

ECE-446
Advanced Digital Logic Design
Laboratory Manual

Written By:
Dr. Jafar Saniie
Andrew Piorunski
Erdal Oruklu

Introduction

This laboratory course is intended to provide the experimenter with experience developing advanced digital logic circuits, and with downloading these circuits to an FPGA programmable logic device. To this end, the VHDL hardware definition language is used for specifying nearly all the circuits for this laboratory course. VHDL code files may be entered into the Xilinx ISE circuit development environment for testing and eventual download to a test board containing a Field Programmable Gate Array (FPGA) programmable logic device.

Some students may be taking this course as a remote student, or with a remote student as a lab partner. There are some other tools they should become familiar with. The first is the Elluminate software. This is the environment through which local and remote students will interact with each other. It is recommended that students read through the tutorial found in the Elluminate Manual, and answer the questions at the end.

The other tools available to remote students are remote interfaces to the Oscilloscope and Function Generator. Documentation for how to access these interfaces can be found in Appendices F and G.

A basic tutorial of the Xilinx ISE development environment, and the VHDL language itself, is contained within the first few Orientation experiments of this laboratory manual; therefore, this introduction will focus on the hardware to be used in this laboratory course. The test board for this course is developed by Digilent, Inc. and is called the Spartan-3E Starter Kit board built around a Xilinx XC3S500E Spartan-3E FPGA device.

1. The Xilinx Spartan-3E FPGA

The Spartan-3E FPGA device is composed of a matrix of Configurable Logic Blocks (CLBs), surrounded by an array of programmable Input/Output Blocks (IOBs). Specifically, the XC3S500E FPGA that is used for this laboratory course contains 1,164 total CLBs in a matrix of 46 rows and 34 columns. A CLB in the Spartan-3E device is organized into four slices. Each CLB is composed of four slices. Each slice, in turn, is composed of two Look-Up Tables (LUT), some control logic, and two storage elements to be used as a flip-flop or latch. There are two types of slices, two SLICEL and two SLICEM per CLB. SLICEM slices have additional hardware, two 16x1 RAM blocks and two 16-bit Shift Registers. This allows SLICEM slices to act as not only logic but also as distributed RAM and Shift Registers. Each LUT and its corresponding storage element are called a Logic Cell (LC). Each LC in the slice is a versatile memory device, and is used to store the functional specification of part of a larger digital logic circuit. These LCs may act as standard memory, shift registers, or driving devices for a portion of a circuit. From benchmark standards, Xilinx LCs (LUT plus storage element) have been found to be more efficient than LUT only designs, making each slice equivalent to 2.25 simple LCs. This provides a total equivalent performance of 10,476 Logic Cells for this device. The functional behavior of each LC in the Spartan-3E device is determined automatically by the Xilinx ISE development environment based upon the circuit to be programmed to the FPGA.

While the logic cells in each CLB may be used as memory storage elements, this could prove wasteful for memory intensive programs. For this reason, the Spartan-3E

FPGA contains twenty blocks of supplementary RAM, each containing 18K bits of storage. Just as with programming the LCs in the FPGA, the usage and programming of these RAM blocks is decided by the compiler in the Xilinx ISE development tool. When the RAM is considered together with the CLBs in the FPGA, the XC3S500E has a total of 500,000 system gates.

The XC3S500E also contains twenty Dedicated Multipliers. The multiplier blocks allow for full 18 bit two's complement multiplication with a full precision 36 bit output. In addition to the smaller multiplication programmable ability of the LUTs these provide the ability to perform efficient multiplication for a wide variety of applications.

In addition, the XC3S500E contains 232 user available IOBs, defining the maximum number of inputs and outputs possible for the device. Each IOB contains three pairs of storage elements, an input pair, an output pair, and a three-state pair. Each of which through a VHDL program, may be used as edge-triggered D-type flip-flops for registered outputs, or may be bypassed for standard combinational logic output functionality. The IOBs on the Spartan-3E Starter Kit board are configured to accept LVTTTL voltage levels of 0.0 volts for logic low, and 3.3 volts for logic high. Any external circuits that may be constructed to interface with the Spartan-3E FPGA **should not exceed** these voltage ranges under any circumstances. If this interface voltage range is not strictly adhered to, permanent damage to the Spartan-3E Starter Kit board and Spartan-3E FPGA may result.

This has been a very cursory description of the structure of the Spartan-3E FPGA used in this laboratory course. For much more detailed information about the device functionality, please consult the datasheet for the Spartan-3E family, which can be found, along with a great deal of other useful information, at the Xilinx web site. The URL for the company web site may be found in the References section at the end of this introduction.

2. The Spartan-3E Starter Kit board

While the Spartan-3E FPGA is a very capable device, it is not something that may simply be dropped into a breadboard and programmed. Typically, complicated devices require complicated interface protocols, and this device is no exception. Fortunately, the people at Digilent, Inc. incorporated much of the hard work interfacing with this FPGA into their 3E main board, allowing for a quite simple to use test system.

The Spartan-3E Starter Kit board interfaces with a computer via its USB port. This connection allows designs created in the Xilinx ISE development environment to be downloaded directly into the Spartan-3E FPGA for testing and analysis. Configuration of the FPGA is achieved through an industry standard protocol called boundary-scan JTAG. This protocol allows anywhere from one to many FPGA devices to be daisy-chained together and programmed sequentially from a single source. This programming source may be a configuration PROM or, as in this laboratory course, an active signal source such as the USB port.

In addition to the JTAG support circuitry, the Spartan-3E Starter Kit board contains an on-board crystal oscillator, a SMA connector for external clock sources, and an 8 pin DIP socket. The boards used in this laboratory course have 50 MHz oscillators mounted. In addition to the crystal oscillator, a pushbutton switch on the main board may also be used as a clock source for designs downloaded into the FPGA. While this pushbutton

switch may be useful for some simple designs, it is not debounced, and may produce undesirable effects when testing certain circuits.

Finally, the Spartan-3E Starter Kit board is equipped with three expansion ports that may be used to interface with other test equipment produced by Digilent, Inc. A large FX2 expansion port and 2 6-pin expansion ports. These 6-pin expansion ports are used most frequently to interface directly with the PMOD-SWITCH boards discussed in the next section of this introduction. Some of the later laboratory experiments, however, will employ a Digilab breadboard inserted to the FX2 Edge Connector in order to interface custom circuitry with the Spartan-3E FPGA.

3. The Spartan-3E Starter Kit board I/O and PMOD-SWITCH Board

As this laboratory course proceeds, the experimenter will most likely become very familiar with the Digilab PMOD-SWITCH interface board and main board I/O devices. From the very first experiment onward, four toggle switches and eight individual LEDs on the main board plus the eight addition switches provided by the PMOD-SWITCH boards will be the primary means of interfacing with a circuit downloaded into the Spartan-3E FPGA. In addition to these standard interface devices, the main board also contains four pushbutton switches and a rotary push button, which may also be used to provide data to the Spartan-3E FPGA. In order to utilize any of these interface devices, inputs and outputs of a VHDL design are assigned to Spartan-3E FPGA pins, which, in turn, are hardwired to specific devices. Appendix A of this laboratory manual contains a Spartan-3E pin to I/O device mapping that will be used throughout this laboratory course.

In addition to the simple I/O devices already discussed, the Spartan-3E Starter Kit board also contains a 2 line by 16 character LCD display. Using LCD is a practical way to display a variety of information using standard ASCII and custom characters. The FPGA controls the LCD via the 4-bit data interface. Although the LCD supports an 8-bit data interface, the Starter Kit board uses a 4-bit data interface to remain compatible with other Xilinx development boards and to minimize total pin count. Most applications treat the LCD as a write only peripheral and never read from the display.

The 2 x 16 character LCD has an internal graphics controller that has three internal memory regions, each with a specific purpose; DD RAM, CG ROM and CG RAM. The Display Data RAM (DD RAM) stores the character code to be displayed on the screen. Most applications interact primarily with DD RAM. The Character Generator ROM (CG ROM) contains the font bitmap for each of the predefined characters that the LCD screen can display. The Character Generator RAM (CG RAM) provides space to create eight custom characters Bitmaps.

Beyond even the LCD display in terms of complexity, the main board also provides both a PS2 port and a VGA port. The PS2 port allows for a mouse or a keyboard to input data to the Digilab test system, and the VGA port allows data to be displayed on a computer monitor. Both of these interface ports require complex timing and control that will not be introduced in the course of the regular laboratory experiments, but may optionally be explored in the final course project.

4. References

For further information on any of these hardware components, feel free to consult the following web sites for datasheets and schematics:

1. For information about the Spartan-3E FPGA see: www.xilinx.com.
2. For information about the Digilab test board see: www.digilentinc.com.

Laboratory Experiment 1:

Code Conversion

1. Purpose

In this lab, you will design and build a simple code converter circuit using the Spartan-3E Starter Kit board, which is designed around the Xilinx Spartan-3E XC3S500E FPGA. During this laboratory, you will gain experience using the Xilinx ISE digital circuit development tools and the ISim circuit simulation tools.

2. Background

Converting data from a human-friendly format into a form that is understandable by a machine is a key concept in any digital logic design environment. Computers and digital logic circuits in general, are only able to interpret data in binary form. Every form of data, from integers to characters, must be converted into a binary string in order for a computer to be able to manipulate and store them. This process is called encoding the data for use in a computer. Literally any data-encoding scheme will work as long as data is converted into a binary form that the computer is programmed to understand. For this reason, there are numerous different binary codes that may all represent the same thing. For instance, the ASCII code (American Standard Code for Information Interchange) utilizes a seven-bit string to represent 128 basic text characters, numbers, and control characters used in word processing. More recently, with the introduction of the Internet and globalization, a new code has been required to handle a vastly greater number of characters from languages throughout the world. Unicode was introduced to handle the vast number of characters and symbols now required for worldwide communication. It uses up to 24-bits and several encoding algorithms to achieve this. Clearly, a machine using ASCII codes would need some form of data conversion processing to understand Unicode, and vice-versa. In fact, many web pages containing foreign languages may not be displayed correctly depending on the Unicode support a given web browser allows.

This laboratory experiment will be focused on data conversion, but nothing as complicated as ASCII to Unicode. When all that is required is to encode integers from 0 through 9, encoding data becomes much simpler, although there are still a great number of different encodings that are possible. Binary Coded Decimal (BCD) is a code that uses four bits to represent each of the digits from 0 to 9. Each integer value is encoded directly in its binary form, producing several unused four-bit combinations. Excess-3 code is similar to BCD in that it uses a four-bit encoding for each integer value, however, when an integer is encoded, its value is first incremented by three, then converted to its binary form. The overall effect is to add three to all the Natural BCD codes. While Natural BCD encoding has six unused codes at the high end of the binary conversion (10 and above), Excess-3 BCD has three unused codes at the low end (2 and below) and three at the high end (13 and above). Table 1 shows the integer values from 0 through 9 and their corresponding Natural BCD and Excess-3 BCD encodings.

Decimal	Natural BCD				Excess-3			
	A3	A2	A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Table 1 – BCD Codes

3. Preliminary Design

In this laboratory, two different code conversion circuits will be designed based upon the data encodings in Table 1. The first circuit will capture an input number from the user in Natural BCD form, and will output its Excess-3 BCD representation. The second circuit that will be built will perform both code conversion processes depending on the state of a selection input, SEL. When the selection input is low, the conversion will take in Natural BCD and output Excess-3 BCD, just like the first conversion circuit described above. However, when the selection input is high, this new circuit reads a number in Excess-3 BCD and outputs its Natural BCD counterpart. Figure 1 below contains block diagrams for both of these circuits.

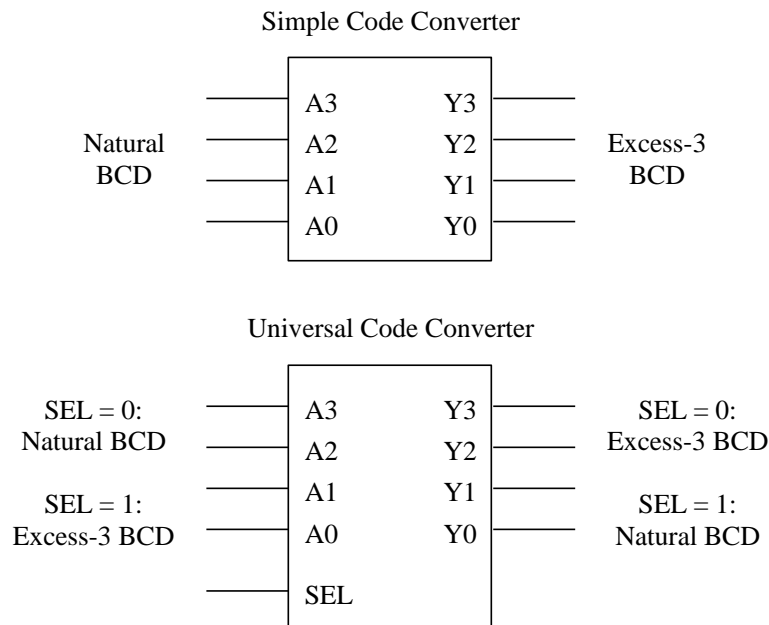


Figure 1 – Converter Circuit Block Diagrams.

With the exception of not including the unused codes for Natural BCD, Table 1 above is essentially the truth table for the first circuit to be designed for this laboratory. Using this truth table, the logic functions for each output of the simple code converter can be relatively easily obtained and minimized using Karnaugh Maps. Recognize that the outputs for the Universal Code Converter depend on one more input, namely the select input, which makes Karnaugh Map minimization slightly more difficult than with the Simple Code Converter. Design of this circuit can be made easier, however, by recognizing that the select input simply chooses between two independent sets of functionality.

Before coming to lab, you should:

1. Develop the logic functions for each output of the Simple Code Converter. Create the full truth table, and minimize the logic using Karnaugh Maps or another preferred method of your choice. Be sure to capitalize upon DON'T CARE conditions where possible.
2. Create the full truth table for the Universal Code Converter circuit. Be sure to include DON'T CARE terms where they apply.
3. From this truth table, derive the logic functions for each output of the Universal Code Converter. Note that the truth table is identical to that for the Simple Code Converter when the select input is zero. You may, therefore, minimize only the half of the truth table corresponding to a select input value of one. The select input may then be used as a conditional check within an IF statement when the circuit is implemented.
4. Bring the output logic functions for each of the two circuits to the laboratory where they will be implemented using VHDL code. The laboratory procedure will then discuss the steps for creating the necessary VHDL code, simulating both designs, and programming Spartan-3E Starter Kit board.

4. In the Lab

The in-lab portion of this laboratory procedure is divided into two main parts: the Simple Code Converter and the Universal Code Converter. The procedure for the Simple Code Converter contains an in-depth walkthrough of project creation, simulation, and programming of the Spartan-3E Starter Kit board, in addition to the basic operations offered by the VHDL language that will be necessary to specify the functionality of the code converter itself. The procedure for the Universal Code converter will then introduce some additional operations offered by the VHDL language that facilitate a simplified specification of the expanded code conversion functionality.

4.1 The Simple Code Converter

4.1.1: Starting a Project

1. Begin by starting the **Xilinx ISE** Project Navigator software. If there is already a project running, close that project and create a **New Project**. This is done under the File drop-down menu, just like creating a new file in most other applications.
2. In the dialog box that pops up, set the directory in which you would like all the files you will create to be saved, and enter the name of the project you would like to create. All the files created by the ISE tools will be placed in a file with the project name specified by you, which itself resides in the directory you picked. In the Top-Level Module Type dropdown, ensure that HDL is listed, and hit **Next >**.
3. In the next dialog box, under Device Family, select Spartan3E. Under Device, select **xc3S500E**, which specifies the type of Spartan-3E FPGA used on the board. Under Package, select **FG320**, which specifies the package type for the FPGA. Under Speed Grade, select **-4**, which specifies the speed grade of the FPGA. For the Synthesis Tool selection, **XST (VHDL/Verilog)** should be selected. The Simulator selection should be set to **ISim (VHDL/Verilog)**. When all of these selections are verified, hit **Next >**. On the next window, hit Finish to conclude creating your project.

4.1.2: Adding a VHDL Code Module to the Project

4. From the previous steps, you should have a project created in the Project Navigator environment. In the Hierarchy in Design window on the upper left side of the screen, you will see a listing for the device you specified under the project name you entered. Right click the device object (**xc3500e-4fg320**) and select **New Source** from the pop-up menu.
5. From the selections provided, choose VHDL Module to create a new VHDL code segment. You will be prompted to enter a name for the file, and then you will be given a screen to enter input and output ports. Enter a port name for each input and each output for the Simple Converter, and select the direction (in or out) for each port. Click **Next >** to be given a summary of your VHDL module.
6. Hit Finish and a window containing VHDL code will be created. Note that a .vhd file has been added to the device in the Sources in Project window of the Project Navigator.

4.1.3: Programming the Simple Converter

7. Before doing anything else, look at the beginning of the VHDL code provided to you. Within the entity definition, you will see the port names you previously entered, followed by their direction and data type.
8. Your code will be entered between the “begin” and the “end” statements in the architecture section of the VHDL code file created for you. Your logic functions may be entered in plain English. For instance: the statement

$Y3 = A3 \bullet A2$

will be entered as:

$Y3 \leq A3 \text{ AND } A2;$

VHDL commands of note:

Combinational output assignment:	<i>output_signal_name <= assigned_value</i>
Logical AND:	<i>signal_1 AND signal_2</i>
Logical NAND:	<i>signal_1 NAND signal_2</i>
Logical OR:	<i>signal_1 OR signal_2</i>
Logical NOR:	<i>signal_1 NOR signal_2</i>
Logical Inverse:	<i>NOT signal_name</i>
Logical Exclusive-OR:	<i>signal_1 XOR signal_2</i>
Logical Exclusive-NOR:	<i>signal_1 XNOR signal_2</i>
Comment Delimiter:	<i>-- comment-text</i>

It should also be noted that each line of code must be terminated with a semicolon (;), just as in C/C++. Using the commands listed above, you may enter the logic functions defining each output of the Simple Converter on your own.

9. Save your VHDL file and synthesize it. This can be done by highlighting the .vhd file in the Hierarchy in Design window, and then double-clicking the **Synthesize XST** process in the Processes window on the lower left side bottom. The development tools will detect any coding errors during this process.

4.1.4: Assigning Pin Numbers to Ports

10. After synthesizing the completed VHDL file, pin numbers must be assigned to each input and output port for programming the Spartan-3E Starter Kit board later. To do this, expand the User Constraints and double-click the **I/O Pin Planning(PlanAhead) – Post-Synthesis**. This will start **PlanAhead** program. Click on yes on the window appearing to create a new constrain file.
11. On PlanAhead application, you’ll see Netlist window on the upper left side of the screen. Click on I/O Ports tab on the bottom of Netlist window. Expand Scalar ports and select a port to assign pin. On I/O Port Properties window on the bottom of Netlist window, assign pin by typing pin location on the Site textbox. To confirm the pin assignment, click on Apply and the pin assignment will be done for the selected

port. Select each port and assign it a pin location (Loc column) based upon the table found in Appendix A of this laboratory manual. Pin numbers may be entered manually.

