

Experiment #5: Introduction to State Machine Design and ROM

Steven Comer
Edward Hazeldine
Michael Stikkel

Implementation Due: 13 Nov 2012
Report Due: 28 Nov 2012

Steven Comer

Edward Hazeldine

Michael Stikkel

Comer

Contents

1.0 Introduction	3
2.0 Method	3
2.1 Background Information	3
2.1.1 Register	3
2.1.2 Read Only Memory	3
2.1.3 Tri-State Buffer	3
2.1.4 Address Decoder	3
2.1.5 Program Counter	3
2.1.6 Arithmetic Logic Unit	4
2.1.7 Control Unit	4
2.1.8 Central Processing Unit	4
2.2 Procedure	4
2.2.1 Tri-state Buffer Design	4
2.2.2 Address Decoder Design	5
2.2.3 ROM Design	5
2.2.4 State Machine Design	5
2.2.5 CPU Design	6
2.2.6 Pin Assignments	7
2.3 Problems and Solutions	7
2.4 Materials and Tools Used	8
3.0 Results	8
3.1 Observations	8
3.2 Testing Procedure	8
3.2.1 Memory Values	8
3.2.2 Program Flow	9
3.2.3 High Level Output	10
3.3 Experiment Evaluation	10
3.4 Warning Explanations	10
3.4.1 Errors	10
3.4.2 Critical Warnings	10
3.4.3 Warnings	11
3.5 Quartus Flow Summary	11
4.0 Conclusion	13
A. Appendix A – Code	14
A.1 CPU	14
A.2 State Machine	18
A.3 ROM	26
A.4 Address Decoder	27
A.5 PORTLED	28
A.6 TSB	29
A.7 PC	30
A.8 ALU	31
A.9 Bitreg	33
A.10 reg	34
A.11 Segdriver	35

1.0 Introduction

The purpose of this experiment was to learn about State Machine design with Read Only Memory (ROM). The experiment made use of components constructed in previous experiments. These included ROM, Program Counter (PC) and Arithmetic Logic Unit (ALU). New components included an Address Decoder, Tri-state Buffers, Central Processing Unit (CPU). The ROM served as a simulated external device providing instructions for the state machine to execute. 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 {C, 9, 6, 3, 3} 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 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.

2.1.4 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.

2.1.5 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

Comer

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

2.1.6 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

2.1.7 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.

2.1.8 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 over a 4-bit data bus and an 8-bit address bus.

2.2 Procedure

See Appendix A for complete VHDL code listing.

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 impedance is outputted. See Figure 2.3 for diagram.

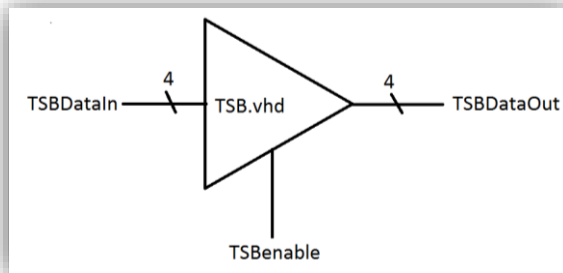


Figure 2.3 - Tri-state Buffer Diagram

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.

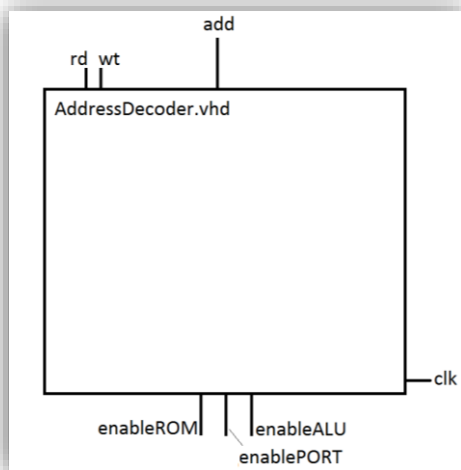


Figure 2.4 - Address Decoder Diagram

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 help 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 which when this is set

the state machine goes 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 2.5 shows the high-level state transition diagram used to construct the state machine.

