# Laboratory Manual and Supplementary Notes

# ECE 495: Computer Engineering Design Laboratory

Version 3

Jason Koonts
Dr. Edwin Hou

Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102

## Contents

# Experiment 1: Event Driven Circuit

## *Objectives*

The objective of this laboratory is to familiarize the student with the design techniques for event driven sequential circuits, and to introduce the student to applications involving EEPROMs.

## *Preparation*

- Review Chapter 2 of *Computer Systems Organization & Architecture*

## *Equipment Needed*

- One switch block
- Red, amber, and green LEDs
- One protoboard
- One 2816 EEPROM
- One 74374 octal D-type latch (or CMOS equivalent)
- Signal generator for clock

## *References*

- John D. Carpinelli, *Computer Systems Organization & Architecture*, Addison Wesley, 2001
- Morris Mano, *Computer Engineering Hardware Design*, Addison Wesley

## *Background*

Event driven sequential circuits differ from combinatorial circuits in that the outputs of the circuit depend not only on the present state of the inputs but also on the past history of the inputs. Thus a sequential circuit has memory. Memory is provided for each bit needed to define a state by using a bit storage device such as a flip-flop. Any type of flip-flop is suitable, but with some designs the J-K type may be better in the sense that the combinatorial logic required for the feedback path is usually (but not always) minimized by this choice. The combinatorial part of the circuit can be implemented using gates, MUXs or any method that is capable of providing the necessary feedback logic. In our case an EEPROM will be used. This will have the added benefit of familiarizing the student with the procedures of working with these very common (and presently inexpensive) read only digital memories. The storage device that will be used will be an octal D-type latch, and it will be seen that the entire design has chip count of two.

In this experiment, an event driven sequential circuit will be implemented. This type of circuit is free running in the sense that the output must respond to an input change in a very short time. Other names commonly used are non-pulse circuits or asynchronous circuits.

### **<u>A Simple Example</u>**

A simple example was chosen in order to acquaint the student with the type of design presented in this experiment. It does not necessarily correspond to anything practical, but was chosen for the simplicity needed to get the requisite familiarity. The state diagram of this system is shown in

Figure 1.1. Each state is defined by the two-bit WZ and each state produces an output defined by the three bits PQR. The inputs KL, needed to make the transitions between the states are indicated on the directed branches connecting the state balloons. It is implicit that when KL does not cause any transition that the system remains in the same state.

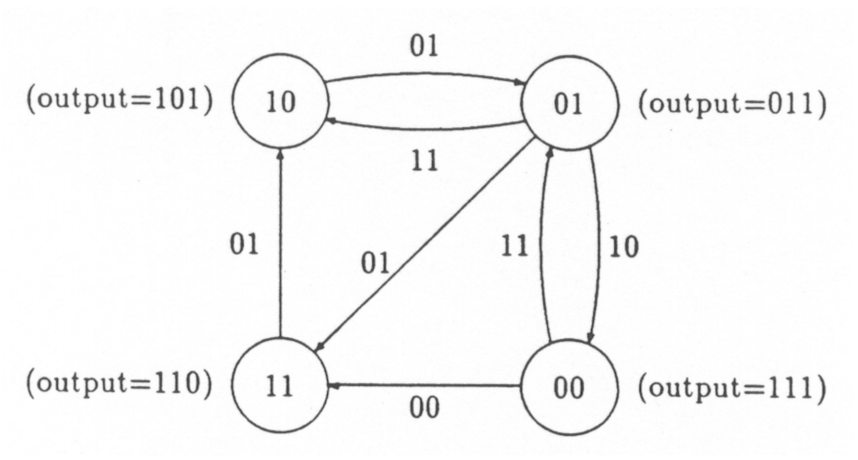


Figure 1.1: The state diagram of a simple event-driven sequential circuit

To gain familiarity with the use of EEPROMs, the two-chip design of the type shown in Figure 1.2 was chosen.

The EEPROM contains the combinatorial logic and the octal D-type latch contains the needed bit storage. Both devices contain more logic than is needed for this design, so expansion should be possible. It is decided to pass the inputs FG through the latch, so that the system clock will control all the events and facilitates the testing of the system. The output bits PQR are derived combinatorially from the state bit WZ. Since the EEPROM has a substantial amount of unused logic we can put it to good use by using three of the EEPROM output lines for the output bits PQR. We determine the bit storage pattern for the EEPROM by creating a state table as shown in Table 1.1. Note that the present state and the inputs supply the address to the EEPROM. The data outputs provide the next state and system outputs.

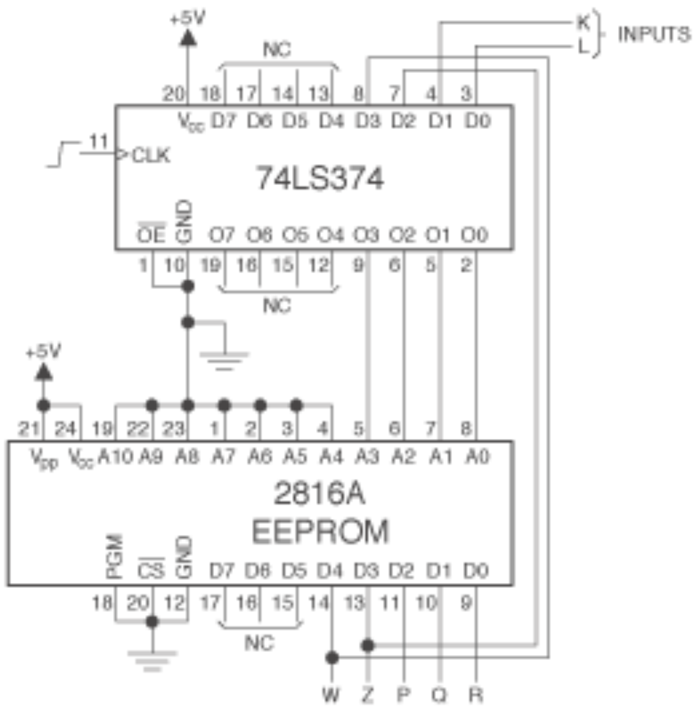


Figure 1.2: Hardware implementation for the simple event-driven sequential circuit of Figure 1.1

EPROMs or EEPROMs are useful for changeable designs. If you don't like your first design, change it and reprogram the ROM. Rewiring is not required if your basic design is sound. We are using an EEPROM because it does not require a 20 minutes erasure using a UV light device.