12. After assigning pin numbers to each port, save the file and close PlanAhead program. Highlight the .vhd file in the Hierarchy in Design window and double-click the Implement Design process in the Processes window. This will map your design to the Spartan-3E device you selected.

4.1.5: Simulating the Simple Converter

Before going to further step, in order to simulate on ISim with VHDL Test Bench, you must have a clock variable declared on your VHDL Source File. Go to Port declaration on your VHDL source code, and type in [CLK : in std_logic;] in the port declaration. Port declaration can be found on the very first lines of your VHDL code, excluding the auto-generated comments.

13. In order to simulate the design you have entered, another new source must be added to the project. Right-click the device and add a New Source. Select **VHDL Test Bench** and give it a name different from the name of the VHDL file it will be testing. This naming difference is important so as not to confuse the simulator. As with the constraints file, the VHDL source to which the simulation file will apply must be selected in the next dialog box. Select your VHDL file, and finish creating the new file.
14. A test bench VHDL file will be created with templates for your convenience. You can specify your own clock period, input values and more. To adjust the clock period, find where it defines a clock period constant. By default, the clock period is set to 10ns. To adjust your input variables, find a comment where it says “insert stimulus here”. You can assign input variables here to simulate your implementation. Assigning method is identical to what we do for VHDL file. Test bench file itself is a VHDL file as well, so you should be comfortable with assigning test values to the input variables. What is important is to include a wait statement between two different test values. You should give at least one clock period wait statement in between them. Otherwise, your test will not work properly as expected. Once you are done writing your own test bench VHDL code, save it.
15. To simulate your test bench, select **Simulation** radio button next to Implementation, on top of the Hierarchy in the Design window. Then, select the VHDL test bench file listed in the Hierarchy in the Design window. In the Processes window, expand the ISim Simulator and double-click the **Simulate Behavioral Model** Program. Before running ISim, right-click on Simulate Behavioral Model and click on Process Properties. Your simulation must fall into the Simulation Run Time specified on the pop-up window, to verify all of your test cases. Change **Simulation Run Time** if the default value is not enough to fulfill your test bench. ISim will start and the output of your design will be simulated for the input waveform that you specified.

16. A new window should appear. It will most likely have a black background with some white vertical lines cutting across it. In order to see the test results, find the **Zoom to Full View** button that should be on the left and on the top of the waveforms:



17. To make checking the simulation results easier, a vertical cursor may be used. The values across all the input/output ports for the time at which the cursor is set will be shown in the column to the right of each port name. The cursor may be moved by clicking within the waveform at a point of interest. By holding the click from one point of interest to another, you can read the time elapsed between two events.
18. When you are certain that the functionality of the Simple Converter is correct, you may exit the ISim simulator and proceed to programming the Spartan-3E Starter Kit board.

4.1.6: Programming the Spartan-3E Starter Kit board

19. To return to the implementation mode, select the implementation radio button. Double-click the **Generate Programming File** process and wait for the program to complete. After the program file is generated, expand the Configure Target Device process with the “+” next to it, and find the **Manage Configuration Project (iMPACT)** executable listed there. Ensure that the Spartan-3E Starter Kit board is connected to power and to your computer’s USB port and run this program.
20. Depending on the version of your ISE, it may be different but the process is the same looking as a big picture. On the upper left on the screen, in iMPACT Flows window, double-click on **Boundary Scan**. Right-click on the right side of the window where it says “**Right click to Add Device or Initialize JTAG chain**”, then click on **Initialize Chain**. On the pop up window, go to your project’s folder (directory) and find *your_project_name*.bit file and open it. This step is very critical. If you select a bit file other than your current project, it may not behave as you desired. In fact, this will load completely different program to the Spartan3E FPGA board. If the program prompts you whether you want to attach an SPI or BPI PROM to this device, click on No. Click on Bypass for the next two pop up window and hit OK on Device Programming Properties window.
21. After choosing the configuration file to be used, the program will present you with an icon representing the .bit programming file that was generated from your code. This icon usually sits on the very left side. Right-click this icon and select **Program**. The board should program, allowing you to test the functionality of your Simple Converter circuit physically, with a success message on the screen.
22. Show the lab instructor your functioning circuit before proceeding to the next section of this laboratory.

4.2 The Universal Code Converter

Much of the basic work for the Universal Code Converter is the same as for the Simple Converter; so, if there are any questions about how to proceed beyond the instruction given here, see the previous section of this laboratory procedure.

23. Open a new project and add a VHDL module to it. The main difference between this circuit and the Simple Converter is the addition of the select signal to change the functionality. While the output functions could be found using five variable Karnaugh Maps, the selectable nature of this circuit lends itself nicely to the use of IF statements.
24. In order to use IF statements, they must be nested within a PROCESS declaration. To make your code into a process, between the “begin” and “end” statements of the architecture code (where your functionality was entered for the Simple Converter); enter PROCESS followed by all the input signals in parentheses on the same line. On the next line, enter BEGIN, then leave a few lines of space and finally enter “END PROCESS;” to complete the process. The result should appear as follows:

```
PROCESS (input_signal1, input_signal2, input_signal3, ...)
BEGIN

END PROCESS;
```

25. In the empty space you left, an IF statement may be entered. The basic format for an IF statement in VHDL is as follows:

```
IF condition check THEN
    Sequence of Statements
ELSIF condition check THEN
    Sequence of Statements
ELSE
    Sequence of Statements
END IF;
```

Similar to programming in C/C++, an IF statement is not required to have an accompanying ELSEIF or ELSE statement, however, every IF statement must have an accompanying END IF to close the functionality. In addition, multiple condition checks may be done within a single IF statement through the use of standard VHDL logical operators, such as AND or OR.

The following operators may be used in the condition check statement:

Equal To:	<i>signal_name</i> = ' <i>bit_value</i> '
Less Than:	<i>signal_name</i> < <i>value</i>

Greater Than: *signal_name > value*
Less Than or Equal To: *signal_name <= value*
Greater Than or Equal To: *signal_name >= value*

If the input signal is being compared to a logic value, this value must be surrounded by single quotes for the synthesizer to interpret its meaning correctly. For the Universal Code Converter, you will be checking the value of the select input signal. When this signal is zero, the sequence of statements implementing the BCD to Excess-3 conversion should be executed. When the signal is one, the Excess-3 to BCD conversion code should be executed instead. Enter the code for both conversions under the appropriate portions of your IF statement and save your VHDL file.

26. Synthesize your design to check for coding errors, and assign pin numbers to the ports on your new program.
27. Modify previous test bench VHDL file to this project and set up your test plan. Simulate your Universal Code Converter circuit using ISim to verify the proper functionality. If your circuit does not function as expected, correct the VHDL code and run the simulation again.
28. When the simulation provides the results you expect, generate the programming file and download it to your Spartan-3E Starter Kit board for physical testing.
29. Verify the circuit functions as required when the select signal is both high and low. When your circuit is functioning correctly, show the lab instructor its operation.

5. References

Additional information about ASCII code may be found at the following website:
www.asciitable.com

Additional information about Unicode may be found at the following website:
www.unicode.org

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 2: Four-Bit Ripple-Carry Adder/Subtractor

1. Purpose

In this lab, you will design and program a four-bit ripple-carry adder/subtractor circuit using modular design techniques. All of the necessary VHDL coding concepts and procedures will be introduced in the laboratory and implemented by the experimenter.

2. Background

Some circuit designs lend themselves very well to expansion through replication of basic processing elements. The ability of a circuit to be expanded through replication depends very much on the nature of the problem that the circuit is solving. If the problem itself allows for partial results calculated from subsets of the full input to be combined to get the correct answer, this type of expandability is often possible. It is frequently easier to replicate the design of a circuit that processes a small subset of a large input set and combine the individual results, than it is to design a single circuit to manipulate the full data set.

One such circuit that may be easily expanded in this way is the ripple-carry adder. By tying single-bit full-adder circuits end-to-end, a ripple-carry adder of any size may be constructed. All that is necessary to achieve the proper functionality is to tie the carry-out port of each block to the carry-in port of the next block, forming a chain. Then, by providing corresponding operand bits to each adder block, the final sum can be calculated bit-by-bit along the chain. The following table contains the truth table for a single-bit full-adder circuit.

Carry-In	A	B	Sum	Carry-Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1 – The truth table for the single-bit full-adder.

A major point in favor of the VHDL language is that it supports modular circuit design very simply and effectively through a hierarchical design practice. The module to be replicated is first coded in VHDL, allowing its functionality to be tested independently of all other portions of a larger design, similar to a function in C/C++. A higher level VHDL module may then reference these lower level sub-modules by including their code

in the larger project. The functionality of the component may then be accessed by passing it an input signal set and utilizing the output signals it generates in turn.

You may recall that a ripple-carry adder circuit, such as the one described earlier, can be quite easily made into an adder/subtractor circuit through the introduction of some strategically placed XOR gates and the introduction of an operation select signal. This circuit requires one XOR gate for each single-bit adder block, whose output is tied to one of the operand inputs of the block. One of the XOR inputs is tied to the operand bit that is either added to or subtracted from the other operand, while the other input is tied to the operation select signal. When the operation select is logic low, the XOR gates have no effect on the operand tied into the other input, so the gates effectively act as buffers. When the operation select input is logic high, however, each of the XOR gates act as an inverter for the operand bit tied to the gate. In this way, we have constructed a circuit that either adds two multi-bit operands, or performs a one's-complement subtraction operation, depending on the setting of the operation select input. To complete the design and produce two's-complement subtraction, the operation select input can also be tied to the carry-in input port of the first adder in the chain. This is the design that will be implemented in this laboratory, and is illustrated in the following schematic:

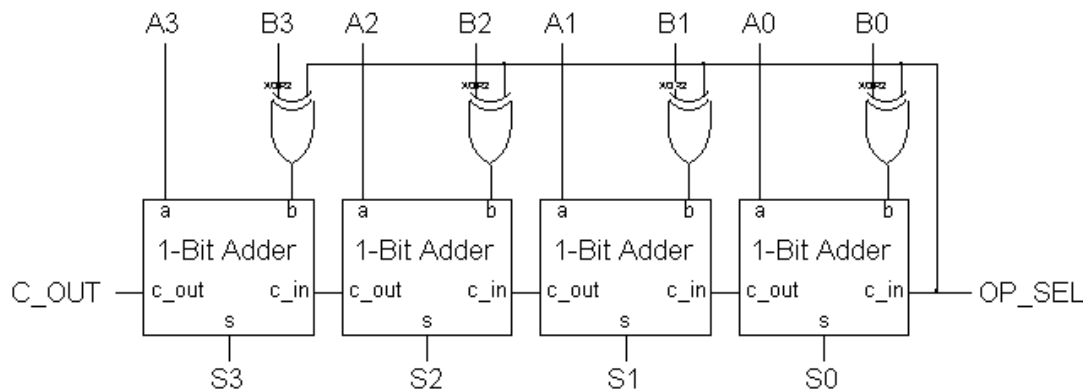


Figure 1 – The schematic diagram for the four-bit ripple-carry adder/subtractor circuit.

3. Preliminary Design

The preliminary design for this laboratory experiment is fairly straightforward. Use any method you choose to develop Boolean equations for the outputs of the one-bit full-adder, the truth table for which is shown in Table 1. In addition, the external XOR gate should be included in each block to make the design of the complete four-bit circuit easier. This may be done quite simply using the Boolean equations for the standard one-bit adder. Every time the B input appears in one of the equations, it may be replaced with the following:

$$(B \oplus \text{op_sel})$$

Before coming to lab, you should:

1. Develop the output logic equations for the one-bit full-adder circuit using the truth table provided above.

2. Replace every occurrence of input B in your logic equations with the logic expression shown above. This will incorporate the XOR gate that is shown externally to each adder block in Figure 1 into the adder block itself, and will make cascading the blocks together easier.
3. Write a VHDL file that will implement your one-bit adder/subtractor module and bring it to the laboratory. Be sure to include input and output ports for both operand bits A and B, the operation select input, the carry-in, the carry-out, and the resulting sum in your design.

4. In the Lab

The primary goal of this laboratory procedure is to introduce the experimenter to the concepts of modular VHDL programming. To that end, this laboratory procedure will detail the new VHDL coding concepts required to implement the modular design, and will not focus on the usage of the ISE programming environment itself. If you have any questions regarding the basic operation of the development environment, please see one of the Xilinx ISE tutorials or a previous laboratory experiment detailing what you require.

The VHDL language allows for straightforward programming with all necessary code included in the definition for a single entity, as has been used up to this point; however, this only scratches the surface of the language capability. Much like a high-level computer language allows for functions to encapsulate sections of code, VHDL allows for the creation of sub-entities containing their own code. The code in these entities is then executed based upon passed in input values and produces results that are assigned to output signals. This can allow a large design to be broken down into more manageable parts, and can make an insurmountable task much more straightforward. This laboratory procedure will walk you through the process of creating these sub-entities and through the process of introducing internal signals that may become necessary when connecting system components together.

1. When you get to the lab, open the Xilinx ISE Project Navigator and create a new project.
2. Add the file for the one-bit adder/subtractor you created earlier to the device in the Sources in Project window. This can be done in a similar fashion to creating a new VHDL module, however, instead of selecting New Source, select Add Source and choose the file you wish to add to the project. It is recommended to name this file something like adder1 to let you know that it is a one-bit adder/subtractor.
3. When you have finished adding your file to the project, simulate it using ISim to verify that it functions as you expect. It is important that you fully test this file, because it will act as a sub-component for a larger project later, which could be difficult, if not impossible to fully test using all the possible input combinations.

4. When you are satisfied that the code you have developed works successfully, create a new VHDL code file in the main project, and name it something like `adder4` to indicate this will be a four-bit adder. This file will contain the higher-level code for the four-bit version of the adder/subtractor. Remember, with this file, there will be four input ports for each operand, an input for the operation select signal, four output ports for the sum, and an output port for the final carry-out.
5. When the skeleton for the new VHDL file is created for you, you must let the program know that the one-bit adder in the other VHDL file will be used within this new file. The single-bit adder must be added as a component within the four-bit adder VHDL file. Declaration of components is done immediately following the architecture definition line in the new VHDL code. It consists of declaring the component name as the name of another VHDL entity in the project, listing the input and output ports of the sub-component, and ending the component definition. An example of the code required to do this is shown below:

```
architecture name of current VHDL entity is
    component entity name of sub-component (i.e. adder1)
        Port(inport1, inport2, ..., inportN : in std_logic;
            outport1, outport2, ..., outportN : out std_logic);
    end component;
```

6. The next step in the process of creating the four-bit adder is to create some internal device signals. These signals will be used to tie the carry-out of one adder block to the carry-in of the next. Since these signals do not need to be sent to the outside world, there is no need to assign ports to them; instead they may be handled internally. There are three of these internal connections required, since the first carry-in is tied to the operation select input, and the last carry-out is sent to an output port for viewing. Internal signals, like the component definitions, go between the “architecture” statement and the “begin” statement signaling the start of the VHDL code. Declaring internal signals may be done using the following command:

```
signal signal1, signal1, ..., signalN : std_logic;
```

Remember, you will need three of these internal signals for this design. You may name them however you wish.

7. The final step in creating the four-bit adder is to actually add the four single-bit adders to the new file, and connect them together to obtain the proper functionality. When a component is instantiated in a VHDL code segment, it must be given a component name followed by a colon and the entity name of the component. A port mapping of signals in the high-level design to component inputs and outputs must then be made for each sub-component in the design. Ports are assigned to each component in the same order that they were listed in the component declaration earlier. Any signal in the high-level VHDL entity may be assigned to an input or output port of the lower-level entity. You may now add four single-bit adder

components to your design, and assign the input and output ports similarly to the following code segment:

```
componentName1 : entityName
    port map (signalToInport1, signalToInport2, ..., signalToInportN,
              signalToOutput1, signalToOutput2, ..., signalToOutputN);
componentName2 : entityName
    port map (signalToInport1, signalToInport2, ..., signalToInportN,
              signalToOutput1, signalToOutput2, ..., signalToOutputN);
.
.
.
componentNameN : entityName
    port map (signalToInport1, signalToInport2, ..., signalToInportN,
              signalToOutput1, signalToOutput2, ..., signalToOutputN);
```

Remember that the signals should be listed in the port mapping statements reflecting the same order as the input and output ports listed in the component declaration statement. It should also be noted that several different kinds of components may be added to a single VHDL file, and they are distinguished in the code above by using different entity names.

8. Save your new file and look at the Sources in Project window. If you have done everything correctly, you should notice that your one-bit adder VHDL file is now nested below the four-bit adder VHDL file. This means that your new file utilizes the code in the previous file in part of its functionality.
9. Assign pin numbers to the input and output ports in your design. Since you will need all eight switches for the two operands, you may use one of the push-button switches for the operation select input.
10. Synthesize your design to ensure that you have no syntax errors. If you have errors, fix them and re-compile your design. When you have no syntax errors, generate the programming file for your design in preparation for programming the Spartan-3E Starter Kit board.
11. Program your Spartan-3E Starter Kit board with your four-bit adder/subtractor and ensure that the circuit functions as you expect in both the addition and subtraction modes. Remember, the carry-out output is only useful when using the addition functionality of the circuit. This value will often have no meaning when using the subtraction functionality. The final result of the subtraction is contained in the four bits of the sum, and is shown in two's-complement form. When your design functions properly, show your lab instructor its behavior.
12. If you do not wish to have multiple files containing your code, any code for sub-entities may be copied into the file that will be using them. This localizes all the code

in a single file, but makes the VHDL harder to read due to the clutter. You may use whichever method you prefer. However, if you combine all the code in a single file, you may need to replace all the “std_logic” signal types with type “bit” in order for the compiler to function properly.

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 3: Hazards and Glitches

1. Purpose

In this lab, you will learn about the transient behavior of certain digital logic circuits, and will gain experience using the logic analyzer to observe the output of a circuit immediately following an input transition. In order to achieve this, the circuits for this laboratory will be made using discrete components, rather than the Spartan-3E Starter Kit board.

2. Background

A glitch is an unwanted transient pulse at the output of a combinational circuit, and a circuit with the potential for a glitch is said to have a hazard. Not all designs will produce glitches, however, if the timing of a circuit is of critical importance, the potential for glitches must be considered and their presence may be corrected. In order to see examples of glitches, the logic analyzer will be required. The operational instructions for the Agilent MSO6032A Logic Analyzer may be found at the end of this laboratory experiment.

2.1 A Circuit with a Static-1 Hazard

Consider the four variable Boolean function $F(A,B,C,D)=\sum(1,3,5,7,8,9,12,13)$. The Karnaugh map for this function is shown below:

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	1	1	1	1
	11	1	1	0	0
	10	0	0	0	0

Figure 1 – The Karnaugh map for the specified Boolean function.