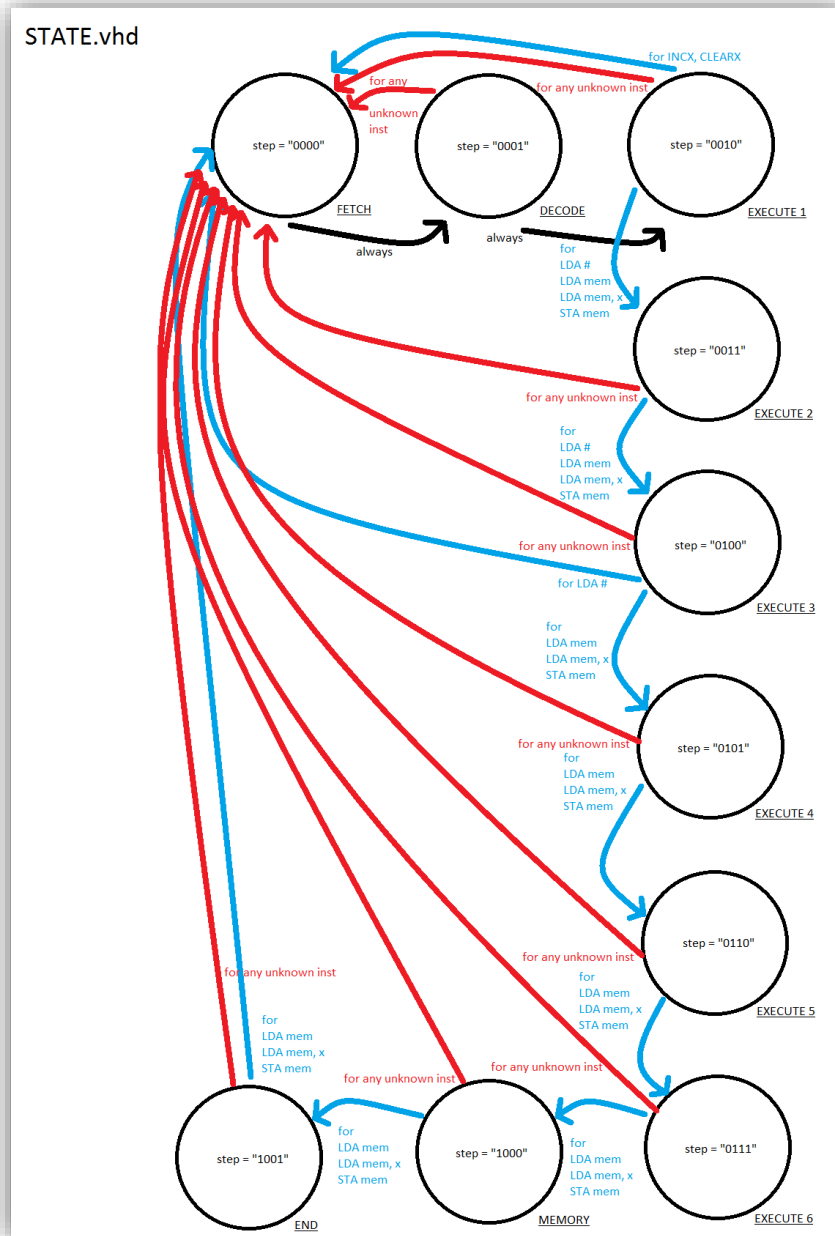


Figure 2.5 - State Transition Diagram

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 2.6 shows the diagram for the entire CPU including its components.

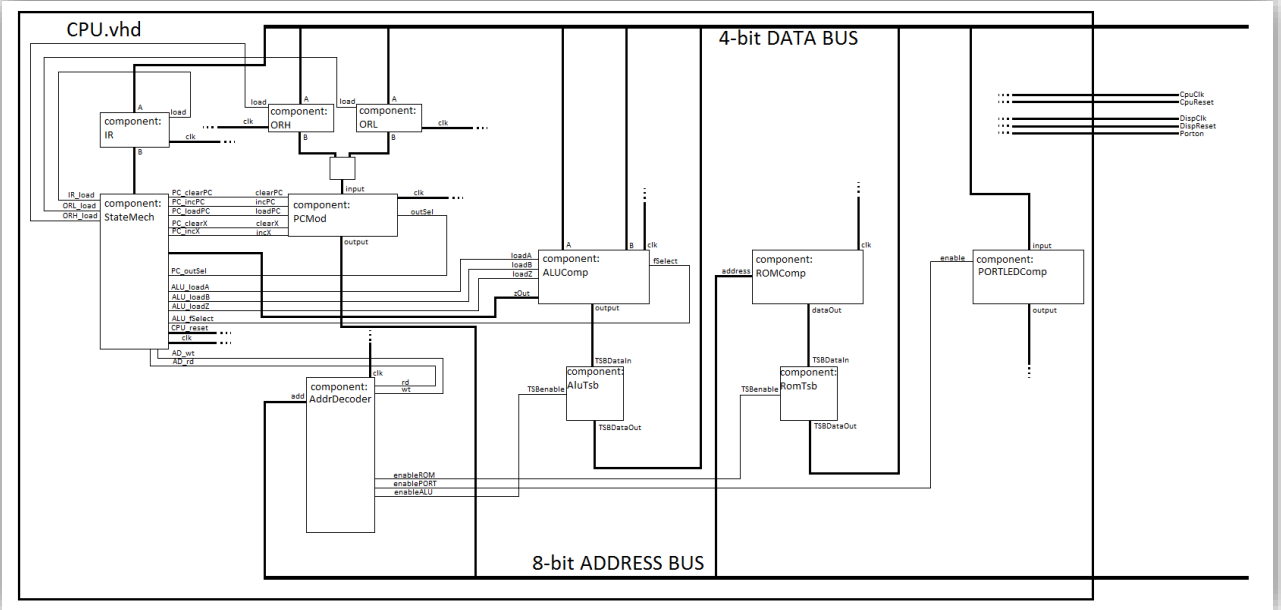


Figure 2.6 - CPU Diagram

2.2.6 Pin Assignments

The full list of pin assignments for the ALU is given in Appendix B. Pin assignments are explained in the order in which they appear in the appendix.