Before assembling a program, source code must be created with an editor or word processor (in ASCII mode). As an example, assume we are creating a source file for the state table in Table 1.1, named TEST.ASM. A partial listing of directives for using the assembler ASM68K.EXE is given below:

```
ORG 0
DC.B $1E, $07, $07, $0B      ; start assembling code from address 0
DC.B $0B, $1E, $07, $15      ; stores 4 bytes of date
DC.B $15, $0B, $15, $15
DC.B $1E, $15, $1E, $1E
END
```

Once the source file is edited it can be assembled using the assembler ASM68K.EXE. The assembler will produce a TEST.HEX file (S-records), which can then be used to program the EEPROM with the *Superpro Programmer* program.

| *Present State* | | | | *Next State* | | | | |
|---|---|---|---|---|---|---|---|---|
| *State* | | *Input* | | *State* | | *Output* | | |
| W | Z | F | G | W | Z | P | Q | R |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Table 1.1: The state table of the simple event-driven sequential circuit

## An Event Driven Annunciator System

The state diagram for the sequential circuit that we wish to design is shown in Figure 1.3. It is a two-alarm system that might be used in factory to signal that various levels of faults are occurring on the production line. One application may be in process control, where the fault could indicate a high pressure. A flashing amber light would indicate the first state of pressure change, a potential hazard. A large change in the system, shown by a flashing red light would indicate an emergency condition.

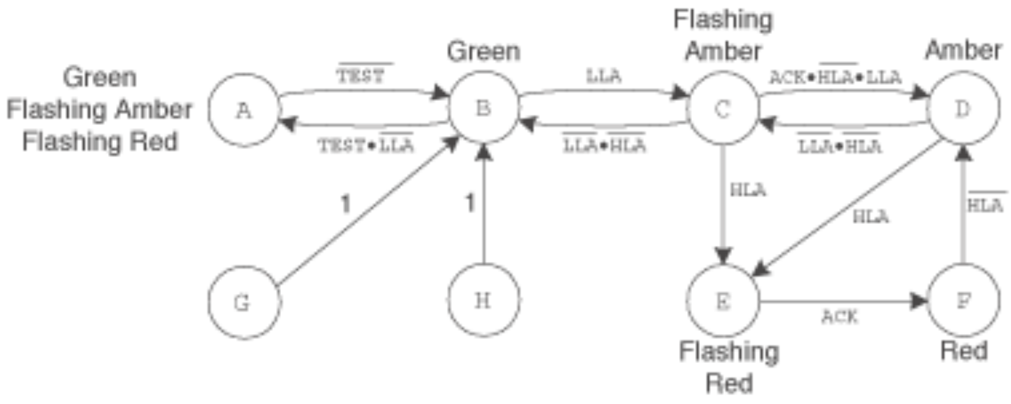


Figure 1.3: State diagram of the annunciator circuit

The annunciator has 6 proper states, so we shall need a 3-bit sequential circuit. Each state is described by the 3 bits UVW. For this example, states A through H have UVW = 000 to 111 respectively. The annunciator monitors the factory for the presence of a low-level alarm signal LLA and a high-level alarm signal HLA. The high-level alarm signal HLA cannot occur if the low-level alarm signal LLA has not occurred first. The circuit operates in the following manner:

1. With no fault signal present the system is stable, UVW equals 001 (B), and the GREEN light is on.
2. When the signal LLA is present, indicating a minor fault, UVW changes to 010 (C), a FLASHING-AMBER light comes on and the GREEN light goes off. If the default disappears (LLA changes to 0), the annunciator returns directly to the normal GREEN state with UVW = 001 (B).
3. If the system is in the minor fault (FLASHING-AMBER) state and an operator intervenes to clear the minor fault he pushes the acknowledge button which contains a momentary switch. The presence of the ACK signal of mere fraction of a second changes UVW to 011 (D) and the FLASHING-AMBER light becomes steady, telling supervisory personnel that someone is trying to clear the fault. If the minor fault is cleared (NOT LLA) the annunciator returns to the normal (GREEN) state by going to UVW = 010 (C) and if HLA is not present to UVW = 001 (B).
4. If the system is in the AMBER or FLASHING-AMBER state and the major fault signal (HLA) is received, the system signals a major fault by changing UVW to 100 (E) that is the FLASHING-RED state. Even if this signal is only momentary, this condition will be maintained indefinitely. The presence of the ACK signal for a mere fraction of a second changes UVW to 101 (F) and the flashing red light becomes steady telling supervisory personnel that someone is trying to clear the major fault. If the major fault is cleared (NOT HLA), the annunciator starts on its path to the normal state by going to UVW = 011 (D) and continuing from there if LLA is also zero.
5. A test pushbutton is included to check all the annunicator lights. On TEST, the GREEN, FLASHING-AMBER and FLASHING-RED lights should be on. The state is identified with UVW = 000 (A).
6. In the event that one of the unused states (UVW = 110 or 111) (G or H) is erroneously entered, the system returns to the start state, UVW = 001 (B).

## Prelab Assignment

Student should obtain their unique state sequences from the laboratory instructor. Each student should prepare a design similar to the one presented in this experiment that will realize the required state assignments for this design.

## Laboratory Assignments

Program the EEPROM to realize the design of the annunciator for the required state assignments. Use the logic analyzer or a logic probe to debug the design and verify its operation to the laboratory instructor.

# Experiment 2: Introduction to Altera DE2 Board

## *Objectives*

The objective of this experiment is to gain an understanding of the Altera DE2 Board and create a simple project using input switches and the output LEDs.

## *Preparation*

- Review the provided user manual and tutorials for the Altera DE2 Board

## *Equipment Needed*

- ALTERA DE2 Board

## *References*

- User manual for Altera DE2 Board

## *Background*

A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components also include memory elements, which may be simple flip flops or more complete blocks of memories

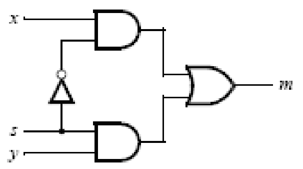
For this course we will be using a programming language called VHDL to program the FPGA found on the Cyclone 2 chip on your Altera DE2 Board. VHDL is a very robust language that can be implemented in a very high level format, using programming concepts such as for loops, if-then statements, and case assignments. VHDL also includes libraries that define adders, subtracters, counters, flip-flops, and more that can be instantiated to create a structural approach to your code.

The purpose of this exercise is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices. We will use the switches $SW_{17-0}$ on the DE2 board as inputs to the circuit. We will use light emitting diodes (LEDs) as outputs to your device.

The DE2 board provides 18 toggle switches, called $SW_{17-0}$, which can be used as inputs to a circuit. There are also 18 red LEDs, called $LEDR_{17-0}$, which can be used to display output values. The DE2 board has hardwired connections between its FPGA chip and the switches and lights. To use $SW_{17-0}$ and $LEDR_{17-0}$ it is necessary to include in your Quartus II project the correct pin assignments, which are given in the *DE2 User Manual*.