The minimized sum of products form for this function is:

$$F = A\bar{C} + \bar{A}D$$

The following figure shows a physical implementation of this function.

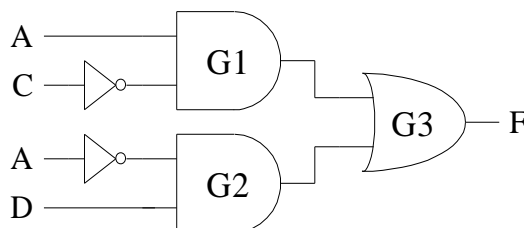


Figure 2 – Circuit implementing the specified logic function.

In order to observe the glitching behavior of this circuit, the output of the circuit may be examined for several input transitions. First, the input data may be changed from $ABCD = 1100$ to $ABCD = 1101$. When the inputs are 1100, the output of gate G1 is 1 while G2's output is 0, forcing the output from G3 to be 1. When the input data changes by a single bit to 1101, the output of gates G1, and hence, G3 do not change, therefore, a glitch will not occur for this input transition. Now consider another single-bit change in the inputs to the circuit, from 1101 to 0101. When A goes low, A' will become high, but only after a gate delay for an inverter, meaning that for a short time, both A and A' are low. This allows the outputs from G1 and G2 to be low at the same time and forces F to go low. When A' finally becomes high, G2 will go high and F will return to the correct value of 1. This situation illustrates a static-1 hazard in the circuit design.

2.2 Correction of a Circuit to Prevent Hazard Conditions

In order to remove hazards in a circuit design, redundant prime implicants must be included to connect any adjacent, but separate, implicants that are already necessary for the design. It is up to the circuit designer to decide which product terms should be added to the sum-of-products function, F, in an effort to eliminate hazard conditions.

3. In the Lab

1. Using the logic analyzer, find the propagation delay for an inverter in a 74LS04 chip. Attach the input for the inverter to a clock signal. Connect the Channel 0 probe of the logic analyzer to this clock signal, and attach another channel probe to the output of the inverter. Use the clock signal input for the inverter to trigger the logic analyzer.
2. Construct the circuit shown in Figure 2 of the Background section of the laboratory using discrete components. Repeat the observation from Step 1 above with one of the inverters in this circuit. Has the propagation delay of the inverter changed and why?
3. Confirm the existence of the glitch discussed in the Background section of this laboratory by providing the circuit with the input transition 1101 to 0101.
4. Construct a version of the circuit that does not exhibit any hazard conditions. Provide this circuit with the same input transition from Step 3 above, and display the outcome on the logic analyzer.

Notes:

- The glitch duration that should result for the circuit being analyzed is very small, on the order of a single inverter delay. In order to see these glitches, the maximum sampling rate for the logic analyzer should be in the range of 2 – 5 ns.
- Logic analyzers are typically used to debug sequential circuits, and their trigger conditions need to be sequential clock or input transitions. Since the circuit being analyzed for this laboratory is combinational, a clock signal should be used to generate the desired input transition for input A.

4. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Logic Analyzer Familiarization

1. Background

You have already become familiar with the use of an oscilloscope for displaying voltage versus time. The oscilloscope can be used as a tool for logic circuits as well as analog circuits, since different voltage ranges represent different logic values. The biggest drawback of doing this is that oscilloscopes have a limited number of channels. In complex logic circuits, we may want to observe several signals simultaneously.

Logic analyzers alleviate this problem. They display logic values that represent the value of digital signals versus time. Logic analyzers can display several of these signals simultaneously, making them ideal for debugging complex circuits. Logic analyzers are no substitute for an oscilloscope when debugging tricky analog problems because they can display only logic one and zero values. However, they excel when you are attempting to debug complex sequential circuits.

Logic analyzers operate by repeatedly sampling data inputs and temporarily storing them while searching for a trigger condition. If no trigger condition is detected, stored values are overwritten by new values as they are sampled. If the trigger condition is detected, then the stored data is shown on the display so that the user can see the values of the signals before, during, and after the trigger condition. In the logic analyzer, these values are displayed as traces similar to an oscilloscope's traces. The traces can be formatted to display binary, octal, hexadecimal or user-defined values.

A major advantage of logic analyzers is that they can collect this data at high speed, making it possible to test circuits at full clock speed where errors are likely to occur. Logic analyzers are extremely useful when debugging sequential circuits, which may contain many signals that change at different times. Examining the signals with a logic analyzer allows you to see if the circuit is behaving properly and, if not, to isolate the source of the problem. Unlike an oscilloscope, which triggers only on a single signal, logic analyzers can trigger on patterns of multiple inputs, making it possible to specify exactly the sequence of events that you wish to examine. Figure 1 shows a diagram of the Agilent MSO6032A logic analyzer's front panel. Notice the similarity between its controls and controls on an oscilloscope. This similarity is intentional to make the logic analyzer easy to learn and use.



Figure 1 – Agilent MSO6032A front view

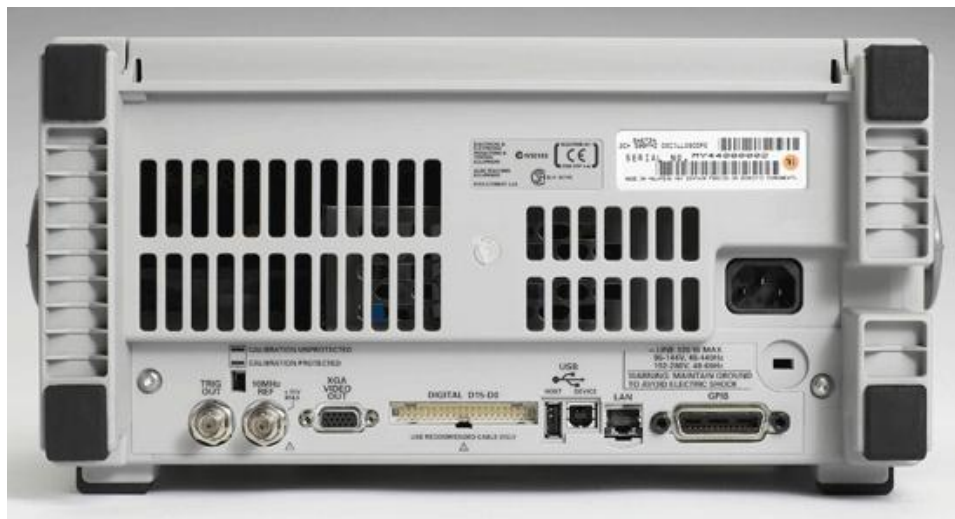


Figure 2 – Agilent MSO6032A back view.

2. Logic Analyzer Basics

There are 16 channel probes connected to the back of the logic analyzer via a ribbon cable. The probes are grouped together into two pods of eight probes, and each probe has a unique number between 0 and 15 that corresponds to its channel number. There is also a ground probe connected to each pod that should be attached to the circuit's ground.

Each probe is connected to a micrograbber. The micrograbber is used to attach to wires or integrated circuit pins. Please be gentle using the probes and micrograbbers - **they are easy to**

break. Always use micrograbbers - **do not insert wires directly into the probes since this may damage the springs inside them.**

To connect a probe to test your circuit, first turn off the power, then connect the micrograbber to the circuit in one of the following ways: 1) to the end of a short piece of wire inserted into the breadboard at your test point or 2) connect the micrograbber directly to an integrated circuit pin. Be careful to avoid shorting two pins together though. It is important to turn off the power so you do not short out your circuit when making the connections. Figure shows an example of connecting the micrograbbers to a circuit under test.

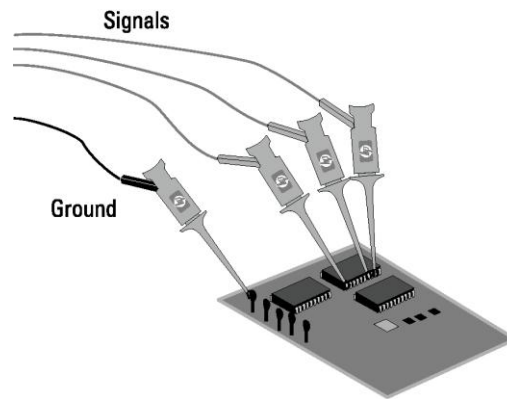


Figure 3 - Micrograbbers Attached To Circuit Under Test.

The controls on the front panel of the logic analyzer can be divided into the following groups: softkeys, channel, horizontal, trigger, storage, and general controls.

Softkeys are located at the bottom on the display and are used in conjunction with other controls. A legend will appear at the bottom of the display above each softkey describing its function depending on what other controls are in use.

Channel controls are used to select which channels will be displayed. Turn the Channel Select knob to position the desired channel on the display. You can assign a label instead of the channel name by pressing the label button and entering a name. The on/off button is used to add or remove a selected channel to or from the display. The position knob is used to move the position of a selected channel up or down with respect to the other display channels. The label button is used in conjunction with softkeys to label different channels with signal names.

Horizontal controls adjust the time scale and a delay from the trigger point to the display. The time/div knob controls the time scale. Turning this knob counterclockwise lengthens the scale up to a maximum of 1 second per division, while turning it clockwise shortens it to a minimum of 2ns/division. The trigger time of each trace is shown at the center of the display with an equal time before and after the trigger. The delay knob can be used to shift where the center point is displayed. This is used to scroll the time axis and display output before and after the trigger.

Trigger controls select how the logic analyzer captures data. Triggers can be specified in three ways. When the Edge button is pressed, the softkeys allow you to select a channel and an edge type of rising \uparrow , falling \downarrow , or glitch \updownarrow . This is similar to oscilloscope triggering. When the Pattern

button is pressed, the softkeys allow you to specify a multiple bit pattern. Each displayed input can be specified as a high, low or don't care value using the softkeys. Any time the inputs match this condition the logic analyzer is triggered. This is especially useful when debugging complex sequential circuits. The Adv button allows the specification of more advanced triggers. The Agilent MSO 6032A manual describes these in detail.

Storage controls determine how data is collected and stored. The Run/Stop key is used to turn the collection of data on (Run) or off (Stop). Pressing the Run button once causes the logic analyzer to continuously wait for a trigger and display data each time a trigger is encountered. Pressing it a second time stops this process. The Single key waits for a trigger condition once and display the result. This is useful when you want to see what happens in response to a single trigger event rather than repeated triggers. The Auto-Store button places the logic analyzer into a mode that displays values for previous traces at half brightness while the current trace is displayed at full brightness. This can be used to see what happens over multiple occurring events.

General controls are used to set up what is displayed. You can modify the display and measure time between events. The Measure Time button allows you to measure time between events. The Save/Recall buttons work with softkeys and allow you to save configurations and restore them. You will often use these buttons in conjunction with the Default softkey to set the logic analyzer to display all channels. The Autoscale button is particularly useful because it will configure the logic analyzer to display all channels on which inputs are active and will guess an appropriate time scale. This is useful for quickly setting up the display, but it will not display channels on which there is no activity. On the bottom left of the display, there is a row of dashes and/or arrows. An up-down arrow indicates activity on a particular probe channel, whereas, a dash does not.

The channel inputs include the probe connector and Logic Levels button. This button allows you to specify which logic family you will be debugging. This should be kept to the default value key TTL, which is the logic family we are using in this course.

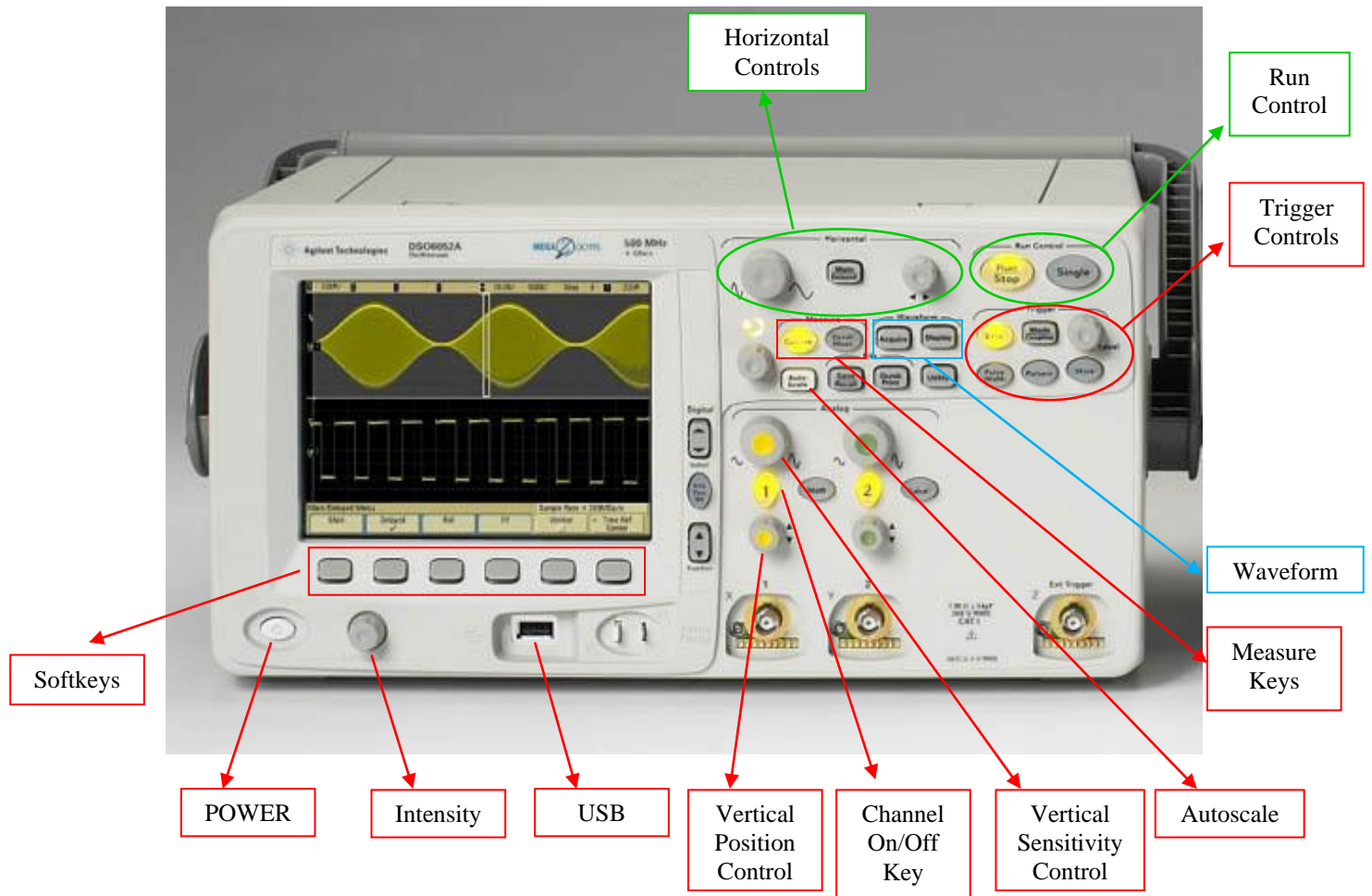


Figure 4 - Front Panel Controls.

Laboratory Experiment 4: Error Correcting Codes

1. Purpose

In this laboratory, the experimenter will be introduced to the concepts involved with error correcting codes. In particular, this experiment utilizes a Hamming Code to detect and correct single bit errors in a data transmission.

2. Background

2.1 Error Detection and Correction Codes

Whenever data is transmitted from one location to another, either between components in a single system or between separate computers, ensuring that the data arrives at its destination error free is of critical importance. As a set of data is being transmitted over some form of communication line, ambient electronic noise may cause one or more bits in the transmission to become corrupted. Often, this corruption takes the form of a bit flip, either from a one to a zero, or vice versa. The probability of one such error occurring for an individual bit in a given transmission is very small, but as the size of the transmission increases, so does the probability of a bit flip occurring somewhere in the transmission. Detecting and/or correcting these bit flips is the goal of every error detection and correction code.

With both types of coding, extra bits are added to the end of a block of data for transmission. Before the data is sent over the communication line, these bits are encoded at the sender using some agreed upon algorithm. When the data block arrives at the receiver, the extra bits may be examined using another algorithm to check if some form of data corruption took place in the transmission. Figure 1 below shows a basic block diagram for the functionality of the encoder and error detector circuits at the sending and receiving systems.

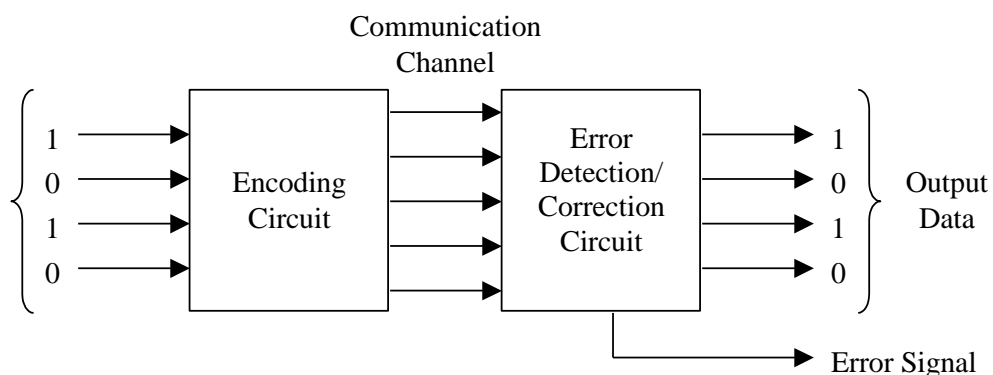


Figure 1 – Encoding and decoding of an error detection/correction code.

There are two schools of thought when it comes to error detection/correction codes. The first uses the extra bits attached to the data message to simply detect if an error occurred in the transmission. These codes are called error detection codes, because if an error is found in the transmission, the only way to get the correct data is to request a retransmission from the data source. One simple error detection code is the even or odd parity code. An even parity uses the XOR of all the bits in the data being sent to generate a parity bit. If the number of ones in the data being sent is even, the parity bit will be zero, however, if the number of ones in the data is odd, the parity bit will be one. An even parity function adds a one to the end of the data being sent if it is required to make the total number of ones sent an even number. An odd parity function works similarly, however, a one is added to the end of the data being transmitted if it is necessary to make the total number of transmitted ones odd. When the transmission is received, the total number of ones that have been received may be checked to see if it is an even or odd number. For an even parity, if an odd number of ones are received, something went wrong during the data transmission.

Since parity functions, such as the ones described above, rely on checking if the number of ones received is even or odd, multiple bit flips may go unnoticed. For instance, if a zero is changed to a one and a one is also changed to a zero in the same transmission, the error will go undetected, as the overall number of ones remains unchanged. Similarly, adding or removing ones in multiples of two will also not be detected by this code. For this reason, more complex error detection codes have been developed, such as the Cyclic Redundancy Check (CRC), which can detect multi-bit errors, but requires more than a single bit to be added for each transmission.