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 HEX4 and the value of the ALU in HEX3.

The values of the current step was displayed on HEX2 and the current instruction number was shown on HEX1. The PORTLED was displayed on HEX0.

The remaining three 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.

2.3 Problems and Solutions

Problem: In CPU.vhd, we lost control of control signal and bus updates.

Comer

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 convent 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 a succinctly coded 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.

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 component. It includes the value, associated instruction, and data type for each memory location. See Figure 3.1 for details.

Address	Value	Instruction	Type
0	0	1	Opcode
1	C	1	Immediate
2	3	2	Opcode
3	6	2	Address HI
4	0	2	Address LO
5	1	3	Opcode
6	1	3	Address HI

7	F	3	Address LO
8	3	4	Opcode
9	6	4	Address HI
10	0	4	Address LO
11	E	5	Opcode
12	F	6	Opcode
13	2	7	Opcode
14	1	7	Address HI
15	C	7	Address LO
16	3	8	Opcode
17	6	8	Address HI
18	0	8	Address LO
19	F	9	Opcode
20	2	10	Opcode
21	1	10	Address HI
22	C	10	Address LO
23	3	11	Opcode
24	6	11	Address HI
25	0	11	Address LO
26	3	12	Opcode
27	6	12	Address HI
28	0	12	Address LO
29	6	-	Data
30	3	-	Data
31	9	-	Data

Figure 3.1 - Memory Values

3.2.2 Program Flow

This portion describes the executed program at the instruction level. There are 12 recognizable instructions which execute, ending with a reset due to unknown instruction. Figure 3.2 lists the action, Accumulator register value, X register value, and PORTLED output for each instruction. Note that Acc refers to the Accumulator register, X refers to the X register and Output refers to the PORTLED output.

#	Instruction	Action	Acc	X	Output
1	LDA, C	Load Acc with immediate 0xC	C	0	0
2	STA @ 0x60	Display Acc on PORTLED	C	0	C
3	LDA @ 0x1F	Load Acc with value at address 0x1F	9	0	C
4	STA @ 0x60	Display Acc on PORTLED	9	0	9
5	CLR X	Clear register X	9	0	9
6	INC X	Increment register X	9	1	9
7	LDA @ 0x1C + X	Load Acc with value at address 0x1D	6	1	9
8	STA @ 0x60	Display Acc on PORTLED	6	1	6
9	INC X	Increment register X	6	2	6
10	LDA @ 0x1C + X	Load Acc with value at address 0x1E	3	2	6
11	STA @ 0x60	Display Acc on PORTLED	3	2	3
12	STA @ 0x60	Display Acc on PORTLED	3	2	3
13	Unknown Instruction	Reset	0	0	3

Figure 3.2 - Program Flow

Comer

3.2.3 High Level Output

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

1. Display C on PORTLED.
2. Display 9 on PORTLED.
3. Display 6 on PORTLED.
4. Display 3 on PORTLED.
5. Display 3 on PORTLED.
6. Reset and repeat.

3.3 Experiment Evaluation

The experiment was completely successful. All state transitions work as specified displaying the correct sequence of {C, 9, 6, 3, 3}. The reset condition also worked correctly. It handled the unknown instruction reached at the end of normal procedure through the program. It also 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.

Critical Warning 1:

Critical Warning (332012): Synopsys Design Constraints File file not found: 'Lab1_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/messages/messages/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 178 total logic elements, 177 total combinational functions, 47 dedicated logic registers, and 47 logic registers. See Figure 3.3 for the full flow summary.