## Part 1

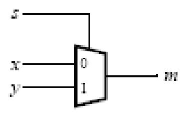
Figure 2.1 shows a sum-of-products circuit that implements a 2-to-1 multiplexer with a select input s. If s = 0 the multiplexer's output m is equal to the input x, and if s = 1 the output is equal to y. Part (b) of the figure gives a truth table for this multiplexer, and part (c) shows its circuit symbol.



a) Circuit

| $s$ | $m$ |
|---|---|
| 0 | $x$ |
| 1 | $y$ |

b) Truth table



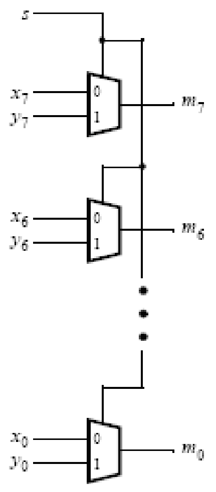
c) Symbol

Figure 2.1: A 2-to-1 multiplexer

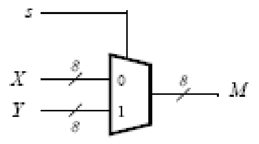
The multiplexer can be described by the following VHDL statement:

```
m <= (NOT (s) AND x) OR (s AND y);
```

You are to use the provided VHDL entity in appendix 2.1 that describes the circuit given in Figure 3a. This circuit has two eight-bit inputs, X and Y, and produces the eight-bit output M. If s = 0 then M = X, while if s = 1 then M = Y. We refer to this circuit as an eight-bit wide 2-to-1 multiplexer. It has the circuit symbol shown in Figure 3b, in which X, Y, and M are depicted as eight-bit wires.



a) Circuit

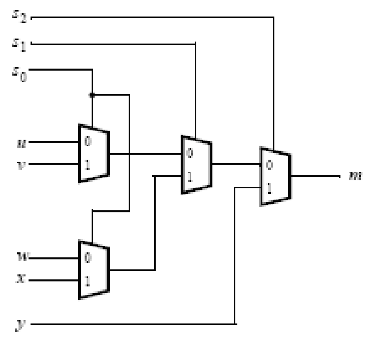


b) Symbol

Figure 2.2: An eight-bit wide 2-to-1 multiplexer

## Part 2

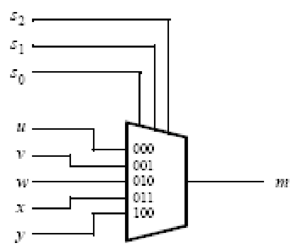
In Figure 2.1 we showed a 2-to-1 multiplexer that selects between the two inputs x and y. For this part consider a circuit in which the output m has to be selected from five inputs u, v, w, x, and y. Part (a) of Figure 2.3 shows how we can build the required 5-to-1 multiplexer by using four 2-to-1 multiplexers. The circuit uses a 3-bit select input s2s1s0 and implements the truth table shown in Figure 4b. A circuit symbol for this multiplexer is given in part (c) of the figure.



a) Circuit

| $s_2$ $s_1$ $s_0$ | $m$ |
|---|---|
| 0 0 0 | $u$ |
| 0 0 1 | $v$ |
| 0 1 0 | $w$ |
| 0 1 1 | $x$ |
| 1 0 0 | $y$ |
| 1 0 1 | $y$ |
| 1 1 0 | $y$ |
| 1 1 1 | $y$ |

b) Truth table



c) Symbol

Figure 2.3: A 5-to-1 multiplexer

Recall from Figure 2.2 that an eight-bit wide 2-to-1 multiplexer can be built by using eight instances of a 2-to-1 multiplexer. Figure 2.4 applies this concept to define a three-bit wide 5-to-1 multiplexer. It contains three instances of the circuit in Figure 4a. The VHDL code for this circuit is given in appendix 2.1.

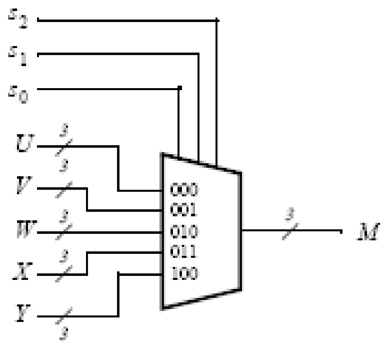


Figure 2.4: A three-bit wide 5-to-1 multiplexer

## *Prelab Assignments*

1. Review the manual and tutorials for the Altera DE2 Board.
2. Understand how to create and compile projects.
3. Review how pin assignments are done.
4. Review how to program the Altera DE2 Board.

## *Lab Assignments*

### Part 1

1. Create a new Quartus II project for your circuit.
2. Include the given VHDL code for the eight-bit wide 2-to-1 multiplexer in your project. Use switch $SW_{17}$ on the DE2 board as the s input, switches $SW_{7-0}$ as the X input and $SW_{15-8}$ as the Y input. Connect the SW switches to the red lights LEDR and connect the output M to the green lights $LEDG_{7-0}$.
3. Include in your project the required pin assignments for the DE2 board.
4. Compile the project.
5. Download the compiled circuit into the FPGA chip. Test the functionality of the eight-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.

### Part 2

1. Create a new Quartus II project for your circuit.
2. Include the given VHDL code for the three-bit wide 5-to-1 multiplexer. Connect its select inputs to switches $SW_{17-15}$, and use the remaining 15 switches $SW_{14-0}$ to provide the five 3-bit inputs U to Y. Connect the SW switches to the red lights LEDR and connect the output M to the green lights $LEDG_{2-0}$.
3. Include in your project the required pin assignments for the DE2 board. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the three-bit wide 5-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs U to Y can be properly selected as the output M.

# Experiment 3: T-Bird Turn Signal

## Objectives

The objective of this experiment is to familiarize the student with the ALTERA DE2 board and VDHL.

## Preparation

- Review Chapter 2 of *Computer Systems Organization & Architecture*

## Equipment Needed

- Altera DE2 Board

## References

- John D. Carpinelli, *Computer Systems Organization & Architecture*, Addison Wesley, 2001
- Morris Mano, Computer Engineering Hardware Design, Addison Wesley

## Background

In this experiment, we will implement the turn signal control of a Thunderbird (T-bird) car using VHDL and the Altera DE2 Board. The left and right turn signals of the T-bird each contain 3 lights. There are three binary inputs to the control, LEFT, RIGHT, and HAZARD. When LEFT = 1 or RIGHT = 1, the lights flash according to the patterns shown in Figure 3.1. Note that LEFT and RIGHT cannot be 1 at the same time and if the switch from LEFT =1 to RIGHT = 1 or RIGHT = 1 to LEFT = 1 occurs in middle of a flashing sequence, the control should starts the new sequence from the beginning. In addition, the HAZARD takes precedence over LEFT and RIGHT. When HAZARD = 1, all six lights will flash on and off.