When it comes to error detection/correction codes, the other school of thought is why stop at simply detecting an error when it occurs. It would be ideal to not only detect an error, but also to fix the error at the receiver, so as not to require retransmission of the data that was just received. These error-correcting codes also use extra bits added to the end of a transmission that are encoded based upon the data being sent, however, the encoding of these bits is a little more complex and allows for the locations of some types of errors to be determined so that they may be corrected.

One such error-correcting code is Hamming Code, which uses parity bits interspersed throughout the data being transmitted to detect all 1-bit errors. Whereas, 1-bit errors could be detected with a single bit using a simple parity function, in order to correct the errors, more bits are required. For an N -bit message, M parity bits will be required, such that M is given by the following formula:

$$M = \lceil \log_2(N + M + 1) \rceil$$

Equation 1

By adding M bits to the transmission, there are $N + M$ possible ways to get a single bit error, and there is one way to get no error at all. The number of parity bits used must be able to represent this number of error/no error conditions; hence, M can be determined by the ceiling of the \log function given in Equation 1. When the message is received, the parity bits may be used to calculate the location of a bit flip, if one occurred during the transmission.

When using Hamming Code, each bit position in the transmitted data (including all the parity bits) is assigned a decimal value, from one onward, for the entire length of the

message. The parity bits are placed at locations corresponding to powers of two in the transmission. Each parity bit is then calculated as a standard even parity of a subset of the total transmission data. When the data is received, a standard even parity error detection circuit may then be used for the appropriate data elements and the parity bit itself to check if an error occurred in the transmission.

For example, consider a data transmission size of 8-bits. According to Equation 1 above, the number of parity bits required for the Hamming Code is four. The placement of the data bits (D_i) and the parity bits (P_i) in the resulting 12-bit Hamming Code would then be as follows:

Bit Position	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}
Coded Data	P_1	P_2	D_1	P_4	D_2	D_3	D_4	P_8	D_5	D_6	D_7	D_8

The binary bit position of each parity bit will have a one in only a single location. Each parity bit is then calculated based upon the bits in the message that also have a one in the same location as the parity bit. For instance, parity bit P_1 would be calculated based upon the message bit locations 3, 5, 7, 9, and 11. The following logic equations may be used to calculate the values of each of the parity bits:

$$\begin{aligned}
 P_1 &= B_3 \oplus B_5 \oplus B_7 \oplus B_9 \oplus B_{11} \\
 P_2 &= B_3 \oplus B_6 \oplus B_7 \oplus B_{10} \oplus B_{11} \\
 P_4 &= B_5 \oplus B_6 \oplus B_7 \oplus B_{12} \\
 P_8 &= B_9 \oplus B_{10} \oplus B_{11} \oplus B_{12}
 \end{aligned}$$

When all 12 bits of the transmission are received, the message must be checked for errors. This can be done through the generation of four check bits obtained using the equations shown above. Consider this, by XOR'ing both sides of each equation shown above with the value on the left side of the equal sign, the parity bit is added to the formula on the right, and the left side is reduced to zero. Therefore, when no transmission errors occur, all the results of these new functions will be zero. However, if an error occurs in one of the bit locations of the transmission, ones will be generated by each of the functions containing that bit location. The number that results from calculating all the check bits in this manner indicates the location of any single bit flip in the transmission. In summary, the computation of the check bits may be done according to the following formulas:

$$\begin{aligned}
 C_1 &= B_1 \oplus B_3 \oplus B_5 \oplus B_7 \oplus B_9 \oplus B_{11} \\
 C_2 &= B_2 \oplus B_3 \oplus B_6 \oplus B_7 \oplus B_{10} \oplus B_{11} \\
 C_4 &= B_4 \oplus B_5 \oplus B_6 \oplus B_7 \oplus B_{12} \\
 C_8 &= B_8 \oplus B_9 \oplus B_{10} \oplus B_{11} \oplus B_{12}
 \end{aligned}$$

Remember when the check bit result is $C_8C_4C_2C_1 = 0000$, this means that no error occurred in the transmission. However, when an error has occurred, the resulting check bit number will contain the location of the bit flip so that it may be corrected.

3. Preliminary Design

In this laboratory experiment, you will design and implement Hamming Code circuits for both four and eight bits. The eight-bit Hamming Code is discussed in detail in the Background section of this laboratory. The four-bit Hamming Code uses the following bit positioning:

Bit Position	B_1	B_2	B_3	B_4	B_5	B_6	B_7
Coded Data	P_1	P_2	D_1	P_4	D_2	D_3	D_4

The parity bits may be calculated using the following formulas:

$$\begin{aligned} P_1 &= B_3 \oplus B_5 \oplus B_7 \\ P_2 &= B_3 \oplus B_6 \oplus B_7 \\ P_4 &= B_5 \oplus B_6 \oplus B_7 \end{aligned}$$

The check bits may be calculated using the following formulas:

$$\begin{aligned} C_1 &= B_1 \oplus B_3 \oplus B_5 \oplus B_7 \\ C_2 &= B_2 \oplus B_3 \oplus B_6 \oplus B_7 \\ C_4 &= B_4 \oplus B_5 \oplus B_6 \oplus B_7 \end{aligned}$$

Each Hamming Code circuit will consist of three parts. The first is the parity generator circuit that takes in the data bits you provide and calculates the appropriate parity bits to be transmitted. In the receiving side of the transmission, you will design a check bit generator circuit, as well as an error corrector circuit that uses the check bit number to correct any single bit errors. Figure 2 below shows a block diagram of the circuit elements and interconnections required to achieve error detection and correction using Hamming Code.

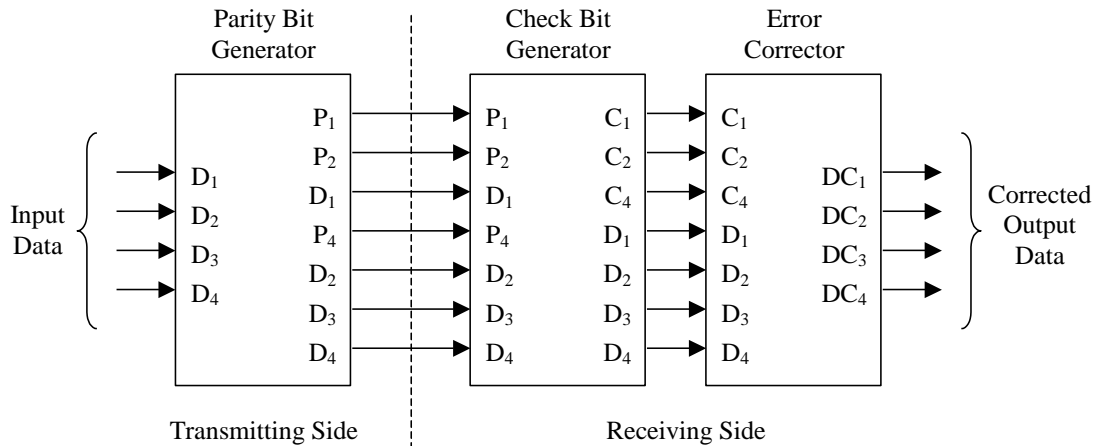


Figure 2 – Block diagram for the four-bit Hamming Code circuit.

Before coming to lab, you should:

1. Design an implementation of the four-bit Hamming Code parity generator circuit shown in Figure 2 above using standard TTL logic components. Draw a schematic diagram of your implementation, including pin numbers and package reference designators, and hand this in at the beginning of the laboratory. How many individual packages are required for your design?
2. Compose a VHDL module that will achieve the four-bit parity generation functionality you designed into the circuit in Part 1 above. Hand in a printout of this file at the beginning of the laboratory.
3. Calculate an estimate for the number of individual packages that would be required to implement the eight-bit parity generation functionality using standard TTL logic components. Due to the size of this circuit, you do not need to actually draw the schematic.
4. Compose a VHDL module that will achieve the eight-bit parity generation functionality previously discussed in the Background section of this laboratory. Hand in a printout of this file at the beginning of the laboratory.
5. Using standard TTL logic components, design an implementation of the four-bit Hamming Code check bit generator circuit shown in Figure 2 above. Draw a schematic diagram of your implementation, including pin numbers and package reference designators, and hand this in at the beginning of the laboratory. How many individual packages are required for your design?
6. Compose a VHDL module that will achieve the four-bit check bit generation functionality. Hand in a printout of this file at the beginning of the laboratory.
7. Calculate an estimate for the number of individual packages that would be required to implement the eight-bit check bit generation functionality using standard TTL logic

components. Again, due to the size of this circuit, you do not need to actually draw the schematic.

8. Compose a VHDL module that will achieve the eight-bit check bit generation functionality previously discussed in the Background section of this laboratory. Hand in a printout of this file at the beginning of the laboratory.
9. Design a circuit that will implement the four-bit error correction functionality shown in Figure 2 above. You do not need to correct errors in the parity bits of the message. You may design this circuit using standard TTL logic components, or you may implement it using a VHDL code module. If you use standard TTL components, you may wish to use a decoder circuit to get a single output from the check bit number, and you may then use XOR gates as selectable inverters. If you decide to use standard TTL logic components for your circuit, draw a schematic diagram including pin numbers and package reference designators, and hand this in at the beginning of the laboratory. If you decide to use a VHDL module for your circuit, hand in a printout of your code at the beginning of the laboratory.
10. Design a circuit that will implement the eight-bit error correction functionality for the 8-bit Hamming Code. Again, you do not need to correct errors in the parity bits of the message. You may use either standard TTL logic components or a VHDL module to implement your design, but because of its size, a VHDL module is recommended. Hand in a copy of your design at the beginning of the laboratory.

4. In the Lab

In this laboratory, the Spartan-3E Starter Kit board will be used to implement and test both the four and eight-bit Hamming Code circuits that you developed in the pre-laboratory exercises.

4.1 Four-Bit Hamming Code

1. Simulate the VHDL code module corresponding to the four-bit parity generator circuit. Verify its functionality before proceeding to the next step in the laboratory procedure.
2. Compile the four-bit parity generator module and program it to your Spartan-3E Starter Kit board for physical testing. When you are sure that the functionality of your circuit is correct, show your lab instructor its behavior.
3. Simulate the VHDL code module corresponding to the four-bit check bit generator circuit. Verify its functionality before proceeding to the next step in the laboratory procedure.
4. Compile the four-bit check generator module and program it to your Spartan-3E Starter Kit board for physical testing. Calculate the correct transmission sequence for

a four-bit data block of your choice and provide this to your circuit. The check bit number that results should be zero. Now change a single bit in the data you are feeding into the circuit. The resulting check bit number should reflect the number of the bit that you altered. When you are sure that the functionality of your circuit is correct, show your lab instructor its behavior.

5. Build or simulate the four-bit error corrector circuit and verify its functionality. Connect all three blocks of the four-bit Hamming Code circuit together on the Spartan-3E Starter Kit board and ensure that they function properly. Fully test the functionality of your circuit by purposely inserting errors into the transmission between the parity generator and the check bit generator blocks. When you are sure that your circuit functions correctly, show your lab instructor its operation.

4.2 Eight-Bit Hamming Code

6. Simulate the VHDL code module corresponding to the eight-bit parity generator circuit and verify its functionality.
7. Simulate the VHDL code module corresponding to the eight-bit check bit generator circuit and verify its functionality.
8. Build or simulate the eight-bit error corrector circuit and verify its functionality. Connect all three blocks of the eight-bit Hamming Code circuit together on the Spartan-3E Starter Kit board and ensure that they function properly. Due to the size of the data message being transmitted, you need only display the final corrected version of the eight bit data on the I/O board LEDs. Fully test the functionality of your circuit by purposely inserting errors into the transmission between the parity generator and the check bit generator blocks. When you are sure that your circuit functions correctly, show your lab instructor its operation.

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 5:

Barrel Shifters

1. Purpose

In this lab, you will be introduced to the concepts of data shifting circuits by designing and implementing a barrel shifter, using the Spartan-3E Starter Kit board.

2. Background

2.1 Barrel Shifter Circuits

Data shifting circuits are of critical importance in CPU design. They are useful for bit-mask manipulation and various other operations that would be cumbersome using other mathematical functions. When designing and building a barrel shifter, there are several functional specifications that must be considered. For instance, will the shifter move data to the left or to the right? More often than not, in order to allow the shifter circuit to be general purpose, it should support both left and right shifts of the data with which it is supplied, based upon some selection input setting. Another issue that must be considered when designing a shifter is what type of shift operation will be performed.

When a logical shift circuit moves data left or right, the data shifted out of the range of the data storage element is dropped. In addition, the empty space created in the storage element with each bit shift is filled with a pre-determined, or runtime specified, bit value that is typically zero. This type of operation is most often useful for bit-mask manipulations.

A circular shift circuit behaves similarly to the logical shift circuit; however, bits that are shifted out of one end of the storage element are fed back into the other end as inputs. This allows all of the original data to be kept, even though it is moved around. Circular shift functionality is useful for certain bit manipulations that may or may not use masks.

Finally, arithmetic shift functionality is set up to achieve very low cost multiplications or divisions by powers of two. To achieve this, a left shift operation will input zeros into the newly vacated LSBs of the data storage element. A right shift operation, on the other hand, will replicate the sign bit of the original data into the MSBs of the data storage element that are emptied during the shift operation. This is the type of shifting circuit that will be designed and implemented in this laboratory.

A final consideration when designing shifting circuits is the amount of shift the circuit will support. A single bit shifting circuit, while simple to design, will not be terribly useful, as multi-bit shift operations will require the data to be fed through the shifter several times. A shifter that will handle a variety of different shift amounts will be more complicated to design, but will ultimately be more useful.

In this laboratory, you will design an 8-bit arithmetic barrel shifter. This circuit will implement the arithmetic shift functionality and will be able to support a shift amount of anywhere from zero to seven bits. The full functionality of this circuit may be achieved using many two-input multiplexer circuits, whose inputs and control signals are connected in a specific manner. Recall that a two-input multiplexer will have two data

inputs and one select line. Depending on the value of the select line, the data fed to its single output will be one or the other of the data input values. The functionality of the multiplexer is shown in the following truth table:

A Input	B Input	Sel Input	Output
A	B	0	A
A	B	1	B

Table 1 – The truth table for a two-input multiplexer.

The following diagram shows how many of these two-input multiplexers may be tied together to create an 8-bit arithmetic barrel shifter circuit. The circuit has eight inputs and eight outputs for taking in and outputting data. There are also three shift inputs used to specify the amount of shift to be performed. Finally, a direction input is used to specify the direction of the shift, zero for left and one for right.

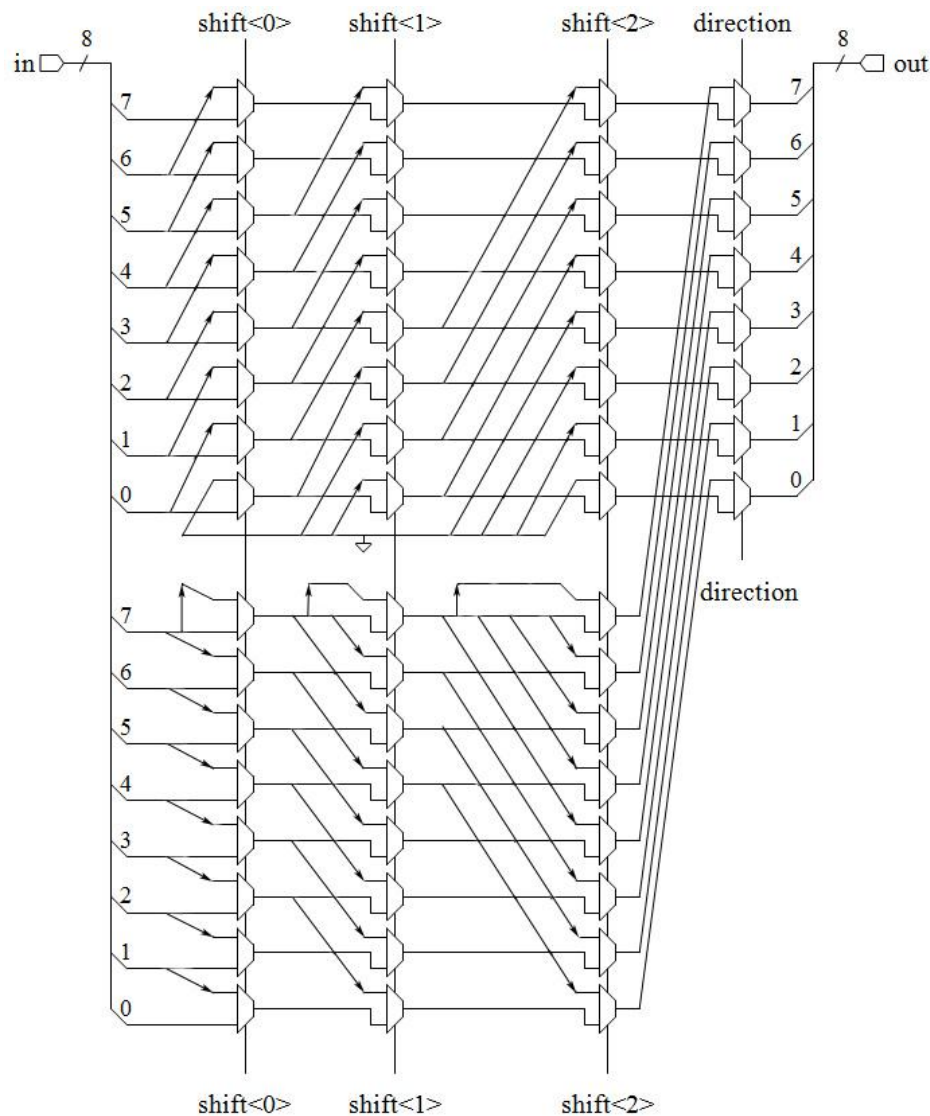


Figure 1 – Block diagram for the 8-bit arithmetic barrel shifter.

3. Preliminary Design

In this laboratory, you will design and implement an 8-bit arithmetic barrel shifter circuit using the block diagram provided in Figure 1 of the Background section of this laboratory.

Before coming to lab, you should:

1. Design a VHDL module to implement the two-input multiplexer functionality. This module will form the basis of the entire shifter circuit and, therefore, it is critical that it functions properly.
2. Using the two-input multiplexer module that you designed in Part 1 above, connect 56 of them together as shown in Figure 1 to implement the arithmetic barrel shifter functionality. In the block diagram provided, the lower input of each multiplexer block corresponds to the input that is passed through the device for a select input of zero.

4. In the Lab

1. Enter your design for the 8-bit arithmetic barrel shifter circuit into the Xilinx ISE development environment.
2. Be sure to test the functionality of your multiplexer module before testing the functionality of your full shifter circuit. If you have made an error in the construction of the multiplexer, it may not be obvious when your shifter circuit does not function properly.
3. After ensuring that your multiplexer circuit functions appropriately, test your implementation of the complete barrel shifter. Be sure to test for all of the different direction and shift amount combinations, as the 8-bit input to the circuit is of no real importance.
4. When you are sure that your barrel shifter circuit functions as you expect, program it to your Spartan-3E Starter Kit board for physical testing. Due to the number of inputs in this design, it may be somewhat difficult to use only the switches and pushbuttons on the Spartan-3E Starter Kit board to fully test your design. Make sure to take advantage of the additional switches provided by the PMOD-SWITCH units. Verify the functionality of your circuit and show the lab instructor your shifter's operation.