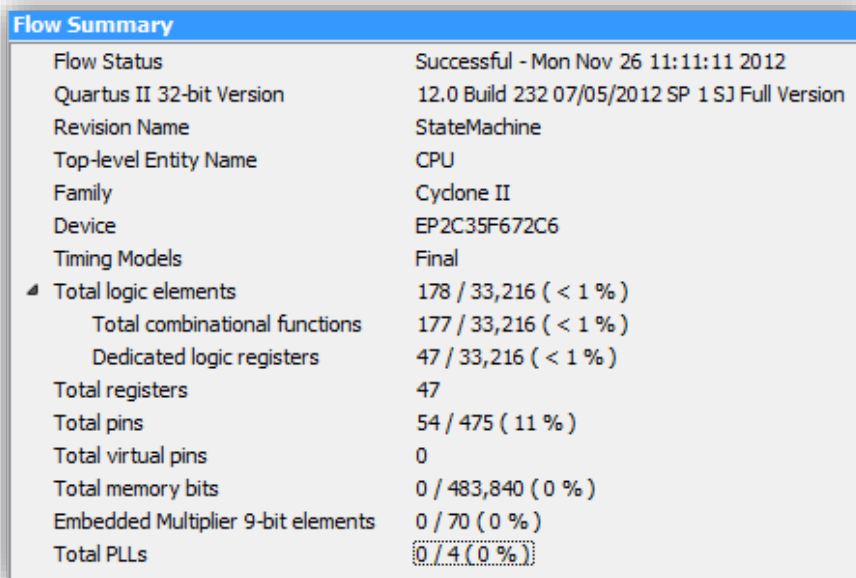
Comer

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 19 conditionals in State.vhd, 1 in PORTLED.vhd, 1 in reg.vhd/reg1.vhd (3 registers used), 1 in ROM.vhd, 1 in SegDriver.vhd, 1 in TSB.vhd (2 TSB used), 5 in AddressDecoder.vhd, 4 in alu.vhd, 5 in CPU.vhd, 6 in pc.vhd.

For purposes of a rough estimate/sanity check, $19+1+(1 \times 3)+1+1+(1 \times 2)+5+4+5+6 = 47$. 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 178 total logic elements, Quartus did not flood the implementation with logic. There is less than 1 % utilization. So, for the purposes of a sanity check, we can say that Quartus did not generate excess logic because less than 1 % is minimal.



Flow Summary	
Flow Status	Successful - Mon Nov 26 11:11:11 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	178 / 33,216 (< 1 %)
Total combinational functions	177 / 33,216 (< 1 %)
Dedicated logic registers	47 / 33,216 (< 1 %)
Total registers	47
Total pins	54 / 475 (11 %)
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 3.3 - Quartus Flow Summary

4.0 Conclusion

The objective of this experiment was to implement a simple CPU which executes a program in ROM.

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 5
-- State
-- Steve Comer
-- Edward Hazeldine
-- Michael Stikkel
-- Updated 29 Oct 2012
-- SU 397 CPU
-----

-- 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;

--Create top-level CPU entity.
ENTITY CPU IS
PORT(
    CpuClk,CpuReset                : IN STD_LOGIC; -- CPU inputs (clock and reset)
    DispClk,DispReset,Porton       : OUT STD_LOGIC; -- CPU 1-bit outputs (clock, reset,
port change)
    DispData,DispAdd1,DispAdd2,DispPort : OUT STD_LOGIC_VECTOR(6 DOWNTO 0); -- CPU segment
displays
    DispStep,DispIns,DispALU        : OUT STD_LOGIC_VECTOR(6
DOWNTO 0) -- CPU segment displays (more)
);
END CPU;

ARCHITECTURE BEHAVIORAL OF CPU IS

-- CPU's state component
COMPONENT 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
    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 COMPONENT;

COMPONENT reg IS

```

Comer

```

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 COMPONENT;

COMPONENT 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 COMPONENT;

COMPONENT 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 COMPONENT;

COMPONENT SegDriver IS
PORT(
      switches : IN STD_LOGIC_VECTOR(3 downto 0); -- 4-bit binary input
      segdispout : OUT STD_LOGIC_VECTOR(0 to 6) -- 7-bit output to
seg-display
);
END COMPONENT;

COMPONENT ROM IS
PORT(
      clk : IN STD_LOGIC;
      address : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8-bit address
select
      dataOut : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) -- 4-bit data out
);

```

Comer

```

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);          -- 8-bit address
input
    rd, wt      : IN STD_LOGIC;
    clk         : IN STD_LOGIC;
    enableROM   : OUT STD_LOGIC;
    enablePORT  : OUT STD_LOGIC;
    enableALU   : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT TSB IS
PORT(
    TSBenable   : IN STD_LOGIC;
    TSBDataIn   : IN STD_LOGIC_VECTOR(3 DOWNTO 0);          -- TSB data in  (4
bit)
    TSBDataOut  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)          -- TSB data out (4
bit)
);
END COMPONENT;