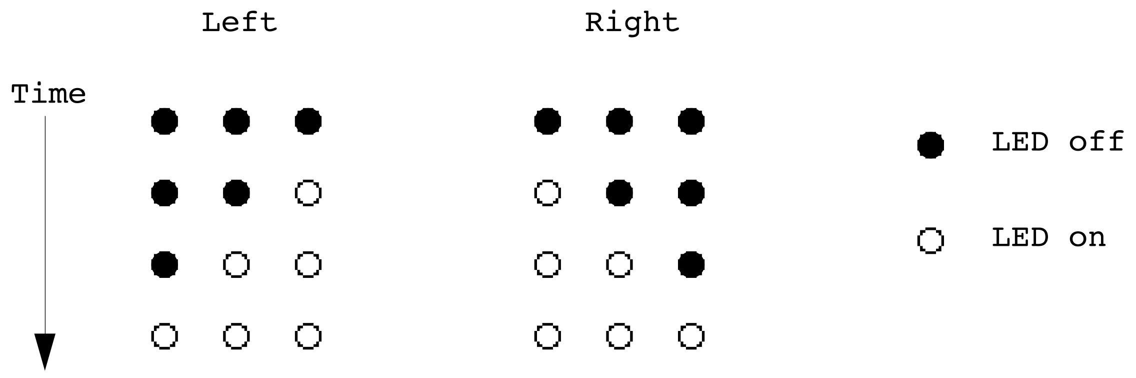


Figure 3.1: T-bird turn signal patterns

This T-bird turn signal control can be implemented with a Moore machine using eight states. Use the LEDs on the DE2 Board as the turn signal display. You can drive the circuit with the system clock on the DE2 Board which runs at about 50 MHz (You need to delay the transitions between states to achieve the proper display).

## *Prelab Assignment*

1. Prepare the state diagram and state transition table for the T-bird turn signal control.
2. Prepare a preliminary VHDL program for your design.
3. Simulate your circuit using the Quartus II Web Edition software.

## *Lab Assignment*

Program, test, simulate and debug your design until it works.

# Experiment 4: Clocks and Timers Using LCD Display

## *Objectives*

The objective of this experiment is to design and implement a few projects that will take advantage of the on-board 50MHz clock. This experiment will be useful in gaining an understanding of how to manipulate the clock for future experiments. In addition, the built-in LCD display will be used for the outputs of the experiment.

## *Preparation*

- Review Chapter 2 of *Computer Systems Organization & Architecture*

## *Equipment Needed*

- ALTERA DE2 Board

## *References*

- John D. Carpinelli, Computer Systems Organization & Architecture, Addison Wesley, 2001
- Morris Mano, Computer Engineering Hardware Design, Addison Wesley
- Altera DE2 Board user manual

## *Background*

In this experiment, you will design a BCD counter, a clock, and a reflex stop-watch to be output on the LCD display of the Altera DE2 Board. Code is given in appendix 4.1 of this document that will output the message "HELLO WORLD" to the LCD display. Use this code and the user manual to understand how to communicate with the LCD display.

### **Part 1**

Implement a 3-digit BCD counter. Display the contents of the counter on the 7-segment displays, *HEX2−0*. Derive a control signal, from the 50-MHz clock signal provided on the Altera DE2 board, to increment the contents of the counter at one-second intervals. Use the pushbutton switch *KEY*0 to reset the counter to 0.

### **Part 2**

Design and implement a circuit on the DE2 board that acts as a time-of-day clock. It should display the hour (from 0 to 23) on the 7-segment displays $HEX_{7-6}$, the minute (from 0 to 60) on $HEX_{5-4}$ and the second (from 0 to 60) on $HEX_{3-2}$. Use the switches $SW_{15-0}$ to preset the hour and minute parts of the time displayed by the clock.

### **Part 3**

Design and implement on the DE2 board a reaction-timer circuit. The circuit is to operate as follows:

1. The circuit is reset by pressing the pushbutton switch $KEY_0$.
2. After an elapsed time, the red light labeled $LEDR_0$ turns on and a four-digit BCD counter starts counting in intervals of milliseconds. The amount of time in seconds from when the circuit is reset until $LEDR_0$ is turned on is set by switches $SW_{7-0}$.
3. A person whose reflexes are being tested must press and hold pushbutton $KEY_3$ as quickly as possible to turn the LED off and freeze the counter in its present state. The count which shows the reaction time will be displayed on the LCD display.

(Hint: Implement this circuit using the 7-segment displays to output the value of the counter to verify functionality. Then modify the code to output to the LCD.)

## Prelab Assignment

1. Using the provided code, understand how to control and communicate with the LCD display.
2. Develop an algorithm for turning the 50MHz clock into a 1Hz clock to be used for the experiments.

## Lab Assignment

Develop VHDL code to implement the designs specified in parts 1, 2, and 3. Be sure to create a separate project for each part and to simulate your circuit to verify functionality. For final demonstration, you should be able to send the BCD counter, the programmable clock, and the reflex stop-watch to the LCD display.

# Experiment 5: Using VHDL Components

## *Objectives*

The objective of this experiment is to familiarize yourself with the way to create and instantiate components into your design. This design implementation stresses a structural approach.

## *Preparation*

- Review Chapter 2 of *Computer Systems Organization & Architecture*

## *Equipment Needed*

- Altera DE2 Board

## *References*

- John D. Carpinelli, *Computer Systems Organization & Architecture*, Addison Wesley, 2001
- Morris Mano, *Computer Engineering Hardware Design*, Addison Wesley

## *Background*

This experiment stresses the use of components in the design of VHDL entities. Components are very useful in a structural approach and can be used to model many simple functions such as flip flops, logic gates, counters, etc. They can also be as complex as ALUs and microsequencers.

### **Part 1**

Figure 5.1 shows a 7-segment decoder module that has the three-bit input c2c1c0. This decoder produces seven outputs that are used to display a character on a 7-segment display. Table 5.1 lists the characters that should be displayed for each valuation of c2c1c0.

The seven segments in the display are identified by the indices 0 to 6 shown in the figure. Each segment is illuminated by driving it to the logic value 0. You are to write a VHDL entity that implements logic functions that represent circuits needed to activate each of the seven segments. Use only simple VHDL assignment statements in your code to specify each logic function using a Boolean expression.