5. Post-Lab

Include a solution to the following question in your laboratory write-up to be turned in next week.

1. Using the block diagram shown in Figure 1 of the Background section of this laboratory as a road-map, design a block diagram that would implement the circular shift functionality for both right and left shift operations.

6. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.
3. Weste, Neil H. E.; and Eshraghian, Kamran. (1993). Principles of CMOS Design: A Systems Perspective Second Edition. Massachusetts: Addison-Wesley Publishing Company.

Laboratory Experiment 6: High-Speed Adder/Subtractor

1. Purpose

In this lab, you will be introduced to high-speed computer arithmetic by designing and implementing a look-ahead carry generation adder/subtractor circuit that may be downloaded to the Spartan-3E Starter Kit board.

2. Background

2.1 Look-Ahead Carry Generator Adders

While the ripple-carry adder circuits that you implemented in an earlier laboratory experiment are quite simple and very capable, they do have a limitation. Due to the way the single-bit adder blocks are cascaded together, in order for the most significant bit of the result to be calculated, all of the earlier bits must have already been calculated. For instance, in order for the second bit of the result to be calculated, the first bit must be calculated first. This phenomenon is due to the fact that the carry-out value of earlier bits is required in order for the current result to be calculated correctly. For calculations requiring very few bits, the additional delay required for calculating all the bits sequentially is insignificant. However, when the data being manipulated grows to a more useful size of 16, 32, or 64 bits, the delay to calculate the final result becomes significant, especially for high-speed computing applications. If a simple ripple-carry adder system were used in a high-speed processor, either addition operations would be very expensive, or, worse yet, the maximum clock rate of the processor may have to be limited.

Fortunately, research has developed look-ahead carry generation circuitry that allows all of the bits for the result to be calculated simultaneously based upon the initial carry in value. The basis for this circuitry involves some Boolean manipulations using the output functions generated by the truth table for the full-adder circuit.

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1 – The truth table for a single-bit full-adder.

The subscripts in the truth table above refer to the bit number of the current output. In other words, using the i^{th} bits of each input operand, and the i^{th} carry (carry-out of previous bit operation), the i^{th} sum value can be generated, along with the next carry

value. Essentially, this is just a change in the naming of the inputs and outputs presented in the truth table of the ripple-carry adder laboratory. From Table 1, the following logic functions can be developed for the outputs of the adder circuit block:

$$\begin{aligned} S_i &= C_i \oplus A_i \oplus B_i \\ C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \end{aligned}$$

The equation for the C_{i+1} output may then be re-written as follows:

$$C_{i+1} = A_i B_i + C_i (A_i + B_i)$$

It is already clear that the carry for the $i+1$ stage of the adder is defined recursively in terms of the carry from stage i with some additional logic surrounding it. From the equation above, some supplementary functions may be defined, allowing the C_{i+1} function to be re-written in a more efficient form. The results of this simplification are as follows:

$$\begin{aligned} G_i &\equiv A_i B_i \\ P_i &\equiv A_i + B_i \\ C_{i+1} &\equiv G_i + P_i C_i \end{aligned}$$

This simple equation for the carry out from an adder block is actually quite powerful. By recursively expanding the equation for C_{i+1} through replacing all the previous C_i 's with their equivalent formulas, the following equations for each carry output can be created:

$$\begin{aligned} C_1 &= G_0 + P_0 C_0 \\ C_2 &= G_1 + P_1 (G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \\ C_{i+1} &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} P_{i-2} \cdots P_0 C_0 \end{aligned}$$

Remember, the C_0 value required by each of these equations is the initial carry in value presented to the adder circuit from an external source. Therefore, all the carry values may be simultaneously calculated after a single gate delay that is required to produce the G_i and P_i functions for each bit pair of the operands. The high-speed adder circuit may then be constructed out of blocks with the following inputs and outputs:

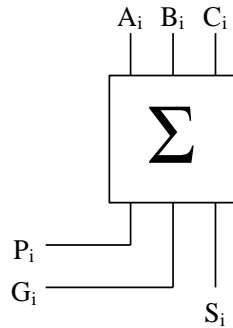


Figure 1 – Component block for the look-ahead carry adder circuit.

It should be noted, however, that nothing in life is free. While the look-ahead carry generation adder circuit does eliminate the delay incurred by cascading full-adders together, this speed comes at a cost of additional logic that must be implemented external to the blocks shown in Figure 1. This logic must implement all of the carry functions that are required for the circuit to function with as many bits as are included in the operands. The overall structure for the look-ahead carry generation adder circuit is shown in the following figure:

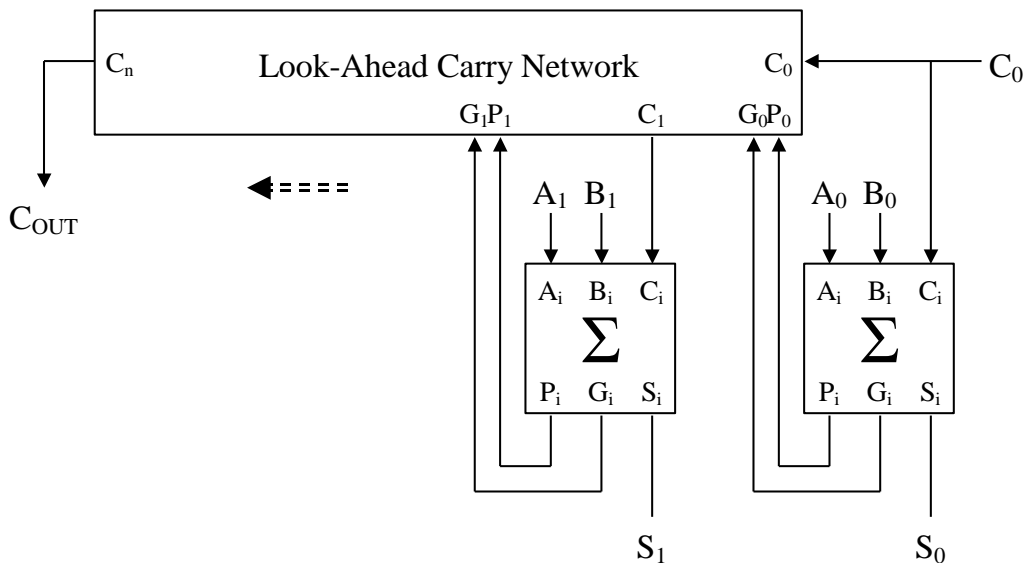


Figure 2 – Block diagram for the look-ahead carry generation adder circuit.

Finally, just as with the ripple-carry adder circuit, the look-ahead carry generation adder circuit may be made into an adder/subtractor by inserting XOR gates in the path of each of the B operand bits. These gates may be used as selectable inverters, just as

before, so when C_0 is zero, the circuit adds the two operands together, and when C_0 is one, the circuit subtracts B from A. If you do not recall how this process works, please consult the laboratory dealing with the ripple-carry adder/subtractor earlier in this manual.

3. Preliminary Design

In this laboratory, you will design and implement a 4-bit look-ahead carry generation adder/subtractor circuit using the principles discussed in the background section of this laboratory. After designing and testing this circuit, you will then cascade two of them together to create an 8-bit adder/subtractor. This circuit will be a hybrid of the ripple-carry and look-ahead carry generation circuits, in that the four most significant and least significant bits will be calculated using the look-ahead carry generation techniques, but in between the two blocks, there will be some sequential overhead incurred due to the cascading.

Before coming to lab, you should:

1. Design a VHDL module to implement the required functionality shown in Figure 1 of the Background section of this laboratory. This module should also include the XOR gate and all connections necessary for selectively inverting the B operand input. This will allow your final circuit to be an adder/subtractor. In order to invert the B operand bits as necessary, your module must include an input for the C_0 value.
2. Using Figure 2 of the laboratory as a template, draw a block diagram for the four-bit look-ahead carry adder/subtractor circuit. Since the block diagram provided only implements an adder circuit, some additional connections are needed in your circuit. In order to invert the B operand bits as necessary, each adder/subtractor module must have a C_0 input, and the external C_0 for the circuit must be provided to each module.
3. Using the VHDL module you created in (1) above, construct a VHDL version of the four-bit look-ahead carry generation adder/subtractor, for which you developed a block diagram in (2). Be sure to include all of the equations necessary to calculate each of the necessary carry values, and feed the appropriate data to each of your adder modules.
4. Draw a block diagram for cascading two of your 4-bit look-ahead carry adder/subtractors together to make an 8-bit adder/subtractor. You do not need to implement look-ahead carry generation functionality between the two modules.
5. Write a VHDL module that cascades two of your 4-bit look-ahead carry adder/subtractor circuits together. In order to allow for the subtraction operation to function properly, your second module will need an external input for C_0 and your first module will need an external output for C_0 . These may then be tied together to allow the B operand bits to be appropriately inverted.

4. In the Lab

4.1 4-Bit Look-Ahead Carry Adder/Subtractor

1. Enter your design for the four-bit look-ahead carry adder/subtractor into the Xilinx ISE development environment.
2. Simulate your design to ensure it functions properly in both the addition and subtraction modes. If something is wrong in simulation, fix the problem and simulate your design again. When you are satisfied that your design functions properly, download it to the Spartan-3E Starter Kit board for testing.
3. When you are satisfied that your design functions properly, show the lab instructor your working circuit before proceeding to the next section of this laboratory.

4.2 8-Bit Look-Ahead Carry Adder/Subtractor

4. Enter your design for the eight-bit look-ahead carry adder/subtractor into the Xilinx ISE development environment.
5. Due to its size and the limited number of switches on the Spartan-3E Starter Kit board, you will only simulate your design for the eight-bit adder/subtractor. Simulate several instances of addition and subtraction, and when you are satisfied that your design functions as expected, show the lab instructor your simulation results.

5. Post-Lab

Include solutions to the following questions in your laboratory write-up to be turned in next week.

1. Design a full eight-bit look-ahead carry generation adder circuit. You only need to develop the necessary functions to implement the design, and draw a block diagram showing how the entire design fits together.
2. Design a circuit to determine if an overflow occurred during an addition or subtraction. Your circuit should utilize the operation being done, as well as the signs of each of the operands and the result to determine if an overflow occurred.

6. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 7:

Sequential Logic Design and Finite State Machines

1. Purpose

In this lab, you will learn the skills required to design and program sequential circuits that may be uploaded to the Spartan-3E Starter Kit board. Students will design and build a finite state machine both through utilizing D flip-flop elements and entering the state diagram directly into the Xilinx ISE development package.

2. Background

2.1 State Diagrams and State Machines

State diagrams are often used for sequential logic design because they provide a simple, visual method to see the overall functionality of the system as a whole. Each state in a finite state machine (FSM) is used to achieve some desired functionality, or is used as a synchronization point between functional state traversals. For any given FSM, a state diagram may be composed showing the next state and associated outputs of the system based upon the current state and inputs presented to the system.

In all state diagrams, the next states that are possible given a current state may be determined by following the transition arrows. Some transitions are taken only when a certain input combination is presented, while others may be taken all the time. The latter type of transition occurs unconditionally at the next clock cycle.

The outputs from the state machine also come in two varieties. In the first, the outputs from the system are only a function of the current state in which the system resides. As long as the system resides in a given state, these unconditional outputs will always have the same value. This type of output may be included in a state diagram by labeling the state bubble with each output and its associated value. When an FSM contains only unconditional outputs, it is called a Moore Machine. Figure 1(a) shows a Moore Machine implementation of a two-bit binary counter with a carry output (C) signal. Before the counter cycles back to zero, it asserts the carry output to allow for potential cascading of multiple counters together.

Whereas, some outputs depend only on the state in which the FSM resides, others may depend upon the input values presented while the system is in a given state. These conditional outputs may not be placed directly in the state bubble in a state diagram because the output from a given state may be different depending on the inputs presented. To list this type of output in a state diagram, a transition arrow out of a state is labeled with an input value and the associated output value it should produce. A transition must be added coming out of a given state for each combination of input conditions that may arise. When an FSM contains only conditional outputs, it is called a Mealy Machine. Figure 1(b) shows a Mealy Machine implementation of a two-bit counter with a carry output and enable input. Depending on the state of the enable input, the FSM will either progress to the next state with the next clock tick or it will stay where it is. Similarly, the

state of the carry output also depends on whether or not the system is enabled for counting.

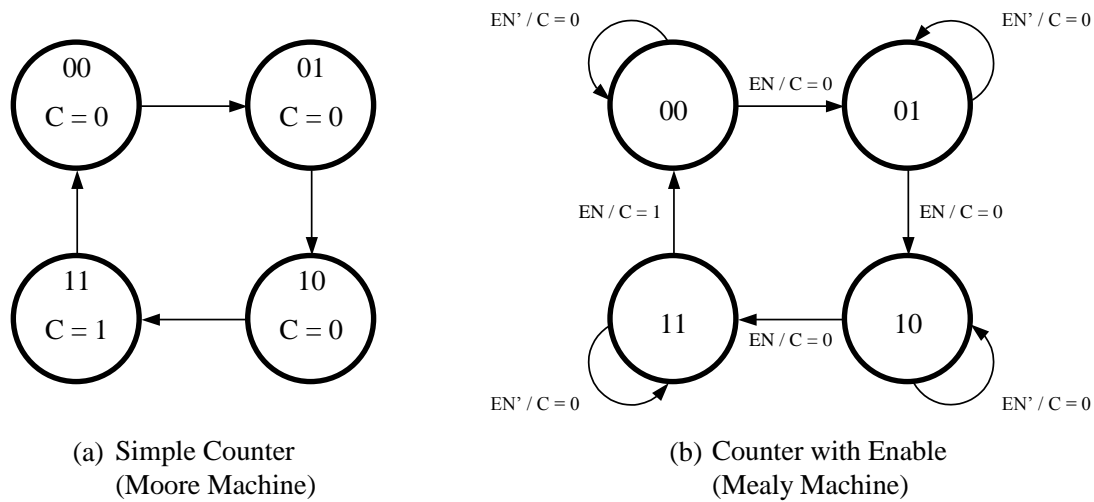


Figure 1 – Moore machine and Mealy machine implementations of the two-bit counter.

In Figure 1, each state in the diagram is labeled with a bit pattern corresponding to the output values of the two flip-flops required to implement the design. More often than not, however, state diagrams are composed in which the states are given symbolic names. These names are often chosen to reflect the functionality that is achieved within a given state, making it easier for an individual to discern the full functionality of the system. Figure 2 shows the same state diagram as was presented in Figure 1(b), but the binary state names are replaced with the names S0, S1, S2, and S3.

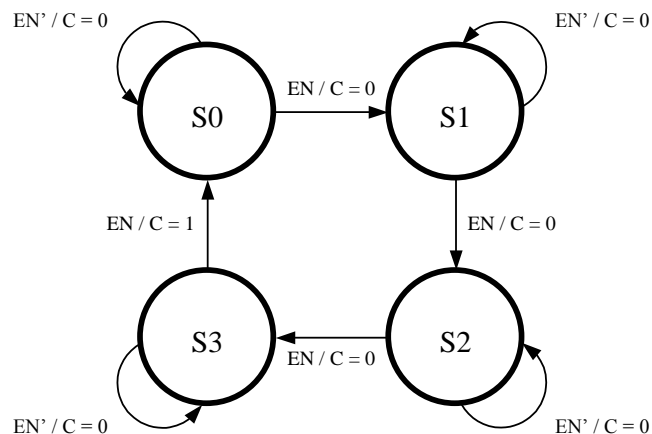


Figure 2 – State diagram with symbolic state names.

When a diagram, such as the one shown in Figure 2, is to be physically implemented, the symbolic state names must be assigned unique binary codes. In the state assignment process, each symbolic state name is associated with a binary state code that will later correspond to the data stored in the flip-flops of the system at any given time slice. The

current state of the system will then be determined by this flip-flop data storage. Depending on the state code selected for each state in the FSM, different implementation costs can be incurred. For the FSMs developed in this laboratory, however, these differing implementation costs based upon state code assignment will not be critical enough to be of concern.

2.2 Implementing Finite State Machines

Implementing a physical manifestation of a state diagram is fairly simple, and may be done through the following steps:

1. Develop a state transition table from the supplied state diagram. This is a kind of truth table for the state machine, showing the possible input combinations for a given current state along with the corresponding next state, and output values that will result. Figure 3(a) shows an example of a state transition table that was developed for the state diagram shown in Figure 2.
2. Decide upon the number of flip-flops that will be required to implement the design. This can be found by the number of bits that are required to list the binary value for the number of states in the system. This is because in order to represent N system states, at least N distinct combinations of flip-flop values must be possible, requiring $\lceil \log_2 N \rceil$ distinct flip-flops. After choosing the number of flip-flops to be used, assign each state a unique binary number with the same number of bits as flip-flops in the system. A new version of the state transition table may then be generated containing these state code assignments. An example of a state code assignment for the state diagram in Figure 2 is shown in Figure 3(b).
3. Choose the type of flip-flops that you will use to implement your design. More complex flip-flops, like JK flip-flops, may reduce the amount of external logic needed to control the flip-flops themselves. However, since D flip-flops are easier to use, they will be utilized in this laboratory. Remember, with a D flip-flop, the value stored at a clock cycle is the same as the input value into the flip-flop at the rising edge of the clock.
4. Compose an excitation table for the system. This table will contain the necessary inputs for each flip-flop that will produce the correct next state at the next clock cycle. These required input values may then be treated like any other circuit output and logic functions may be developed to pass the correct input values to each flip-flop based on the current state and system inputs. Remember, for D flip-flops, this table is equivalent to the state transition table created in Step 2. For more complex flip-flops, a new table must be generated for the required flip-flop input values.
5. Derive logic functions for each flip-flop input, and for each system output signal based upon the excitation table you generated in Step 4. Remember, when using D flip-flops, the excitation table will be similar to the one shown in Figure 3(b).

IN	Present State	Next State	OUT
0	S0	S0	0
1	S0	S1	0
0	S1	S1	0
1	S1	S2	0
0	S2	S2	0
1	S2	S3	0
0	S3	S3	0
1	S3	S0	1

(a) State Transition Table

IN	Q0	Q1	D0	D1	OUT
0	0	0	0	0	0
1	0	0	0	1	0
0	0	1	0	1	0
1	0	1	1	0	0
0	1	0	1	0	0
1	1	0	1	1	0
0	1	1	1	1	0
1	1	1	0	0	1

(b) Excitation Table

Figure 3 – Tables required when performing sequential logic design.

2.3 The Turn Signal FSM

Now that the basics of sequential logic design have been discussed, the FSM that will be designed in this laboratory may be introduced. Figure 4 below shows a block diagram of the system that will be designed, and how its outputs may be mapped to a real-world application. The back of a car has three lights, one on each side and one in the center. These lights will be used for turn signals, as well as for hazard flashers, and therefore, each will be tied to an output of the FSM. In addition, the FSM has three inputs corresponding to the right and left turn signal selections, as well as a system clock.