SIGNAL Update                                     :
STD_LOGIC;          -- Update Signal (flips on rising edge)
SIGNAL CStep
SIGNAL dataBus, AdataBus, RdataBus                : STD_LOGIC_VECTOR(3 DOWNTO 0); -- data
bus sized signals
SIGNAL addressBus                                : STD_LOGIC_VECTOR(7 DOWNTO 0); -- address bus
SIGNAL SIR_load, SORH_load, SORL_load             : STD_LOGIC; -- IR ORH and ORL load signals
SIGNAL SEPort, SEAlu, SERom                       : STD_LOGIC; -- Enable Signals for Port ALU and
ROM
SIGNAL Salu_Z                                     :
STD_LOGIC; -- Enable Z for ALU
SIGNAL SPC_clearPC, SPC_incPC                      : STD_LOGIC; -- PC
ClearPC and incPC signals
SIGNAL SPC_loadPC, SPC_incX, SPC_clearX           : STD_LOGIC; -- PC LoadPC, incX, and clearX
signal
SIGNAL SALU_loadA, SALU_loadB, SALU_loadZ : STD_LOGIC; -- ALU loadA, loadB, and loadZ signals
SIGNAL SAD_wt, SAD_rd                             :
STD_LOGIC; -- Address Decoder read and write signals
SIGNAL SPC_outSel                                 :
STD_LOGIC_VECTOR(1 DOWNTO 0); -- PC output select
SIGNAL SALU_fSelect                               :
STD_LOGIC_VECTOR(2 DOWNTO 0); -- ALU function select
SIGNAL Sinst, SaluOut, SRomOut                    : STD_LOGIC_VECTOR(3 DOWNTO
0); -- Inst. Reg <-> State machine and ALU and ROM outputs
SIGNAL SPCinput                                   :
STD_LOGIC_VECTOR(7 DOWNTO 0); -- PC inputs (from ORH and ORL)
SIGNAL portLED_data                              :
STD_LOGIC_VECTOR(3 DOWNTO 0); -- port LED input

SIGNAL TIR_load, TORH_load, TORL_load             : STD_LOGIC; -- next state values for
corresponding signals
SIGNAL TPC_clearPC, TPC_incPC                     : STD_LOGIC; -- next
state values for corresponding signals
SIGNAL TPC_loadPC, TPC_incX, TPC_clearX           : STD_LOGIC; -- next state values for
corresponding signals
SIGNAL TALU_loadA, TALU_loadB, TALU_loadZ : STD_LOGIC; -- next state values for corresponding
signals

```


Comer

```

SIGNAL TAD_wt, TAD_rd                                     :
STD_LOGIC; -- next state values for corresponding signals
SIGNAL TPC_outSel                                         :
STD_LOGIC_VECTOR(1 DOWNT0 0); -- next state values for corresponding signals
SIGNAL TALU_fSelect                                       :
STD_LOGIC_VECTOR(2 DOWNT0 0); -- next state values for corresponding signals

BEGIN

-- clock light process
PROCESS (CpuClk)
BEGIN
    DispClk <= not CpuClk;
END PROCESS;

-- reset light process
PROCESS(CpuReset)
BEGIN
    DispReset <= CpuReset;
END PROCESS;

-- port led output process
PROCESS(SEPort)
BEGIN
    Porton <= SEPort;
END PROCESS;

-- update signal flip 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;

-- update process (updates all signal values on update)
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;
    IF (SEAlu = '1' AND SERom = '0' AND SEPort = '0') THEN
        dataBus <= AdataBus;
    ELSIF (SEAlu = '0' AND SERom = '1' AND SEPort = '0') THEN
        dataBus <= RdataBus;
    ELSIF (SEAlu = '0' AND SERom = '1' AND SEPort = '1') THEN
        dataBus <= SALuOut;
    ELSIF (SEAlu = '1' AND SERom = '0' AND SEPort = '1') THEN
        dataBus <= SALuOut;
    ELSE
        dataBus <= "ZZZZ";
    END IF;
END PROCESS;

-- Components (appropriately named)

```

Comer

```
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);

StateMech                        : State PORT
MAP(Sinst,CpuClk,CpuReset,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);

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);

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);
StepSEG                          : SegDriver PORT MAP(CStep, DispStep);
InsSEG                           : SegDriver PORT MAP(Sinst,DispIns);
ALUSEG                           : SegDriver PORT MAP(SAluOut,DispALU);

END BEHAVIORAL;
```

A.2 State Machine

```
-- Lab 5
-- State
-- Steve Comer
-- Edward Hazeldine
-- Michael Stikkel
-- Updated 29 Oct 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
    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;
```

Comer

```

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)
-- Fetch, Decode, Execute, Memory
BEGIN
    --Fetch
    IF(step = "0000")THEN
        IR_load <= '1';
        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 <= "111";
        AD_wt <= '0';
        AD_rd <= '1';
        tstep <= "0001";

    --Decode
    ELSIF(step = "0001")THEN
        IR_load <= '0';
        PC_incPC <= '1';
        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 <= '1';
        tstep <= "0010";
--Excute 1
ELSIF(step = "0010")THEN
        IR_load <= '0';
        PC_incPC <= '0';
        PC_loadPC <= '0';
        PC_outSel <= "00";
        ORL_load <= '0';
        ORH_load <= '0';
        ALU_loadA <= '0';
        ALU_loadB <= '0';
        ALU_loadZ <= '0';
        CASE instruction IS
            --INCX
            WHEN "1111" =>
                PC_incX <= '1';
                PC_clearPC <= '0';
                PC_clearX <= '0';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";
                tstep <= "0000";
            --CLEARX
            WHEN "1110" =>
                PC_incX <= '0';
                PC_clearPC <= '0';
                PC_clearX <= '1';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";
                tstep <= "0000";
            -- LDA #
            WHEN "0000" =>
                PC_incX <= '0';
                PC_clearPC <= '0';
                PC_clearX <= '0';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";
                tstep <= "0011";
            -- LDA mem
            WHEN "0001" =>
                PC_incX <= '0';
                PC_clearPC <= '0';
                PC_clearX <= '0';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";
                tstep <= "0011";
            -- LDA mem,x
            WHEN "0010" =>
                PC_incX <= '0';
                PC_clearPC <= '0';
                PC_clearX <= '0';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";
                tstep <= "0011";
            -- STA mem
            WHEN "0011" =>
                PC_incX <= '0';
                PC_clearPC <= '0';
                PC_clearX <= '0';
                AD_wt <= '0';
                AD_rd <= '1';
                ALU_fSelect <= "101";

```