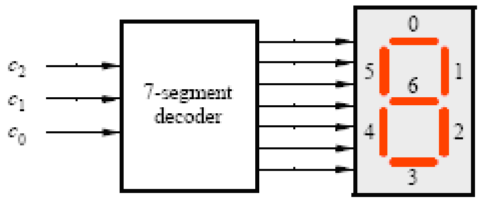


Figure 5.1: A 7-segment decoder

| $c_2c_1c_0$ | Character |
|---|---|
| 000 | H |
| 001 | E |
| 010 | L |
| 011 | O |
| 100 | |
| 101 | |
| 110 | |
| 111 | |

Table 5.1: Character codes

## Part 2

Consider the circuit shown in Figure 5.2. It uses a three-bit wide 5-to-1 multiplexer to enable the selection of five characters that are displayed on a 7-segment display. Using the 7-segment decoder from Part 1 this circuit can display any of the characters H, E, L, O, and 'blank'. The character codes are set according to Table 5.1 by using the switches $SW_{14-0}$, and a specific character is selected for display by setting the switches $SW_{17-15}$.

Note that we have used the circuits from part 1 of this experiment and part 2 of experiment 1 as subcircuits in this code. Use five 7-segment displays rather than just one. You will need to use five instances of each of the subcircuits. The purpose of your circuit is to display any word on the five displays that is composed of the characters in Table 1, and be able to rotate this word in a circular fashion across the displays when the switches $SW_{17-15}$ are toggled. As an example, if the displayed word is HELLO, then your circuit should produce the output patterns illustrated in Table 5.2.

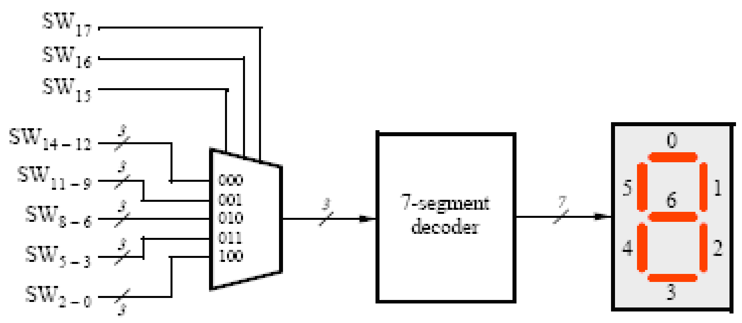


Figure 5.2: A circuit that can select and display one of 5 characters

| $SW_{17}$ $SW_{16}$ $SW_{15}$ | Character pattern | | | | |
|---|---|---|---|---|---|
| 000 | H | E | L | L | O |
| 001 | E | L | L | O | H |
| 010 | L | L | O | H | E |
| 011 | L | O | H | E | L |
| 100 | O | H | E | L | L |

Table 5.2: Rotating the world HELLO on 5 displays

## Prelab Assignment

1. Review part 2 of lab one to understand how the 5-1 multiplexer was designed.
2. Draw a block diagram of your final design for part 2 using components from experiment 1, part 2 and experiment 5, part 1.

## Lab Assignment

Develop VHDL code to implement the designs specified in parts 1 and 2 of this experiment. Be sure to create a separate project for each part and to simulate your circuit to verify functionality. For final demonstration, you should be able to decode the characters on a HEX display for part one and rotate those characters on 5 HEX displays for part 2.

# Experiment 6: ALUs and the LPM Library

## *Objectives*

The objective of this experiment is to design and develop code for an ALU that can perform 2's complement addition and subtraction. Also, we introduce utilizing the LPM library to use components that have been predefined by Altera's libraries.

## *Preparation*

- Review the design of a ripple-carry adder and how to use the LPM library

## *Equipment Needed*

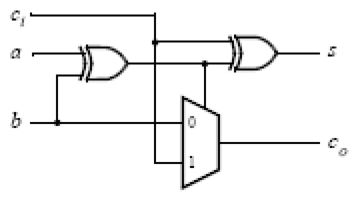
- Altera DE2 Board

## *References*

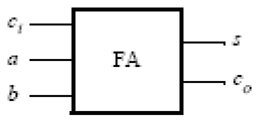
- John D. Carpinelli, *Computer Systems Organization & Architecture*, Addison Wesley, 2001
- Morris Mano, *Computer Engineering Hardware Design*, Addison Wesley

## *Background*

We begin by reviewing the workings of a 4-bit ripple-carry adder. Figure 6.1a below shows the circuit for the full adder. Figure 6.1b shows what the symbol looks like and figure 6.1c shows the truth table. Figure 6.1d displays the 4-bit ripple-carry adder.



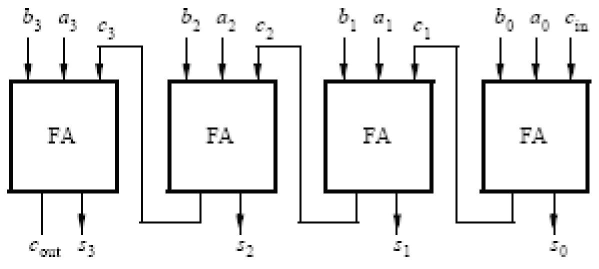
a) Full adder circuit



b) Full adder symbol

| $b$ | $a$ | $c_i$ | $c_o$ | $s$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Figure 6.1: A ripple-carry adder

## Part 1

You are to create an 8-bit version of the adder and include it in the circuit shown in Figure 6.2. Your circuit should be designed to support signed numbers in 2's-complement form, and the overflow output should be set to 1 whenever the sum produced by the adder does not provide the correct signed value.

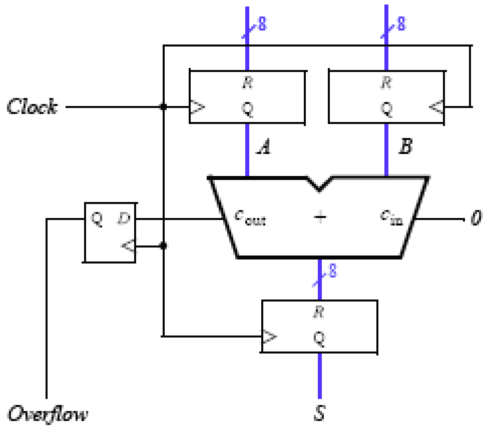


Figure 6.2: An 8-bit signed adder with registered inputs and outputs

Include the required input and output ports in your project to implement the adder circuit on the DE2 board. Connect the inputs A and B to switches $SW_{15-8}$ and $SW_{7-0}$, respectively. Use $KEY_0$ as an active-low asynchronous reset input, and use $KEY_1$ as a manual clock input. Display the sum outputs of the adder on the red $LEDR_{7-0}$ lights and display the overflow output on the green $LEDG_8$ light. The hexadecimal values of A and B should be shown on the displays $HEX_{7-6}$ and $HEX_{5-4}$, and the hexadecimal value of S should appear on $HEX_{1-0}$. Use the LPM defined components to create the necessary registers and the D flip-flop.