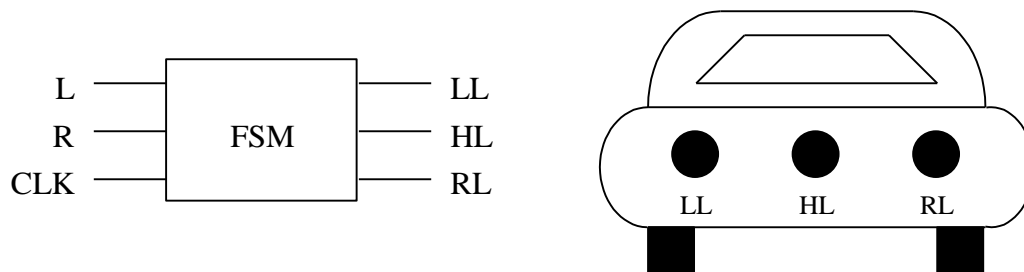


Figure 4 – Block diagram and real world mapping for the Turn Signal FSM.

When neither of the turn signal control inputs is asserted high, none of the lights for the car should glow, and the system should stay in this idle state. If a left turn signal is requested by asserting the left input while keeping the right input low, the left light of the car should flash on and off with the same frequency as the system clock signal. Similarly, if a right turn signal is requested by asserting the right input while keeping the left input low, the right light of the car should flash on and off with the same frequency as the system clock signal. Finally, asserting both the right and left inputs simultaneously indicates a hazard condition. In this situation, the left and right lights should both be lit while the hazard light is not for one clock cycle. Then the hazard light should light while the others turn off for the next clock cycle. Finally, all lights should turn off for one additional clock cycle. All of these output conditions should repeat for as long as the input requirements are satisfied.

A state diagram for this system is shown in Figure 5 below. Each state in the diagram is given a symbolic state name to make it easier to determine what is going on for each combination of input conditions. In addition, this system may be implemented as a Moore Machine since the output light pattern depends only on the current state of the system. The binary number shown in each state of Figure 5 may be used, not only for the flip-flop values indicating the current system state, but may also act as the system outputs tied directly to the three lights on the car. This method of state assignment is called “output coded” state assignment, because the system state may double for the system outputs directly. This type of state assignment only works for Moore Machines because the outputs of the system only rely upon the current system state. Transitions in the diagram are either labeled with the input conditions that must be true for the transition to be taken, or are labeled with a one. Transitions labeled with a 1 are always taken at the next clock cycle, regardless of the inputs into the system.

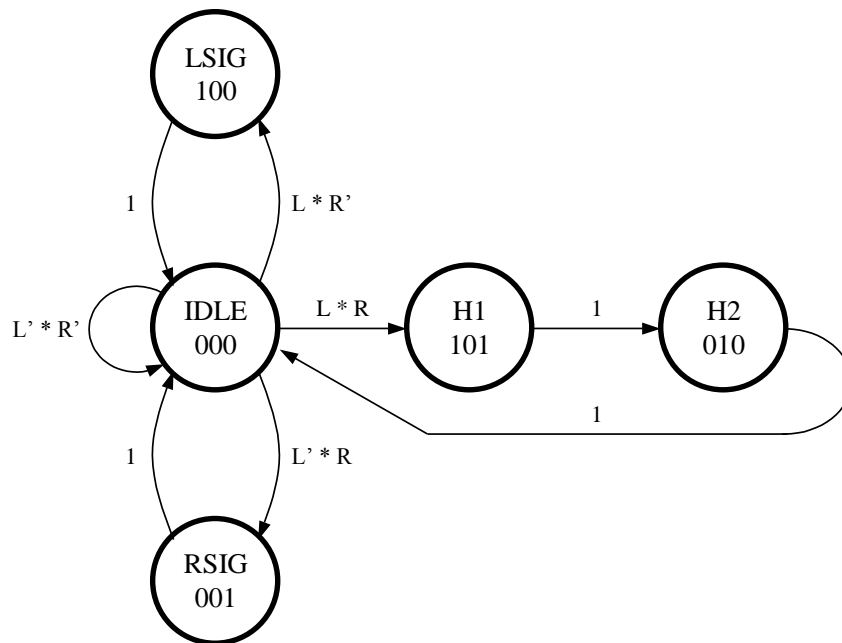


Figure 5 – State diagram for the Turn Signal FSM.

3. Preliminary Design

The preliminary design for this laboratory consists of creating an implementation for the Turn Signal FSM. The counter, which was discussed as part of the background section of this laboratory, was only used to discuss the concepts of sequential logic design and will not be implemented in the laboratory.

Before coming to lab, you should:

1. Generate a state transition table for the Turn Signal FSM using the state diagram provided in Figure 5.
2. As can be seen from the state code assignments in the state diagram, this circuit will require the use of three flip-flops. For ease, this circuit will be implemented using D flip-flops, therefore, the flip-flop inputs may be named D0, D1, and D2, and the outputs may be named Q0, Q1, and Q2. With this in mind, write an excitation table for the Turn Signal FSM including the input signals L and R, as well as the flip-flop inputs and outputs. Keep in mind that each car taillight may be tied to one of the flip-flop outputs in the final design.
3. Generate the excitation logic functions for each flip-flop in the Turn Signal FSM. These functions should use the system inputs and the current state to derive the input signal value to each flip-flop that will be stored at the next clock cycle. Bring these functions and the state diagram for the FSM to the laboratory for implementation on the Spartan-3E Starter Kit board.

4. In the Lab

The Xilinx ISE development environment allows for sequential circuits to be entered in several different formats, depending on the nature of the problem at hand. One simple entry method uses standard VHDL code and the modular design techniques with which you should now be familiar. In this method, a VHDL module must first be created to implement the functionality of a flip-flop of the desired type. This module may then be used in conjunction with logic equations for the inputs and outputs, just as though a circuit were being made out of physical components.

While this method may be used for any situation that you will encounter, for large problems, it may be tedious or undesirable to reduce a state diagram to this level for implementation. To accommodate this situation, the ISE development environment allows sequential designs to be directly entered at the state diagram level. The VHDL code to implement the state diagram is then automatically generated for you. This laboratory procedure will discuss both ways to enter the turn signal FSM design into the development environment.

Sequential Logic Design Using Flip-Flops

1. The first part of this procedure is concerned with creating the D flip-flop module to be used in the FSM implementation. To that end, create a new project in the ISE Project Navigator and add a VHDL module file to it. The input ports for this module should include a D input and a CLK input, and the outputs should include a Q output and optionally a Q_L output for the inverse of Q.
2. The key to creating any flip-flop in VHDL is edge detection on the clock signal. Edge detection can be achieved quite simply within a process you create in your VHDL

file. If you do not remember how to create a process, please see the laboratory procedure for the BCD code conversion in this manual. Both the D and CLK input signals will be used within the process, so note them accordingly in the process declaration.

3. Edge detection is then quite simple and may be achieved with an IF statement within your new process. The first condition that must be checked is the level of the CLK signal. For a rising-edge triggered flip-flop (which we will use) the CLK should be high in order for the flip-flop to trigger. This condition alone is not enough, however, so an additional condition must also be checked. The IF statement must also check for an *event* on the CLK input signal in order to achieve true edge-triggered functionality. An event will be triggered on a given signal for a number of different reasons, including a change from low-to-high or high-to-low. To check for an event on a signal, enter the signal name followed by a single quote and the word “event,” as follows:

signal_name'event

The following code illustrates the proper format for an edge-detection IF statement, and in fact, is the code required to produce an edge-triggered D flip-flop:

```
process (CLK, D)
begin
    if CLK = '1' and CLK'event then
        Q <= D;
        Q_L <= not (D);
    end if;
end process;
```

4. When you finish entering your flip-flop VHDL file, synthesize your code to eliminate any syntax errors and simulate your design to ensure that it works appropriately.
5. When it comes to sequential logic design, the only other issue that must be dealt with is the clock input when assigning pin numbers. Pin C9, is mapped to a 50 MHz crystal oscillator on the board. This crystal oscillator is too fast to be useful in its raw state. To remedy this, Appendix B of this laboratory manual contains code modules for a 1 Hz clock generator and a selectable frequency clock generator. By adding one of these modules to your design and utilizing the 50 MHz crystal oscillator signal, you will be able to generate a slower clock frequency for your sequential designs. Assign pin numbers to all the inputs and outputs of the D flip-flop, and program it to your test board to ensure that it works as expected.
6. **THIS STEP IS CRITICAL.** After testing your D flip-flop on the Spartan-3E Starter Kit board, **REMOVE** the clock divider module that you added to it to produce the slower clock frequency. You should save the D flip-flop version that takes in a clock signal from the outside and uses that directly for its edge-detection. This is important when it comes to simulating higher-level designs that use the flip-flop you

just generated. If you were to leave in the clock rate division, when a design that uses this flip-flop is simulated, upwards of 50,000,000 clock cycles would have to go by before you would see the states changing!

7. When you have verified that your D flip-flop design works appropriately, use this module in conjunction with the logic equations you developed for the turn signal FSM to write a VHDL file that achieves the required functionality. Create a new project and enter this file for testing. Be sure to tie the clock input for the FSM to the clock inputs of all the flip-flop elements you place in your design. At this point, do not include any of the clock divider modules in your FSM design. One of these modules may be included after your design is simulated and before it is programmed to the Spartan-3E Starter Kit board. In order to use the output values of the flip-flops as inputs to logic equations, you will need to create internal signals that you can copy the values into. These signals can then be used in logic equations to be fed into the inputs of the flip-flops. Also, do not forget to add the VHDL file for the D flip-flop you generated to this new project, so that the code will be accessible.
8. Synthesize the Turn Signal FSM design, and simulate it to ensure that it functions as you expect. Remember, when it comes to assigning pin numbers to the input and output ports of your turn signal FSM, be sure to assign the clock port to the pin discussed earlier.
9. When you are sure that your design functions as you expect, you may add one of the clock divider modules to the design and use its output as the clock inputs to your flip-flops. You will need to re-compile your design and generate the programming file for download. When you have completed your design, download it to your Spartan-3E Starter Kit board and show its functionality to your lab instructor.

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 8: Traffic Light Controller

1. Purpose

Typically, traffic light control is a field dominated by micro-controllers. Very complex timing of these lights can be handled with some simple software running on off-the-shelf products. For very simple traffic lights, however, it may be possible to implement their timing functionality with simple sequential logic circuits. In this lab, you will design and implement two traffic light controller circuits using the Spartan-3E Starter Kit board.

2. Background

For complex, modern traffic lights, often, the use of a micro-controller to handle all the timing functionality of the light is essential. Coordinating the three main signal lamps for each direction, left and right turn arrows, and walk and don't walk signals is not an easy task. It may also be beneficial to alter the timing of the light based upon traffic sensors and time of day. Handling all this functionality is simply too daunting a task for simple logic circuits. In addition, the programming capabilities of a micro-controller allow the timing to be altered relatively easily when necessary. With this said, there are some simple applications in which the cost of developing a micro-controller system and its corresponding software would simply not be justified. For these situations, a simple sequential logic circuit may be constructed to provide all the required functionality.

2.1 Simple Two-Way Intersection

The first circuit that will be designed for this laboratory experiment is one for controlling a traffic signal at a very simple intersection. Consider an intersection of a north/south road and an east/west road, as shown in Figure 1 below.

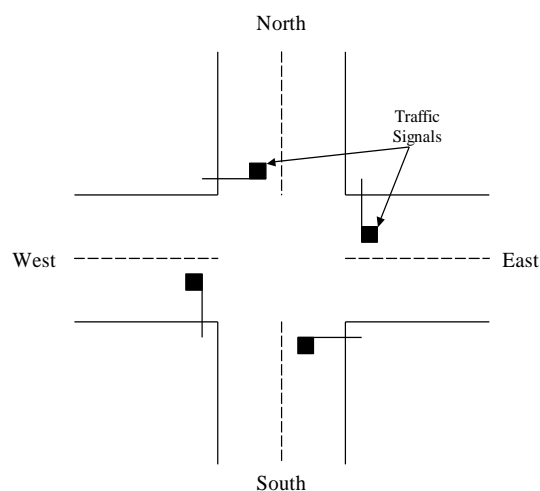


Figure 1 – Illustration of the intersection for the first traffic light controller.

For this first, very simple problem, each traffic signal only has three lamps: green for proceed, yellow for changing, and red for stop. A timer alone governs the changing of these lights and no additional sensor information is obtained regarding traffic flow or any other operating conditions. In addition, consider that the north/south road is busier than the east/west road; therefore, it is desirable to give longer green lights to the north/south road. The north/south road may be given ten time-unit green lights, while the east/west road is only given five time-unit green lights (for the purposes of this laboratory, a time-unit may be assumed to be the length of one clock cycle of a chosen frequency). Just as with a typical traffic light, when one direction is given a green light, the other direction must be given a red light. For the transition from green to red, a yellow light is given to the directions that had a green light previously, for a duration of one time-unit. Finally, when the yellow light transitions to red, all four directions are provided with a red light for one time-unit, allowing for a margin of safety in the event someone decides to run the new red light. This functionality is summarized in the following flowchart.

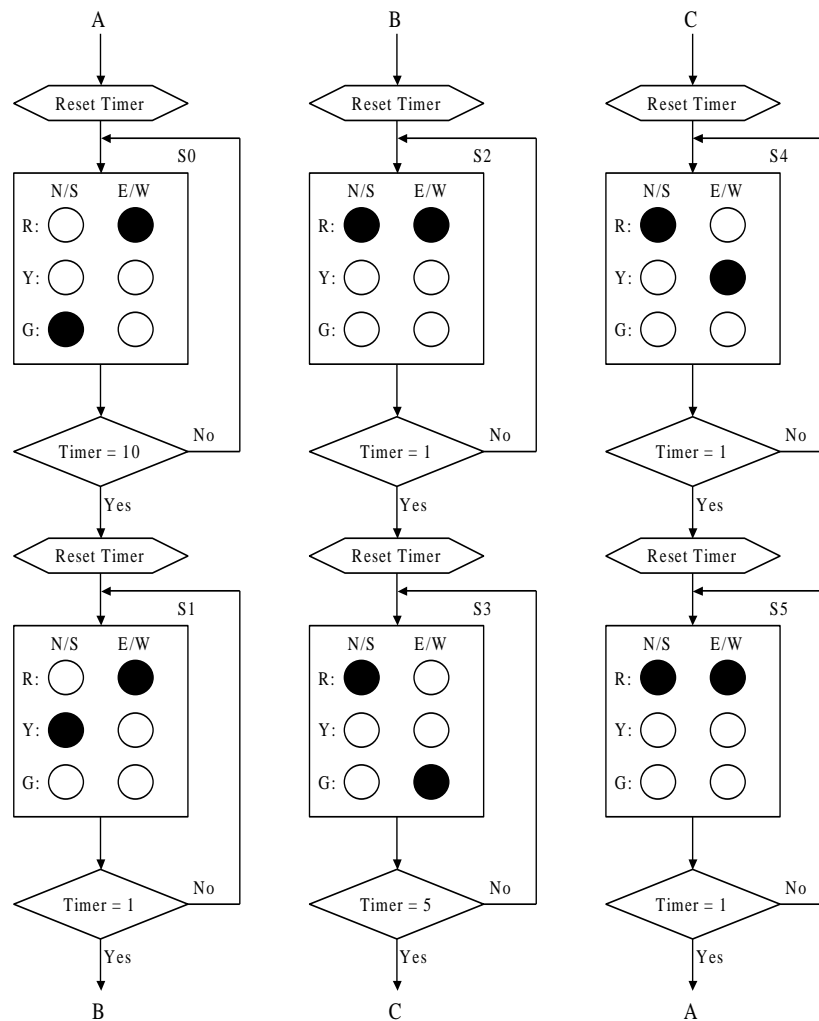


Figure 2 – Flowchart for the simple traffic light controller.

2.2 Two-Way Intersection with Traffic Sensors

For the second circuit to be developed in this laboratory experiment, consider that the previous traffic light no longer satisfies the needs of the traffic patterns on these two streets. If the traffic flow on the north/south street increases drastically while the traffic on the east/west street remains very low, the timing that was provided in the previous section could produce undesirable delays on the north/south road. If there are no cars waiting for the red light on the east/west street, there is no reason to change the north/south signal from green to red. When a car arrives at the east/west traffic light, the same timing presented in the previous section may be initiated. In order to achieve this functionality, automobile sensors are inserted at the intersection as shown in the following figure.

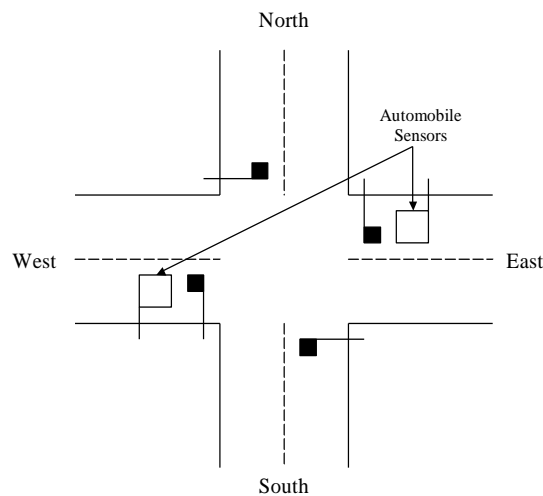


Figure 3 – Illustration of the intersection with the automobile sensors added.

The addition of the automobile sensors allows the traffic light controller to adapt to changing traffic patterns. This will allow the north/south street to have an unrestricted traffic flow in the event that no cars are waiting at either of the cross street traffic signals. The functionality of these automobile sensors, in conjunction with the traffic light timing, is summarized in the following flowchart.