```

                                tstep      <= "0011";
--Error
WHEN OTHERS =>
    PC_incX      <= '0';
    PC_clearPC   <= '1';
    PC_clearX    <= '1';
    AD_wt        <= '1';
    AD_rd        <= '1';
    ALU_fSelect  <= "111";
    tstep        <= "0000";

END CASE;
--Excute 2
ELSIF(step = "0011")THEN
    IR_load      <= '0';
    PC_incX      <= '0';
    PC_loadPC    <= '0';
    PC_outSel    <= "00";
    ORL_load     <= '0';
    ALU_loadB    <= '0';
CASE instruction IS
-- LDA #
WHEN "0000" =>
    PC_incPC     <= '1';
    PC_clearPC   <= '0';
    PC_clearX    <= '0';
    ORH_load     <= '0';
    ALU_loadA    <= '1';
    ALU_loadZ    <= '1';
    AD_wt        <= '0';
    AD_rd        <= '0';
    ALU_fSelect  <= "101";
    tstep        <= "0100";
-- LDA mem
WHEN "0001" =>
    PC_incPC     <= '1';
    PC_clearPC   <= '0';
    PC_clearX    <= '0';
    ORH_load     <= '1';
    ALU_loadA    <= '0';
    ALU_loadZ    <= '0';
    AD_wt        <= '0';
    AD_rd        <= '1';
    ALU_fSelect  <= "101";
    tstep        <= "0100";
-- LDA mem,x
WHEN "0010" =>
    PC_incPC     <= '1';
    PC_clearPC   <= '0';
    PC_clearX    <= '0';
    ORH_load     <= '1';
    ALU_loadA    <= '0';
    ALU_loadZ    <= '0';
    AD_wt        <= '0';
    AD_rd        <= '1';
    ALU_fSelect  <= "101";
    tstep        <= "0100";
-- STA mem
WHEN "0011" =>
    PC_incPC     <= '1';
    PC_clearPC   <= '0';
    PC_clearX    <= '0';
    ORH_load     <= '1';
    ALU_loadA    <= '0';
    ALU_loadZ    <= '0';
    AD_wt        <= '0';
    AD_rd        <= '1';
    ALU_fSelect  <= "101";
    tstep        <= "0100";
--Error
WHEN OTHERS =>
    PC_incPC     <= '0';

```

```

PC_clearPC <= '1';
PC_clearX  <= '1';
ORH_load   <= '0';
ALU_loadA  <= '0';
ALU_loadZ  <= '0';
AD_wt      <= '1';
AD_rd      <= '1';
ALU_fSelect <= "111";
tstep      <= "0000";

END CASE;
--Excute 3
ELSIF(step = "0100")THEN
  IR_load    <= '0';
  PC_incPC   <= '0';
  PC_incX    <= '0';
  PC_loadPC  <= '0';
  PC_outSel  <= "00";
  ORH_load   <= '0';
  ORL_load   <= '0';
  ALU_loadA  <= '0';
  ALU_loadB  <= '0';
  ALU_loadZ  <= '0';
  CASE instruction IS
    -- LDA #
    WHEN "0000" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0000";
    -- LDA mem
    WHEN "0001" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0101";
    -- LDA mem,x
    WHEN "0010" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0101";
    -- STA mem
    WHEN "0011" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0101";
    --Error
    WHEN OTHERS =>
      PC_clearPC <= '1';
      PC_clearX  <= '1';
      AD_wt      <= '1';
      AD_rd      <= '1';
      ALU_fSelect <= "111";
      tstep      <= "0000";

  END CASE;
--Excute 4
ELSIF(step = "0101")THEN
  IR_load    <= '0';
  PC_incPC   <= '0';
  PC_incX    <= '0';
  PC_loadPC  <= '0';
  PC_outSel  <= "00";

```