## Part 2

Modify your circuit from Part 1 so that it can perform both addition and subtraction of eight-bit numbers. Use switch $SW_{17}$ to specify whether addition or subtraction should be performed. Connect the other switches, lights, and displays as described for Part I. If subtraction is performed your circuit should convert the answer to its absolute value form and use $LEDG_8$ to indicate that the solution is negative.

## Part 3

Create a circuit using the predefined adder circuit called lpm.add.sub instead of your ripple-carry adder. This circuit will maintain 2's complement form if an overflow is found on a subtraction.

## *Prelab Assignment*

1. Create a block diagram of your 8-bit ripple-carry adder.
2. Create algorithm to convert a negative number in 2's complement to its absolute value form for use in part 2.

## *Lab Assignment*

Develop VHDL code to implement the designs specified in parts 1, 2, and 3 of this experiment. Be sure to create a separate project for each part and to simulate your circuit to verify functionality. For final demonstration, you should be able to show the correct functionality of the adder and subtracter specified for each part.

# Experiment 7: Microcoded CPU Design

## *Objectives*

The objective of this experiment is to design and implement a microcoded CPU using the ALTERA DE2 board. A VHDL program with a structural description will be developed to model the CPU. This experiment will also familiarize the student will the lpm library components.

## *Preparation*

- Review Chapter 7 of *Computer Systems Organization & Architecture*

## *References*

- John D. Carpinelli, *Computer Systems Organization & Architecture*, Addison Wesley, 2001
- Morris Mano, *Computer Engineering Hardware Design*, Addison Wesley

## *Equipment Needed*

- ALTERA DE2 board
- Signal generator for clock

## *Background*

In this experiment, the student will design a microcoded CPU based on the ALTERA DE2 board. A VHDL program based on a structural description will be developed to model the CPU. You must use the lpm library components to design the CPU and the RAM. Two components, the ALU and a simple microsequencer based on Experiment 2 will be provided.

### CPU Specifications

The CPU has the following specifications:

- The CPU can access 256 8-bit words of RAM. This implies that an 8-bit Memory Address Register (MAR) and an 8-bit Program Counter (PC) are needed
- The CPU accesses the 8-bit memory via an 8-bit Memory Data Register (MDR)
- The CPU has two internal 8-bit registers: the accumulator (A) and a general-purpose register (R)
- The CPU has a 1-bit condition flip-flop (Z), which derives its input as described below

The CPU is capable of executing the instructions shown in Table 5.1 (The instructions shown in italics are for extra credit.) Note that V is defined as the logical OR of the bits of A; this is similar in function to a zero flag. Also note that LOAD, LOADSP, STORE, JUMPZ and JUMP use 8-bit address (Γ) as part of the instruction code. In addition to the CPU design, your VHDL code must include a 256 words 8-bit RAM for programs and data.

- *Extra credit*: Implement the LOADSP, PUSH and POP instructions, along with the 8-bit stack pointer (SP) used by these instructions

| **Instruction code** | **Instruction** | **Function** |
|---|---|---|
| 00000000 | NOP | No operation |
| 00100000 Γ | LOAD Γ | A ← M[Γ] |
| 00110000 Γ | STORE Γ | M[Γ] ← A |
| 01000000 | MOVE | R ← A |
| 01010000 | ADD | A ← A + R |
| 01100000 | AND | A ← A and R |
| 01110000 | TESTNZ | Z ← not V |
| 01110001 | TESTZ | Z ← V |
| 10000000 Γ | JUMP Γ | PC ← Γ |
| 10010000 Γ | JUMPZ Γ | If (Z = 1) then PC ← Γ |
| 11000000 Γ | *LOADSP Γ* | SP ← Γ |
| 11010000 | *PUSH* | M[--SP] ← A |
| 11100000 | *POP* | A ← M[SP++] |
| 11110000 | HALT | PC ← 0, stop microsequencer |

Table 7.1: CPU instruction set

## ALU and Microsequencer

Two components are provided to reduce the complexity of this experiment. A 2-function (add and logical-and) 8-bit ALU is provided and is shown in Figure 7.1. The component declaration for the ALU is as follows:

```
component exp7_alu is
port (a, b: in std_logic_vector(7 downto 0);
      op: in std_logic_vector(0 downto 0);
      result: out std_logic_vector(7 downto 0));
end component;
```

where *a* and *b* are the 8-bit inputs, *result* is the 8-bit output and *op* is a 1-bit function selection:

$$result = \begin{cases} a+b, & op = 0. \\ a \text{ and } b, & op = 1. \end{cases}$$

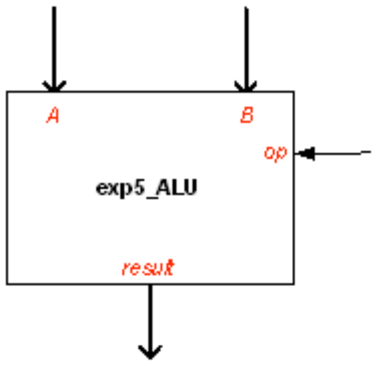


Figure 5.1: 8-bit ALU with add and logical-and functions

The VHDL code for the ALU is included in Appendix 7.1. A simple microsequencer is also provided (Figure 7.2). The component declaration for the microsequencer is as follows:

```
component exp7_useq is
generic (uROM_width: integer;
         uROM_file: string);
port (opcode: in std_logic_vector(3 downto 0);
      uop: out std_logic_vector(1 to (uROM_width-9));
      enable, clear: in std_logic;
      clock: in std_logic);
end component;
```

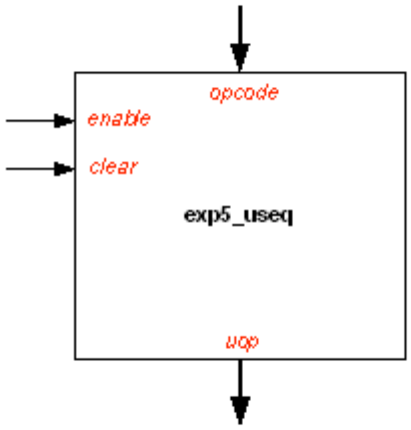


Figure 7.2: Simple microsequencer

The microsequencer contains a 256-word micro-ROM. The width of each micro-ROM word is declared through the parameter *uROM_width.* The parameter *uROM_file* is the name of the mif (memory initialization file) for the contents of the micro-ROM. The organization of the content of the micro-ROM is shown in Figure 5.3.

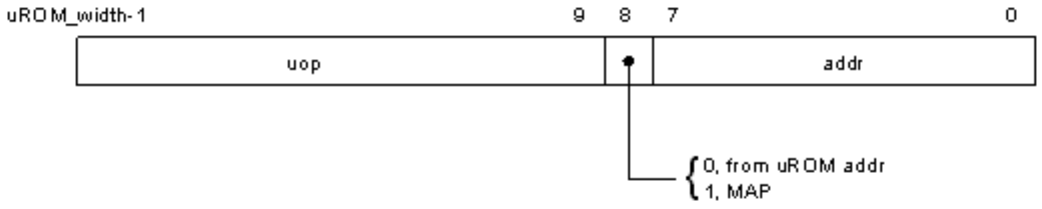


Figure 7.3: Organization of micro-ROM content.

