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012
--       Implements a one bit synchronous register with a load bit and a clear input.
---------------------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

---------------------------------------------------------------------------------------------

-- Create and Entity which acts as a one bit register.
ENTITY bitreg IS

PORT(A                  : IN  STD_LOGIC;                                -- The input value
for the register
           clk                  : IN  STD_LOGIC;                                -- Clock
input used for synchrnous load
           clear                : IN  STD_LOGIC;                        -- Clears the value
in the register whenever this

           B                          : OUT STD_LOGIC := '0'       -- The output value of the
register
);
END bitreg;

----------------------------------------------------------------------------------------------


ARCHITECTURE Behavioral OF bitreg IS

         -- Declare a signal used to store what value that the clear input is.
         SIGNAL reset : STD_LOGIC := '0';

BEGIN

         -- This process activates whenever the clock or clear changes
         PROCESS (clk, clear)
         BEGIN

                  IF clear = not reset THEN             -- If the value of clear changed
                           B <= '0';                                             --
         then set the output to zero
                           reset <= clear;                                  --
         and update the new value of clear
                  ELSIF (clk = '1') THEN                         -- Otherwise, if the clock is
high
                           B <= A;                                                         --
                  END IF;

         END PROCESS;

END Behavioral;
```

A.10 reg

```
----------------------------------------------------------------------------------------------
-- Lab 3
-- reg
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012
--       Implements a four bit synchronous register with a load bit.
----------------------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------------

-- Create an Entity that acts as a four bit register.
ENTITY reg IS

-- Input and output for the Register
PORT(A                       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);            -- The input value
for the register
           clk                        : IN  STD_LOGIC;
           load              : IN  STD_LOGIC;

           B                          : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)             -- The
output, the value of the register
);

END reg;

----------------------------------------------------------------------------------------------
```

```
ARCHITECTURE Behavioral OF reg IS

BEGIN

         -- This is the synchronous load command. When the clock is high and load is set, then
the value
         --                  of the register is updated
         PROCESS (clk)
         BEGIN
                  IF (rising_edge(clk) and load = '1') THEN                -- If the clock is on
its rising edge and load is set,
                           B <= A;
                  END IF;
         END PROCESS;

END Behavioral;
```

A.11 Segdriver

```
----------------------------------------------------------------------------------------------
-- Lab 3
-- reg
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012
--       Implements a four bit synchronous register with a load bit.
----------------------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------------

-- Create an Entity that acts as a four bit register.
ENTITY reg IS

-- Input and output for the Register
PORT(A                     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);            -- The input value
for the register
           clk                      : IN  STD_LOGIC;
           load             : IN  STD_LOGIC;

           B                        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)             -- The
output, the value of the register
);

END reg;

----------------------------------------------------------------------------------------------


ARCHITECTURE Behavioral OF reg IS

BEGIN

         -- This is the synchronous load command. When the clock is high and load is set, then
the value
         --                  of the register is updated
         PROCESS (clk)
         BEGIN
                  IF (rising_edge(clk) and load = '1') THEN                -- If the clock is on
its rising edge and load is set,
                           B <= A;
                  END IF;
         END PROCESS;
END Behavioral;
```

A.12 ram

```
---------------------------------------------------------------------------------------------
-- Lab 6
-- RAM
-- Steve Comer
-- Updated 14 Nov 2012
--       128x4 RAM (128 addresses, 4-bit data)
---------------------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

---------------------------------------------------------------------------------------------

ENTITY ram IS

-- Input and output for the RAM
PORT(
         clk          : IN STD_LOGIC;
         readEnable   : IN STD_LOGIC;
         writeEnable  : IN STD_LOGIC;
         address      : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         dataIn              : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         dataOut      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
         --writeFlag    : OUT STD_LOGIC
);

END ram;

---------------------------------------------------------------------------------------------

ARCHITECTURE Behavioral OF ram IS

         TYPE tRam IS ARRAY (128 TO 255) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
         SIGNAL sRam : tRam;
         --
         --SIGNAL writeFlag_temp : STD_LOGIC;
         --

BEGIN

         Read_Write_Process : PROCESS(clk)
         --Read_Write_Process : PROCESS(address,dataIn,readEnable,writeEnable)

         BEGIN

                  --
                  --writeFlag <= '0';
                  --

                  IF rising_edge(clk) THEN
                           IF writeEnable = '1' THEN
                                    IF readEnable = '0' THEN
                                             --
                                             --writeFlag <= '1';
                                             --
                                             sRam(conv_integer(address)) <= dataIn;
                                             dataOut <= "0000";
                                    END IF;
                           ELSIF readEnable = '1' THEN
                                    IF writeEnable = '0' THEN
                                             --
```

```
                                        --writeFlag <= '0';
                                        --
                                        dataOut <= sRam(conv_integer(address));
                                END IF;
                        --
                        --ELSE
                        --      writeFlag <= '0';
                        --
                        END IF;
                END IF;

        END PROCESS Read_Write_Process;


END Behavioral;
```