```

ORH_load    <= '0';
ALU_loadA   <= '0';
ALU_loadB   <= '0';
ALU_loadZ   <= '0';
CASE instruction IS
  -- LDA mem
  WHEN "0001" =>
    PC_clearPC <= '0';
    PC_clearX  <= '0';
    ORL_load   <= '1';
    AD_wt      <= '0';
    AD_rd      <= '1';
    ALU_fSelect <= "101";
    tstep      <= "0110";
  -- LDA mem,x
  WHEN "0010" =>
    PC_clearPC <= '0';
    PC_clearX  <= '0';
    ORL_load   <= '1';
    AD_wt      <= '0';
    AD_rd      <= '1';
    ALU_fSelect <= "101";
    tstep      <= "0110";
  -- STA mem
  WHEN "0011" =>
    PC_clearPC <= '0';
    PC_clearX  <= '0';
    ORL_load   <= '1';
    AD_wt      <= '0';
    AD_rd      <= '1';
    ALU_fSelect <= "101";
    tstep      <= "0110";
  --Error
  WHEN OTHERS =>
    PC_clearPC <= '1';
    PC_clearX  <= '1';
    ORL_load   <= '0';
    AD_wt      <= '1';
    AD_rd      <= '1';
    ALU_fSelect <= "111";
    tstep      <= "0000";
END CASE;
--Excute 5
ELSIF(step = "0110")THEN
  IR_load    <= '0';
  PC_incPC   <= '0';
  PC_incX    <= '0';
  PC_loadPC  <= '0';
  PC_outSel  <= "00";
  ORH_load   <= '0';
  ORL_load   <= '0';
  ALU_loadA  <= '0';
  ALU_loadB  <= '0';
  ALU_loadZ  <= '0';
  CASE instruction IS
    -- LDA mem
    WHEN "0001" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0111";
    -- LDA mem,x
    WHEN ("0010") =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';
      AD_rd      <= '1';
      ALU_fSelect <= "101";
      tstep      <= "0111";

```

```

-- STA mem
WHEN "0011" =>
    PC_clearPC <= '0';
    PC_clearX  <= '0';
    AD_wt      <= '0';
    AD_rd      <= '1';
    ALU_fSelect <= "101";
    tstep      <= "0111";

--Error
WHEN OTHERS =>
    PC_clearPC <= '1';
    PC_clearX  <= '1';
    AD_wt      <= '1';
    AD_rd      <= '1';
    ALU_fSelect <= "111";
    tstep      <= "0000";

END CASE;

--Excute 6
ELSIF(step = "0111")THEN
    IR_load <= '0';
    PC_incPC <= '0';
    PC_incX  <= '0';
    PC_loadPC <= '0';
    ORH_load <= '0';
    ORL_load <= '0';
    ALU_loadA <= '0';
    ALU_loadB <= '0';
    ALU_loadZ <= '0';
    CASE instruction IS
        -- LDA mem
        WHEN ("0001") =>
            PC_clearPC <= '0';
            PC_clearX  <= '0';
            PC_outSel   <= "10";
            AD_wt      <= '0';
            AD_rd      <= '1';
            ALU_fSelect <= "101";
            tstep      <= "1000";

        -- LDA mem,x
        WHEN ("0010") =>
            PC_clearPC <= '0';
            PC_clearX  <= '0';
            PC_outSel   <= "01";
            AD_wt      <= '0';
            AD_rd      <= '1';
            ALU_fSelect <= "101";
            tstep      <= "1000";

        -- STA mem
        WHEN "0011" =>
            PC_clearPC <= '0';
            PC_clearX  <= '0';
            PC_outSel   <= "10";
            AD_wt      <= '1';
            AD_rd      <= '0';
            ALU_fSelect <= "101";
            tstep      <= "1000";

        --Error
        WHEN OTHERS =>
            PC_clearPC <= '1';
            PC_clearX  <= '1';
            PC_outSel   <= "00";
            AD_wt      <= '1';
            AD_rd      <= '1';
            ALU_fSelect <= "111";
            tstep      <= "0000";

    END CASE;

-- Memory
ELSIF(step = "1000")THEN
    IR_load <= '0';
    PC_incX  <= '0';
    PC_loadPC <= '0';

```



```

ORH_load    <= '0';
ORL_load    <= '0';
ALU_loadB   <= '0';
CASE instruction IS
  -- LDA mem
  WHEN "0001" =>
    PC_incPC    <= '1';
    PC_clearPC  <= '0';
    PC_clearX   <= '0';
    PC_outSel   <= "10";
    ALU_loadA   <= '1';
    ALU_loadZ   <= '1';
    AD_wt       <= '0';
    AD_rd       <= '1';
    ALU_fSelect <= "101";
    tstep       <= "1001";
  -- LDA mem,x
  WHEN "0010" =>
    PC_incPC    <= '1';
    PC_clearPC  <= '0';
    PC_clearX   <= '0';
    PC_outSel   <= "01";
    ALU_loadA   <= '1';
    ALU_loadZ   <= '1';
    AD_wt       <= '0';
    AD_rd       <= '1';
    ALU_fSelect <= "101";
    tstep       <= "1001";
  -- STA mem
  WHEN "0011" =>
    PC_incPC    <= '1';
    PC_clearPC  <= '0';
    PC_clearX   <= '0';
    PC_outSel   <= "10";
    ALU_loadA   <= '0';
    ALU_loadZ   <= '0';
    AD_wt       <= '1';
    AD_rd       <= '1';
    ALU_fSelect <= "101";
    tstep       <= "1001";
  --Error
  WHEN OTHERS =>
    PC_incPC    <= '0';
    PC_clearPC  <= '1';
    PC_clearX   <= '1';
    PC_outSel   <= "00";
    ALU_loadA   <= '0';
    ALU_loadZ   <= '0';
    AD_wt       <= '1';
    AD_rd       <= '1';
    ALU_fSelect <= "111";
    tstep       <= "0000";
  END CASE;
-- End
ELSIF(step = "1001") THEN
  IR_load      <= '0';
  PC_incPC     <= '0';
  PC_incX      <= '0';
  PC_loadPC    <= '0';
  PC_outSel    <= "00";
  ORH_load     <= '0';
  ORL_load     <= '0';
  ALU_loadA    <= '0';
  ALU_loadB    <= '0';
  ALU_loadZ    <= '0';
  CASE instruction IS
    -- LDA mem
    WHEN "0001" =>
      PC_clearPC <= '0';
      PC_clearX  <= '0';
      AD_wt      <= '0';

```