The lower order 8-bit (bits 7 through 0) is the address of next microinstruction when bit 8 is 0. If bit 8 is 1, then the next microinstruction address is obtained through a mapping function. The mapping function generates an 8-bit value by concatenating the 4-bit *opcode* input with “0000”. The length of the micro-operation field equals to (*uROM_width* – 9). For example, if *uROM_width* is 20, then *uop* goes from 1 to 11 that corresponds to bits 19 through 9 of the micro-ROM content. The other 3 inputs to the microsequencer are:

1. *clock* – the system clock.
2. *enable* – enables the microsequencer so that transitions can occur.
3. *clear* – resets the address register in the microsequencer to 0.

The VHDL code for the microsequencer is included in Appendix 7.2. The 256x8 RAM should be implemented with the lpm library component, *lpm_ram_dq*.

## Prelab Assignment

1. Prepare a preliminary hardware block diagram design for your CPU. Explain the purpose of the blocks and components.
2. Prepare a preliminarily commented version of the microcode for your system.

## Lab Assignment

Develop VHDL code using a structural approach for the CPU described above. Program, debug and test your design. For final demonstration, your CPU should be able to run all the test cases and display the result of A using the two 7-segment displays on the DE2 board.

## Appendix 2.1

Part 1

```
-- ECE 495 - Experiment 2, part 1
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY part1 IS

      PORT(X, Y:        IN STD_LOGIC_VECTOR(7 downto 0);
             M:               OUT STD_LOGIC_VECTOR (7 downto 0);
             sel:       IN STD_LOGIC);
END part1;


ARCHITECTURE apart1 OF part1 IS
      BEGIN
            -- generate function for each spot in our 8-element vector
            M_G: for i in 7 downto 0 generate
                  -- function to multiplex the two inputs
                  M(i) <= (NOT sel AND X(i)) or (sel AND Y(i));
            end generate;
end apart1;
```

Part 2

```
library IEEE;
use IEEE.std_logic_1164.all;
ENTITY part2 IS
      PORT(sel, U, V, W, X, Y:      IN STD_LOGIC_VECTOR(2 downto 0);
             M:         OUT STD_LOGIC_VECTOR (2 downto 0));
END part2;
ARCHITECTURE apart2 OF part2 IS

-- need signals for 3 outputs (one from each mux)
SIGNAL mux1, mux2, mux3:      STD_LOGIC_VECTOR(2 downto 0);
      BEGIN
            -- generate the output signals for each of the muxes
            mux1_G: for i in 2 downto 0 generate
                  mux1(i) <= (NOT sel(0) AND U(i)) or (sel(0) AND V(i));
            end generate;
            mux2_G: for i in 2 downto 0 generate
                  mux2(i) <= (NOT sel(0) AND W(i)) or (sel(0) AND X(i));
            end generate;
            mux3_G: for i in 2 downto 0 generate
                  mux3(i) <= (NOT sel(1) AND mux1(i)) or (sel(1) AND
mux2(i));
            end generate;
            M_G: for i in 2 downto 0 generate
                  M(i) <= (NOT sel(2) AND mux3(i)) or (sel(2) AND Y(i));
            end generate;
END apart2;
```

## Appendix 4.1

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

ENTITY lcd_disp IS
       PORT(reset, clk:                              IN STD_LOGIC;
               LCD_RS, LCD_EN, LCD_ON:      OUT STD_LOGIC;
               LCD_RW:                                      BUFFER STD_LOGIC;
               LCD_DATA:                                    INOUT STD_LOGIC_VECTOR(7 downto 0));
END lcd_disp;

ARCHITECTURE alcd_disp OF lcd_disp IS
       TYPE states IS (hold, func_set, display_on, mode_set, write_char1,
                                            write_char2, write_char3, write_char4, write_char5,
                                            write_char6, write_char7, write_char8, write_char9,
                                            write_char10, write_char11, write_char12,
return_home,
                                            toggle_e, reset1, reset2, reset3, display_off,
display_clear);
       SIGNAL state, next_command: states;
       SIGNAL data_bus: STD_LOGIC_VECTOR(7 downto 0);
       SIGNAL clk_count_400Hz: STD_LOGIC_VECTOR(19 downto 0);
       SIGNAL clk_400Hz: STD_LOGIC;
       SIGNAL char1_data, char2_data:       STD_LOGIC_VECTOR(7 downto 0);

BEGIN
       LCD_ON <= '1';

-- bidirectional tri-state LCD data bus
       LCD_DATA <= data_bus WHEN LCD_RW = '0' ELSE "ZZZZZZZZ";

       PROCESS
       BEGIN
              -- must slow down clock to 400Hz for LCD use
              WAIT UNTIL rising_edge(clk);
              IF reset = '0' THEN
                     clk_count_400Hz <= X"00000";
                     clk_400Hz <= '0';
              ELSE
                     IF clk_count_400Hz < X"0F424" THEN
                            clk_count_400Hz <= clk_count_400Hz + 1;
                     ELSE
                            clk_count_400Hz <= X"00000";
                            clk_400Hz <= NOT clk_400Hz;
                     END IF;
              END IF;
       END PROCESS;

       PROCESS (clk_400Hz, reset)
       BEGIN
              IF reset = '0' THEN
                     state <= reset1;
                     data_bus <= X"38";
                     next_command <= reset2;
                     LCD_EN <= '1';
                     LCD_RS <= '0';
                     LCD_RW <= '0';

              ELSIF rising_edge(clk_400Hz) THEN
                     CASE state IS
-- Set Function to 8-bit transfer and 2 line display with 5x8 Font size
-- see Hitachi HD44780 family data sheet for LCD command and timing details
                            WHEN reset1 =>
                                   LCD_EN <= '1';
                                   LCD_RS <= '0';
                                   LCD_RW <= '0';
```