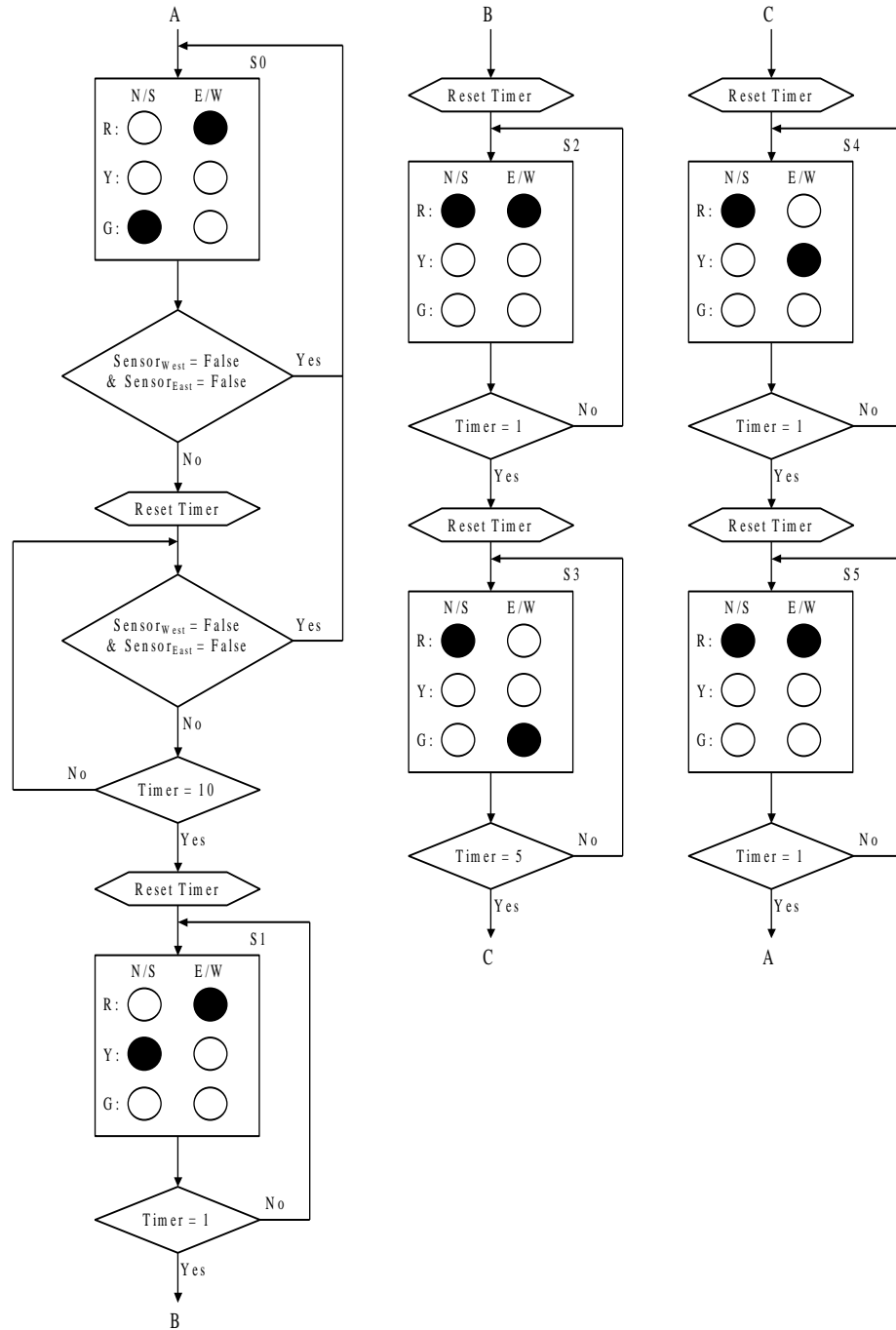


Figure 4 – Flowchart for the traffic light controller including automobile sensors.

3. Preliminary Design

In this laboratory, you will design and implement both of the traffic light controller circuits presented in the Background section of this laboratory. In your implementation, you may use either of the 50-MHz clock divider modules presented in Appendix B of this laboratory manual. You may find it desirable to utilize the selectable frequency module, as you will not be forced to wait for five and ten-second timing cycles to complete.

Before coming to lab, you should:

1. Design a VHDL implementation of the traffic light controller presented in Background Section 2.1 of this laboratory experiment. Do not include a clock divider module in your code yet, as you will need to simulate your design before implementing it. In order to convert the flowchart provided into functional VHDL code, you may find it helpful to first develop a state transition diagram encompassing the functionality of the flowchart. This step is not necessary, however, and your design may be developed directly from the flowchart itself. In addition, you will probably find the use of variables in your VHDL code useful. Variables may be declared within VHDL processes (like those used for IF statements) and use the following syntax:

variable variable_name : type := initialization_value;

Any variable declarations are placed between the process declaration statement and the “begin” statement for the process. The variable type may be: integer, real, bit, or any number of other more complex data types that will not be necessary for this laboratory. When assigning a value to a variable, the “:=” assignment operator must be used. Finally, standard mathematical operations, such as addition, subtraction, multiplication, and division may be performed on variables, as they are treated in a manner similar to C++ variables. For an example of simple variable use, consult the clock divider code modules in Appendix B of this laboratory manual; both utilize variables to store a count.

2. Design a VHDL implementation of the second traffic light controller presented in Background Section 2.2 of this laboratory experiment. Again, do not include a clock divider module in your design yet, as you will need to simulate it before programming it to the Spartan-3E Starter Kit board.

4. In the Lab

4.1 Two-Way Intersection with No Sensors

1. Enter your design for the first two-way traffic controller into the Xilinx ISE development environment.
2. Simulate your design to ensure the timing of the lights is appropriate and fits the specifications of the flowchart in Background Section 2.1 of this laboratory experiment. If your design does not simulate properly, correct all inconsistencies before continuing to the next section of this laboratory procedure.
3. Include one of the 50-MHz clock divider modules from Appendix B of this laboratory manual in your design, and connect its output to the clock signal of your traffic light controller. Either clock divider module may be used with this design; however, the selectable frequency module may make physical testing of your circuit less time consuming.
4. Program the Spartan-3E Starter Kit board with your design and physically test your circuit. When you have verified that your design functions properly, show the lab instructor its behavior before moving on to the next step in this laboratory.

4.2 Two-Way Intersection with Sensors

4. Enter your design for the second two-way traffic controller into the Xilinx ISE development environment.
5. Simulate your design to ensure the timing of the lights is appropriate and fits the specifications of the flowchart in Background Section 2.2 of this laboratory experiment. Be sure to simulate all combinations of sensor inputs. If either sensor or both are triggered, the traffic light timing should be initiated. The only way for the timing process to be halted is for both sensors to be turned off. If your design does not simulate properly, correct all design problems before continuing on to the next procedure in this laboratory.
6. Include one of the 50-MHz clock divider modules from Appendix B of this laboratory manual in your design, and connect its output to the clock signal of your traffic light controller.
7. Program the Spartan-3E Starter Kit board with your design and physically test your circuit. When you have verified that your design functions properly, show the lab instructor its behavior before completing this laboratory.

5. Post-Lab

Include a solution to the following question in your laboratory write-up to be turned in next week.

1. Design an implementation of the traffic light controller circuit without traffic sensors using D flip-flops with asynchronous reset capabilities. In order to do this, it may be beneficial to design a timer circuit based upon a shift register. After a reset, a signal may be shifted through the shift register, and tap points may be included in the design to provide a signal for each of the required timing events. Begin this design by creating a schematic diagram for this timer, or a timer of your choosing, keeping in mind that the clock frequency used is the same as the base time unit for the system. Next, generate the state transition table, excitation table, and a schematic diagram for your main controller circuit, obeying the flowchart in Figure 2 from the Background section of this laboratory. After completing this, generate the output functions controlling the traffic signals based upon the current state of your controller circuit. You may simply develop the functions, and do not need to draw the schematic diagrams representing them.

6. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 9: Data Encryption Using LFSRs

1. Purpose

This laboratory experiment will introduce some concepts of data encryption and cryptography that may be implemented using an FPGA and the Spartan-3E Starter Kit board. The student will then be given a set of encrypted ASCII text and will decrypt it using the methods specified in this experiment.

2. Background

First, it should be stated that the cryptography method presented in this lab is a very simplified version of what is used in most secure data communications today. In general, the more complex the encryption protocol, the harder it will be for an external entity to crack the cipher and decode a data transmission. Implementation of standard cryptographic algorithms, while fairly simple and straightforward with a computer, is not a simple task when given only an FPGA. For this reason, the cryptographic algorithm presented in this laboratory will focus only on encrypting eight-bit blocks of data using an eight-bit key sequence.

2.1 Basics of Cryptography

The general idea behind cryptography is a very simple one, and revolves around the bitwise XOR function. At the sender, a cryptographic key sequence, K , is generated by an encryption algorithm, and is combined with the plain text, P , that is to be encrypted. The resulting cipher text may then be transmitted to the receiver, which must then use the same key sequence to retrieve the plain text information from the cipher. This behavior is possible because of the following relation:

$$P \oplus K \oplus K = P \oplus 0 = P$$

The entire encryption/decryption process is shown in the following diagram:

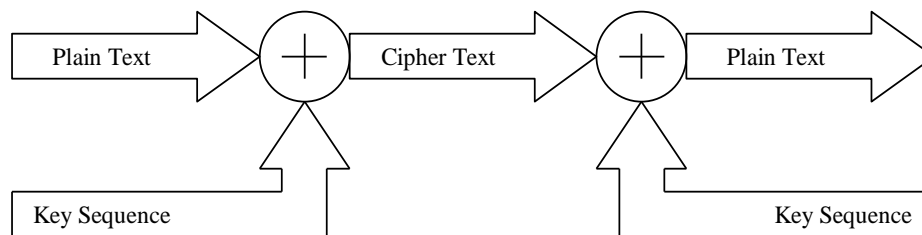


Figure 1 – The basic encryption/decryption process.

While this process seems quite simple, as usual, the details present the greatest hurdle to overcome. If a fixed key sequence were always to be used for a given encryption

algorithm, it would be a fairly simple matter to crack that key given enough time. The algorithm would then be rendered useless because the key sequence is known and does not change. Several things may be done to increase the security level of an algorithm including: increasing the key length, changing the key during transmission, applying several keys to a single data block, and a variety of other techniques. In this laboratory, the encryption algorithm used will change the key value after each plain text character is encoded or decoded. With this technique, it must be ensured that both the sender and receiver are using the same key at any given time, and that they are using a key sequence that is not easily deciphered.

2.2 Generating Encryption Keys

In general, it is desirable for the series of keys used to encrypt or decrypt cipher text to be a pseudo-random bit sequence. For instance, using a simple counter circuit, while an easy way to generate changing key values, produces an encryption that is easy to crack. Once one key value is discovered, all the following key values are automatically known. Using a random sequence of keys to encrypt data means that even if one key is discovered, all the other data will remain secure as none of the other keys are automatically known. In reality, a true random sequence is impossible to produce, however, a fairly good simulation may be achieved using a linear feedback shift-register (LFSR).

An LFSR, with several strategically placed XOR gates, will produce a fairly random sequence in its bit positions. A problem with this design arises if the LFSR happens to contain all zeros. In this situation, the use of XOR logic will never change any of the bits, and the device will be stuck in that state. To avoid this problem, XNOR logic may be used to create the next state of the shift register. Before going on, it would be beneficial to show a diagram of one such shift register.

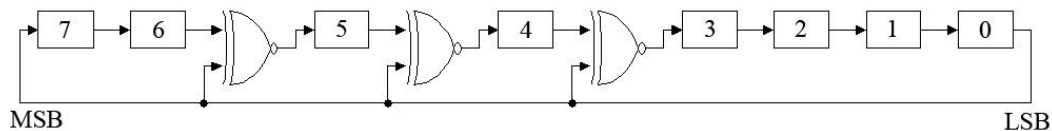


Figure 2 – An LFSR design used to generate key values.

As is shown in Figure 2 above, the feedback relationship of the LFSR may be specified using a polynomial representation. For a given length LFSR, there are multiple polynomials that provide reasonable, but different, random sequencing. Therefore, the choice of this polynomial adds an additional degree of security over simply the use of a random key sequence. The design of the LFSR shown in Figure 2, with its characteristic polynomial, will be used throughout this description, and will be implemented later in the laboratory.

In order to ensure that the sending and receiving devices are using the same key sequences to manipulate data, two things must be specified for a given data transmission: the starting value of the LFSR bits, and the number of shifts between sampled key values. The starting values of the LFSR bits provide a reference point from which all the keys will be generated. Since the sender and receiver devices are both using the same key

generation polynomial, they will be guaranteed to get the same key sequence if incremented correctly. Allowing the number of LFSR shifts between sampled key values to be greater than one provides an additional degree of security over simply using a single increment. This way, even if an external entity knows the encryption polynomial being used and the starting key, they may still be unable to decrypt a transmission if they do not know the correct increment value being used.

For the purposes of this laboratory experiment, the encryption circuit that will be implemented must allow the initial LFSR bit values to be set to a specified initial key. In addition, a constant increment value of only one shift between key samplings will be used throughout this laboratory. While this design will not benefit from the added degree of security that stems from allowing a shift greater than one between samplings, it will be less complex to implement in the laboratory.

3. Preliminary Design

In this laboratory experiment, each data block to be encrypted will consist of a single byte. To give the data some meaning, this byte will contain an ASCII character from one of the two tables contained in Appendix E. Your job, for this experiment, is to construct an encryption/decryption circuit using the LFSR shown in Figure 2. When an *LFSR_Set* signal is provided to the LFSR circuit, the byte value set as input to the circuit should be stored in the bits of the LFSR. When an *encrypt/decrypt* signal is provided to your circuit, it should shift the bits of the LFSR once, take the bitwise XOR of the LFSR and the input byte, and display this result to the user. Since the encryption and decryption functions both utilize the bitwise XOR functionality, this circuit will act as both an encryption and decryption device depending on whether the input data byte is plain text or cipher text.

Before coming to lab, you should:

1. Write a VHDL code module that implements the required functionality of the encryption key LFSR. This LFSR must have the ability to be set to an initial value, and to increment by a single shift when appropriate signals are provided. Depending on how you decide to implement your LFSR, you may find the FOR loop in VHDL a useful tool. This loop must be entered within a process, and has the following format:

```
FOR index IN lower_bound TO upper_bound LOOP
    Sequence of Statements
END LOOP;
```

The upper and lower bounds of the loop may be exchanged and the TO replaced with DOWNTO in order to have the loop count down instead of up.

2. Use your VHDL module to manipulate an input data byte into either cipher text or plain text depending on the type of data with which it is presented. Be sure to provide your LFSR module with the appropriate control signals from the I/O controls. You may find the pushbutton-switch debouncing module presented in Appendix D of this laboratory manual useful for this purpose.

4. In the Lab

1. Enter your design for the data encryption/decryption circuit into the Xilinx ISE development environment.
2. Simulate your design to ensure the proper functionality of the LFSR. If you included the pushbutton-switch debouncing module in your design from the pre-laboratory, you must remove it from your circuit temporarily to allow for simulation of your circuit. If your design does not simulate properly, correct all inconsistencies before continuing to the next section of this laboratory procedure.
3. Program the Spartan-3E Starter Kit board with your design and physically test your circuit using the following data byte sequence and initial key value.
Enter the following key value into your encoder/decoder: 34 (hex)
Decrypt the following sequence of numbers:

67 (hex)
6A (hex)
D1 (hex)

To verify if your design is working properly, the sequence you should get after decryption is:

45 (hex)
43 (hex)
45 (hex)

Which, if you consult the ACSII tables in Appendix E of this laboratory manual, translates to the following character string:

ECE

4. After you have verified that your decryption circuit produces the correct results, decrypt the following sequence of characters and include both the decrypted hexadecimal results and the translated ASCII string in your laboratory report to be turned in next week.

Initial key value: C7 (hex)

Data sequence:

AB	94	94	28	75	15	FC	07	D5	08
92	15	D8	05	28	79	65	6E	B5	AE
38	DE	0E	C5	AF	80	93	2D	D5	FE

5. Finally, encrypt the following sequence of ASCII characters. Be sure to include the hexadecimal results of the encryption and the translated cipher text ASCII string in your laboratory report to be turned in next week.

ASCII string: Goodbye

Initial key value: 72 (hex)

Data sequence:

47 6F 6F 64 62 79 65

5. Post-Lab

Include a solution to the following questions in your laboratory write-up to be turned in next week.

1. While the encryption/decryption circuit implemented in the laboratory did not utilize shift amounts greater than one, the use of this technique is desirable to create some additional security. If the LFSR is built using D flip-flops, compose the logic functions to be presented to each of the flip-flop inputs in order to allow **three** shifts for each rising edge of the clock. Each flip-flop in the 8-bit LFSR needs its own input function, and you must carry the appropriate XNOR logic as shown in Figure 2 through your logic functions.

6. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.
3. Meyer-Baese, Uwe. (2001). Digital Signal Processing with Field Programmable Gate Arrays. Berlin: Springer-Verlag.

Laboratory Experiment 10:

D/A Converters

1. Purpose

This laboratory experiment will introduce some basic digital-to-analog (D/A) conversion circuits. The Spartan-3E Starter Kit board and an oscilloscope will be employed to construct a counter module that will be used in this laboratory and to check digital signal converted to analog signal.

2. Background

Digital processing has become increasingly important for manipulating analog signals needed in applications ranging from communications to motor control. In order to achieve these processing requirements, a method of converting analog signals into a digital format and back is necessary. An analog-to-digital converter is used to sample an analog signal at a given time, and to generate a digital approximation of the signal voltage at the time it was sampled. A series of these digital approximations may then be chained together to get a digital approximation of the original analog signal. Since only a finite number of bits are used to represent the signal voltage level, some precision is lost, but this conversion allows a digital processing element to manipulate the analog signal approximation. After manipulation of the digital signal, it may be necessary to convert the digital representation of the signal back to an analog waveform. To achieve this, a digital-to-analog converter receives a digital bit pattern and converts this bit pattern to an analog voltage level at a single output. This laboratory experiment will introduce an example of digital-to-analog circuit to be implemented with the Spartan-3E Starter Kit board and some additional, externally supplied circuit elements.

2.1 D/A Conversion

Creating a simple D/A converter circuit is an easy task, requiring only a set of resistors. For this reason, and because the A/D converter requires the use of a D/A converter, the D/A converter will be addressed first prior to the next laboratory. The R-2R resistor ladder network is a very simple circuit that achieves all the functionality required of a D/A converter. This circuit works by accepting a series of binary input bits, and weighting them through the resistor network to provide an analog voltage output. A schematic for this circuit is shown in Figure 1 below. For the purposes of this laboratory, the value of R shown in Figure 1 will be 1 K Ω .

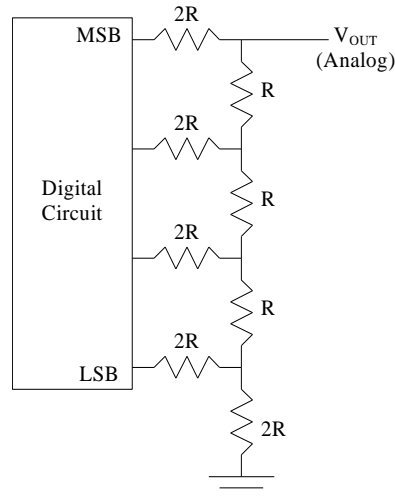


Figure 1 – An R-2R resistor ladder D/A converter.

While Figure 1 shows a four-bit version of the R-2R resistor ladder D/A converter, it may be expanded for larger numbers of bits, using the same design principles used to create the smaller version. The accuracy of this type of D/A converter is highly dependent on the tolerances of the resistors used, and on variations in the voltage levels of the digital inputs to the circuit. For these reasons, while this circuit is simple to design and construct, it is rarely used in the digital signal processing industry. Commercially available D/A converters often employ techniques such as current source networks, or pulse-width modulation and filtering to achieve a greater degree of accuracy, predictability, and uniformity of the analog output voltage.