```

        AD_rd      <= '1';
        ALU_fSelect <= "101";
        tstep      <= "0000";
    -- LDA mem,x
    WHEN "0010" =>
        PC_clearPC <= '0';
        PC_clearX  <= '0';
        AD_wt      <= '0';
        AD_rd      <= '1';
        ALU_fSelect <= "101";
        tstep      <= "0000";
    -- STA mem
    WHEN "0011" =>
        PC_clearPC <= '0';
        PC_clearX  <= '0';
        AD_wt      <= '0';
        AD_rd      <= '1';
        ALU_fSelect <= "101";
        tstep      <= "0000";
    --Error
    WHEN OTHERS =>
        PC_clearPC <= '1';
        PC_clearX  <= '1';
        AD_wt      <= '1';
        AD_rd      <= '1';
        ALU_fSelect <= "111";
        tstep      <= "0000";
    END CASE;
-- ERRORS
ELSE
    IR_load      <= '0';
    PC_incPC     <= '0';
    PC_clearPC   <= '1';
    PC_loadPC    <= '0';
    PC_clearX    <= '1';
    PC_incX      <= '0';
    PC_outSel    <= "00";
    ORL_load     <= '0';
    ORH_load     <= '0';
    ALU_loadA    <= '0';
    ALU_loadB    <= '0';
    ALU_loadZ    <= '0';
    ALU_fSelect  <= "111";
    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;
-----

```

Comer

```
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
    --setting up the values of the ROM
    TYPE tROM IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
    CONSTANT cRom : tROM := (
        0 => "0000",
        1 => "1100",
        2 => "0011",
        3 => "0110",
        4 => "0000",
        5 => "0001",
        6 => "0001",
        7 => "1111",
        8 => "0011",
        9 => "0110",
        10 => "0000",
        11 => "1110",
        12 => "1111",
        13 => "0010",
        14 => "0001",
        15 => "1100",
        16 => "0011",
        17 => "0110",
        18 => "0000",
        19 => "1111",
        20 => "0010",
        21 => "0001",
        22 => "1100",
        23 => "0011",
        24 => "0110",
        25 => "0000",
        26 => "0011",
        27 => "0110",
        28 => "0000",
        29 => "0110",
        30 => "0011",
        31 => "1001");

BEGIN

    Read_Process : PROCESS(clk)
        --Returning the value requested
        VARIABLE trunc_addr : STD_LOGIC_VECTOR(4 DOWNTO 0);
        BEGIN
            IF rising_edge(clk) THEN
                trunc_addr := address(4 DOWNTO 0);
                dataOut <= cROM(conv_integer(trunc_addr));
            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 --
```

Comer

```
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
);
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;

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

    IF(wt = '0' AND rd = '0')THEN
        enALU_var := '1';
    ELSIF(wt = '0' AND rd = '1')THEN
        IF(conv_integer(add) <= 31)THEN
            enROM_var := '1';
        END IF;
    ELSIF(wt = '1' AND rd = '0')THEN
        IF(conv_integer(add) = 96)THEN
            enPORT_var := '1';
        END IF;
    END IF;

    enableALU <= enALU_var;
    enablePORT <= enPORT_var;
    enableROM <= enROM_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;
```

Comer

```
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";                                --
increment
            ELSIF incX = '1' THEN
                xREG <= xREG + '1';
            END IF;

            END IF;

        END PROCESS;

        -- Update the output
        PROCESS(outSel)
        BEGIN
            CASE (outSel) IS

```

Comer

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

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

        clk
        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 its value if a clock input is given
    -- while load = '1'
    COMPONENT reg
        PORT(A
the register's input
            : 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;

    -- 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
    --
    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
function was performed then
                clear <= not clear;
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
for the register
        clk
input used for synchrnous load
        clear
in the register whenever this

        B
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';

```

Comer

```
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 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;

```