```
                                                data_bus <= X"34";
                                                state <= toggle_e;
                                                next_command <= reset2;
                                        WHEN reset2 =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"34";
                                                state <= toggle_e;
                                                next_command <= reset3;
                                        WHEN reset3 =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"38";
                                                state <= toggle_e;
                                                next_command <= func_set;
                                        -- states above needed for pushbutton reset of LCD display
                                        WHEN func_set =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"34";
                                                state <= toggle_e;
                                                next_command <= display_off;
                                        -- turn off display and turn off cursor
                                        WHEN display_off =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"08";
                                                state <= toggle_e;
                                                next_command <= display_clear;
                                        -- turn on display and turn off cursor
                                        WHEN display_clear =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"01";
                                                state <= toggle_e;
                                                next_command <= display_on;
                                        -- turn on display and turn off cursor
                                        WHEN display_on =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"0C";
                                                state <= toggle_e;
                                                next_command <= mode_set;
                                        -- set write mode to auto increment address and move cursor to the
right
                                        WHEN mode_set =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '0';
                                                LCD_RW <= '0';
                                                data_bus <= X"06";
                                                state <= toggle_e;
                                                next_command <= write_char1;
                                        -- write hex character in first LCD character location
                                        WHEN write_char1 =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '1';
                                                LCD_RW <= '0';
                                                data_bus <= X"48"; -- H
                                                state <= toggle_e;
                                                next_command <= write_char2;
                                        -- write hex character in second LCD character location
                                        WHEN write_char2 =>
                                                LCD_EN <= '1';
                                                LCD_RS <= '1';
                                                LCD_RW <= '0';
```

```
		data_bus <= X"45"; -- E
		state <= toggle_e;
		next_command <= write_char3;
	-- write hex character in third LCD character location
	WHEN write_char3 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"4C"; -- L
		state <= toggle_e;
		next_command <= write_char4;
	-- write hex character in fourth LCD character location
	WHEN write_char4 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"4C"; -- L
		state <= toggle_e;
		next_command <= write_char5;
	-- write hex character in fifth LCD character location
	WHEN write_char5 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"4F"; -- O
		state <= toggle_e;
		next_command <= write_char6;
	-- write hex character in sixth LCD character location
	WHEN write_char6 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"20"; -- space
		state <= toggle_e;
		next_command <= write_char7;
	-- write hex character in seventh LCD character location
	WHEN write_char7 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"57"; -- W
		state <= toggle_e;
		next_command <= write_char8;
	-- write hex character in eighth LCD character location
	WHEN write_char8 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"4F"; -- O
		state <= toggle_e;
		next_command <= write_char9;
	-- write hex character in nineth LCD character location
	WHEN write_char9 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"52"; -- R
		state <= toggle_e;
		next_command <= write_char10;
	-- write hex character in tenth LCD character location
	WHEN write_char10 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
		LCD_RW <= '0';
		data_bus <= X"4C"; -- L
		state <= toggle_e;
		next_command <= write_char11;
	-- write hex character in eleventh LCD character location
	WHEN write_char11 =>
		LCD_EN <= '1';
		LCD_RS <= '1';
```

```
                                        LCD_RW <= '0';
                                        data_bus <= X"44"; -- D
                                        state <= toggle_e;
                                        next_command <= write_char12;
                                -- write hex character in twelveth LCD character location
                                WHEN write_char12 =>
                                        LCD_EN <= '1';
                                        LCD_RS <= '1';
                                        LCD_RW <= '0';
                                        data_bus <= X"21"; -- !
                                        state <= toggle_e;
                                        next_command <= return_home;
                                -- Return write address to first character postion
                                WHEN return_home =>
                                        LCD_EN <= '1';
                                        LCD_RS <= '0';
                                        LCD_RW <= '0';
                                        data_bus <= X"80";
                                        state <= toggle_e;
                                        next_command <= write_char1;
                                -- The next two states occur at the end of each command to the LCD
                                -- Toggle E line - falling edge loads inst/data to LCD controller
                                WHEN toggle_e =>
                                        LCD_EN <= '0';
                                        state <= hold;
                                -- Hold LCD inst/data valid after falling edge of E line

                                WHEN hold =>
                                        state <= next_command;
                        END CASE;
                END IF;
        END PROCESS;
END alcd_disp;
```

## Appendix 7.1

VHDL code for exp7_alu

```
-- ECE 485 Exp. 7
-- 8-bit ALU
-- Dr. Hou
--
-- result <= a + b,   if op = 0
--        <= a and b, if op = 1
--
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity exp7_alu is
port (a, b: in std_logic_vector(7 downto 0);
      op: in std_logic_vector(0 downto 0);
      result: out std_logic_vector(7 downto 0));
end exp7_alu;

architecture structural of exp7_alu is
signal add_result, and_result: std_logic_vector(7 downto 0);
signal mux_data: std_logic_2D(1 downto 0, 7 downto 0);
begin
  alu_adder: lpm_add_sub
    generic map (lpm_width=>8)
    port map (dataa=>a, datab=>b, result=>add_result);
  and_result <= a and b;
  for_label: for i in 7 downto 0 generate
    mux_data(0,i) <= add_result(i);
    mux_data(1,i) <= and_result(i);
  end generate;
  alu_mux: lpm_mux
    generic map (lpm_width=>8, lpm_size=>2, lpm_widths=>1)
    port map (data=>mux_data, result=>result, sel=>op);
end structural;
```

# Appendix 7.2

VHDL code for exp7_useq

```
-- ECE 485 Exp. 7
-- uSequencer
-- Dr. Hou
--
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity exp7_useq is
  generic (uROM_width: integer := 25;
           uROM_file: string := "");
  port (opcode: in std_logic_vector(3 downto 0);
        uop: out std_logic_vector(1 to (uROM_width-9));
        enable, clear: in std_logic;
        clock: in std_logic);
end exp7_useq;

architecture structural of exp7_useq is

signal uROM_address: std_logic_vector (7 downto 0);
signal uROM_out: std_logic_vector (uROM_width-1 downto 0);
signal uPC_mux_data: std_logic_2D(1 downto 0, 7 downto 0);
signal uPC_mux_sel: std_logic_vector(0 to 0);
signal uPC_mux_out: std_logic_vector(7 downto 0);
signal temp: std_logic_vector(7 downto 0);
begin
  temp <= opcode & "0000";
  for_label: for i in 0 to 7 generate
    uPC_mux_data(0, i) <= uROM_out(i);
    uPC_mux_data(1, i) <= temp(i);
  end generate;
  uPC_mux_sel(0) <= uROM_out(8);
  uPC_mux: lpm_mux
    generic map (lpm_width=>8, lpm_size=>2, lpm_widths=>1)
    port map (result=>uPC_mux_out, data=>uPC_mux_data, sel=>uPC_mux_sel);
  uPC: lpm_ff
    generic map (lpm_width=>8)
    port map (clock=>clock, data=>uPC_mux_out, q=>uROM_address,
              sclr=>clear, enable=>enable);
  uROM: lpm_rom
    generic map (lpm_widthad=>8, lpm_width=>uROM_width, lpm_file=>uROM_file)
    port map (address=>uROM_address, q=>uROM_out, inclock=>clock,
              outclock=>clock);

  uop <= uROM_out(uROM_width-1 downto 9);
end structural;
```