2.2 Expansion Connectors – FX2 Breadboard

Since we are required to connect the physical circuitry to the Spartan-3E Starter Kit board, an expansion board will be used to connect R-2R resistor ladder D/A converter to the Spartan-3E Starter Kit board. FX2 Breadboard will be used for an expansion board connecting via Hirose 100-pin FX2 Connector (J3) on the Spartan-3E Starter Kit board. There are various pin connections available for users including 40 I/O pins to connect with the Spartan-3E Starter Kit board and the FX2 Breadboard. These I/O connections are shared with other compartments on the Spartan-3E Starter Kit board, so you must be aware of the pin assignment when using an expansion board. The detailed pin assignment is listed on Table 15-1, Page 115 from Xilinx Spartan-3E Starter Kit Board User Guide. Also, you must pay attention to jumper settings on FX2 Breadboard to deliver power to $V_U(5.0V)$ and $V_{CC}(3.3V)$ busses. Read FX2 Breadboard Reference Manual to gain correct jumper settings for FX2 Breadboard.

3. Preliminary Design

In this laboratory, the R-2R resistor ladder D/A converter circuit will be constructed and tested. In addition to the external circuitry required, this circuit will require a four-bit counter to be constructed, using the Spartan-3E FPGA and VHDL. This counter must not only have the basic four-bit counter functionality, but must also have enable and synchronous clear inputs, as well as an additional triggering output. The counter should only advance when the enable input is logic high, and should reset to zero when the clear input is asserted. The triggering output should be logic high when the output of the counter is zero. This output will be used to trigger an oscilloscope in the laboratory in order to view the output voltage from the D/A converter.

Before coming to lab, you should:

1. Compute the analog output voltage, V_{OUT} , for each input combination to the R-2R resistor ladder network shown in Figure 1, assuming a logic high digital input voltage of 3.3-volts and a logic low input voltage of 0.0-volts. Use a value of 1 K Ω for each R shown in the figure. List the input combinations along with the corresponding output voltages in a table and turn this in as part of your pre-laboratory.

HINT: You may wish to use superposition and Thevenin's theorem rather than mesh or nodal analysis.

2. Design a VHDL implementation of the four-bit counter described above. Be sure to include the functionality to enable/disable the counter, and to clear its outputs. In addition, include the triggering output to be presented as high when the output of the counter is zero. Your design should use the selectable frequency clock divider module from Appendix B of this laboratory. Include a printout of the VHDL code with your pre-laboratory work.

4. In the Lab

1. Enter your design for the four-bit counter circuit into the Xilinx ISE development environment.
2. Simulate your design to ensure the proper functionality of the counter. If you included the clock division module in your design from the pre-laboratory, you must remove it from your circuit temporarily to allow for a manageable simulation. If your design does not simulate properly, correct all inconsistencies before continuing to the next section of this laboratory procedure.
3. Construct the R-2R D/A converter circuit on the FX2 Breadboard. Connect the inputs for this circuit to the outputs from your counter module running on the Spartan-3E Starter Kit board, and set the counter to be continuously enabled. You should set your clock divider to produce 1Hz or 10Hz, in order to check the output correctly.

4. Connect the analog output of your D/A converter circuit to an oscilloscope, and use the *trigger* output of your counter module to trigger the oscilloscope. Measure the voltages output by your D/A converter for each input combination. Record these voltages in a table, and create a plot of the input digital value versus the output analog voltage. How do your measured voltages differ from the “ideal” voltages that you calculated in Pre-Laboratory Exercise 1? Be sure to discuss any differences you observe in your laboratory report.

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 11: A/D Converters

1. Purpose

This laboratory experiment will introduce some analog-to-digital (A/D) conversion circuits. The Spartan-3E Starter Kit board will be employed to use the counter module constructed for the circuit in the previous laboratory.

2. Background

Digital processing has become increasingly important for manipulating analog signals needed in applications ranging from communications to motor control. In order to achieve these processing requirements, a method of converting analog signals into a digital format and back is necessary. An analog-to-digital converter is used to sample an analog signal at a given time, and to generate a digital approximation of the signal voltage at the time it was sampled. A series of these digital approximations may then be chained together to get a digital approximation of the original analog signal. Since only a finite number of bits are used to represent the signal voltage level, some precision is lost, but this conversion allows a digital processing element to manipulate the analog signal approximation. This laboratory experiment will introduce an example of this circuit to be implemented with the Spartan-3E Starter Kit board and some additional, externally supplied circuit elements.

The circuits involved with A/D conversion are more complex than the simple R-2R resistor ladder network presented in the previous laboratory. The basic idea for A/D conversion is that of comparing a digital approximation for an analog signal with a sampled value of the signal. A block diagram of this process is shown in the following figure.

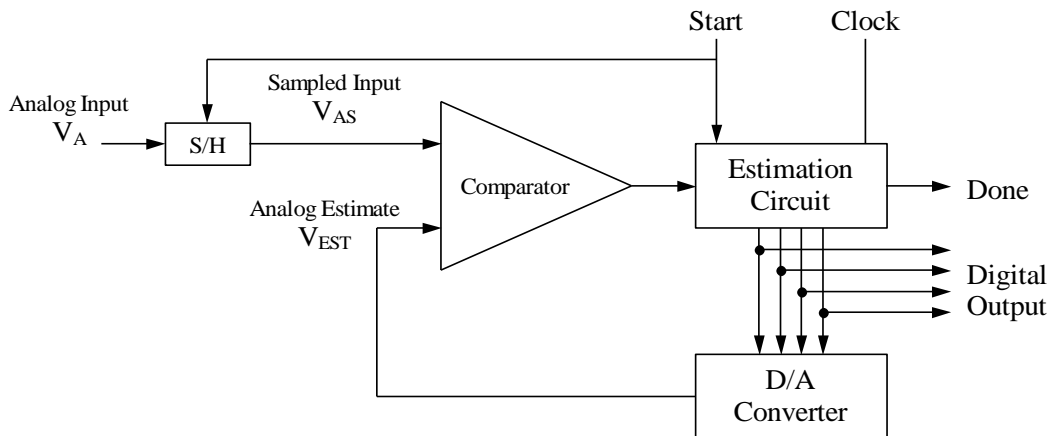


Figure 1 – Block diagram of a D/A converter.

The analog voltage signal to be converted is first fed into a Sample and Hold (S/H) circuit. When the S/H circuit sees a rising edge of the *Start* signal, the analog voltage at its input is sampled, and this value is retained until the next rising edge on the *Start* signal. For the purposes of this experiment, we will not be concerned with the Sample and Hold circuitry; instead, we will simply be providing a constant analog input voltage to the A/D converter. The assertion of the *Start* signal is also used to begin the conversion process itself. The comparator block checks the sampled analog voltage, V_{AS} , against the current analog estimate of the digital output signal, V_{EST} . The estimation circuit adjusts its output with each clock cycle until the estimated voltage matches the sampled voltage as closely as possible. At this point, the estimation circuit asserts the *Done* signal and holds its output at the current value until the next *Start* signal is asserted.

While the previous discussion described the basic A/D conversion functionality, there are still a few details of the estimation circuit that have not been addressed. The simplest estimation circuit is composed of a counter that is initialized to zero when the start signal is asserted, and counts up one step with each clock tick if $(V_{AS} - V_{EST}) > 0$. If this condition is not true, $(V_{AS} - V_{EST}) \leq 0$, the counter is disabled and its outputs will correspond to the digital conversion of the analog signal. This estimation circuit forms the heart of the sequential estimation A/D converter circuit that will be constructed in this laboratory. As long as the analog estimate of the digital output is less than the sampled value of the original analog signal, the estimate is increased incrementally, until the estimate becomes larger than the sampled analog voltage. This behavior produces the following output voltage behavior for a given analog input voltage.

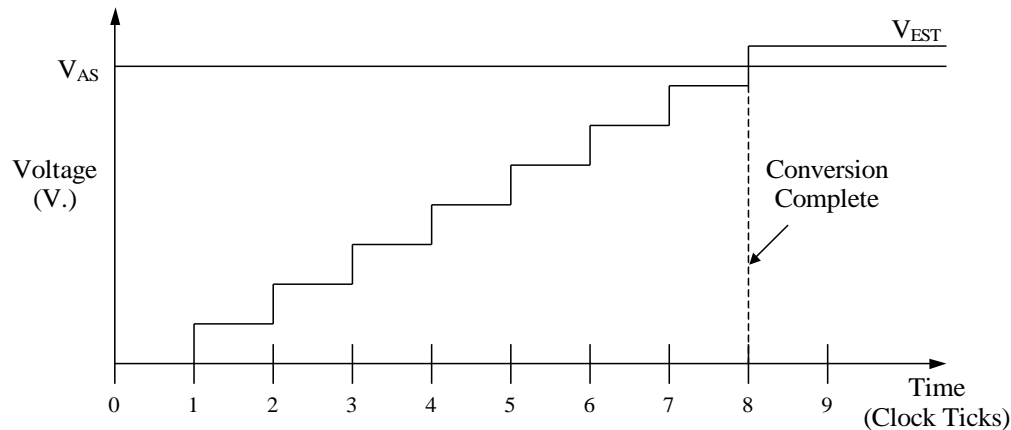


Figure 2 – Output waveform for a sequential estimation A/D conversion process.

3. Preliminary Design

In this laboratory, the sequential estimation A/D converter circuit will be constructed and tested. In addition to the external circuitry required, this circuit will require the four-bit counter constructed in the previous laboratory using the Spartan-3E FPGA and VHDL.

Before coming to lab, you should:

1. Design a counter based sequential estimation A/D converter using the R-2R D/A converter, the counter circuit you designed for the previous laboratory, and the comparator circuit provided in Figure 3 below. The converter circuit should clear its output when the *Start* signal is asserted, and should count up until the comparator indicates that $V_{EST} > V_A$. At this point, the estimation circuit should stop counting, and a *Done* signal should be asserted (you may use a logic low assertion). Be sure to include device pin numbers in your design.

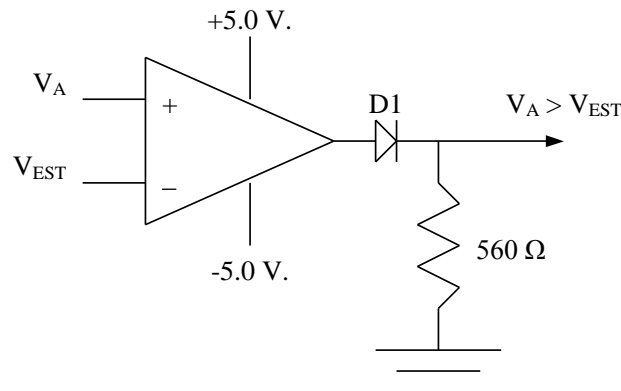


Figure 3 – Schematic for the comparator circuit.

2. Assuming that the output of the estimation circuit for the A/D converter is in the 0.0 to 3.3 voltage range, what is the input voltage range for V_A , assuming the use of the four-bit R-2R resistor ladder network?
3. Explain a method for modifying Figure 1 to allow the conversion of both positive and negative analog input voltages. Do not design any circuits for this bipolar converter; instead, simply outline the procedure. Remember, the D/A converter circuit only produces positive voltages.
4. If the bipolar converter from the previous question were to be constructed, what would be the new range for the analog input voltage V_A ?
5. What will happen to the output voltage from the estimation circuit if the analog input voltage is increased beyond the range calculated in the previous question?

4. In the Lab

1. Construct the sequential estimation A/D converter you designed in Pre-Laboratory Exercise 3 and verify its functionality. For the +5V/-5V in the comparator, use the Power Supply. For V_A , use the function generator, outputting no AC signal, and a DC offset of your desired analog voltage. Make sure all grounds (from the power supply, function generator, oscilloscope, Spartan-3E, and your circuit) are tied together. To prevent damage to the Spartan-3E Starter Kit board, test the output of your comparator circuit with the oscilloscope **before** connecting it to an input pin of the Spartan-3E FPGA. Be sure that the logic high output of the comparator does not exceed 3.3 volts, and that the logic low output does not drop below 0.0 volts. If either of these voltage thresholds is exceeded, adjust the power supply voltages to the operational amplifier to get the output voltages within the correct range.
2. Create a table and record the voltage ranges corresponding to each output bit pattern (0 to 15) for your circuit. How do these input voltage ranges compare to those of an “ideal” A/D converter? When you are sure your converter circuit is functioning properly, demonstrate its operation to your laboratory instructor.
3. Calculate the worst-case conversion time for your A/D converter and discuss this result in your laboratory report.

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Laboratory Experiment 12:

Successive Approximation A/D Converter

1. Purpose

The previous laboratory experiment introduced the concepts of D/A and A/D conversion, and this laboratory will introduce a new A/D conversion technique. A successive approximation A/D converter will be designed and implemented, using the Spartan-3E Starter Kit board. This type of A/D converter produces more uniform conversion times than the sequential estimation circuit used in the previous laboratory.

2. Background

The sequential estimation A/D converter circuit designed in the previous laboratory experiment, while simple to construct, produces conversion times that may vary widely. For a small analog input voltage, the conversion time to a digital value will be relatively short, but for a large analog input voltage, a four-bit A/D converter will have a worst-case conversion time of 16 clock cycles. There are several techniques that may be used to improve upon the performance of the A/D converter, and one of these techniques is the Successive Approximation conversion approach.

2.1 Successive Approximation A/D Conversion

Rather than simply counting up to obtain an input voltage estimate, the successive approximation A/D converter sets individual output bits and tests their effects on the estimated voltage. A given digital conversion process begins with setting the most significant bit of the digital output to one, and observing the comparison of the estimated voltage, V_{EST} , with the actual analog input, V_A . If V_{EST} is greater than V_A , then the most significant bit is reset to zero, and the process repeats with the next most significant bit. If, on the other hand, the estimate is less than the analog input voltage, the first bit is allowed to stay at one, and the next most significant bit is set and tested. This process is repeated until all the bits of the digital output have been appropriately set. Ultimately, the resulting digital output will have an analog conversion that is slightly less than the actual analog input. This may be compared to the sequential estimation A/D converter, where the analog conversion of the digital output was slightly greater than the actual analog voltage input. The overall conversion process is faster than that for the worst-case of the sequential estimation A/D converter. Whereas, an n -bit sequential estimation A/D converter has a worst-case conversion time of $2^n - 1$ clock cycles, an n -bit successive approximation A/D converter has a worst-case conversion time of n clock cycles. The following figure shows an example of a complete conversion process, using a successive approximation A/D converter.

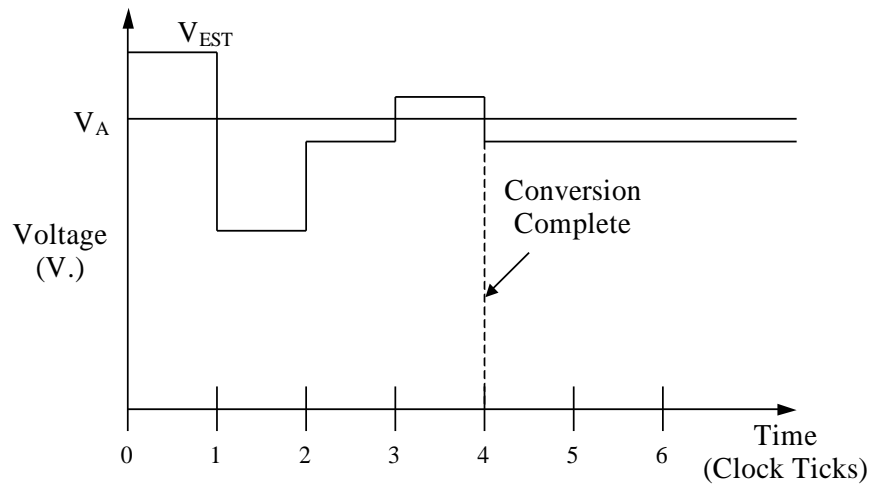


Figure 1 – A complete successive approximation A/D conversion process.

2.2 Constructing a Successive Approximation A/D Converter

All of the conversion functionality of the successive approximation A/D converter may be summed up quite nicely in a flowchart describing an Algorithmic State Machine (ASM). Figure 2 below shows just such a flowchart.

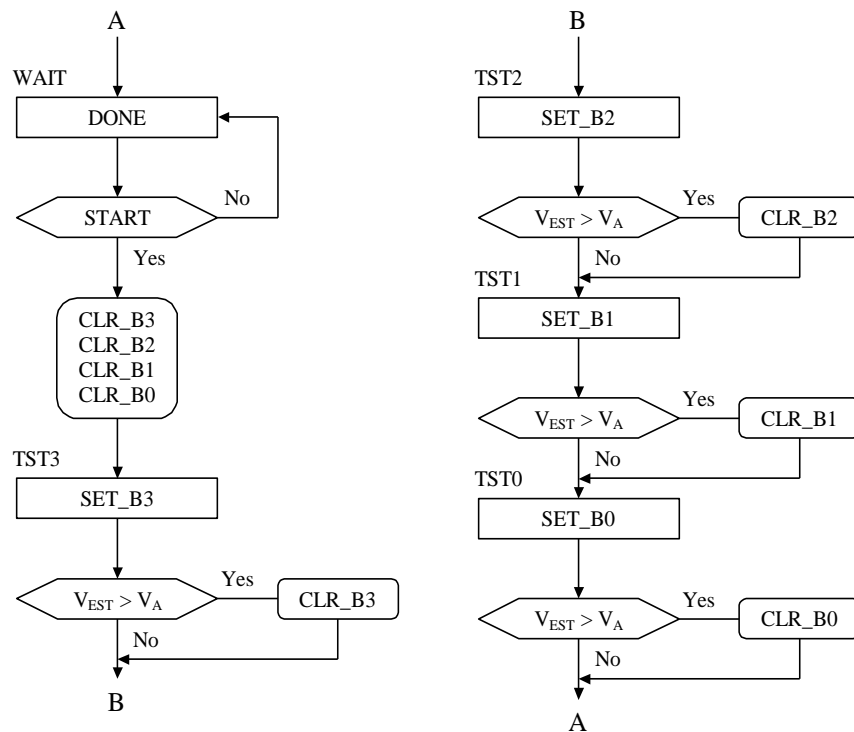


Figure 2 – ASM chart for the successive approximation A/D converter.

In the initial WAIT state, this circuit loops, waiting for a *START* signal telling it to initiate a new conversion process. While the circuit waits, it outputs a *DONE* signal in order to let any interface hardware know that the previous conversion has completed, the output may be read, and a new conversion may be initiated at any time. When the *START* signal is asserted, the bits of the digital output (B3 to B0) are cleared in preparation for the set-and-test operations that will begin shortly. In the TST3 state, the most significant bit of the digital output is set to one, and the resulting voltage estimate is compared to the analog input, using an external comparator circuit. If the estimate is larger than the analog input, the bit that was set must be cleared. If the estimate is still less than the analog input voltage, the bit that was set may be left alone. When this testing process is completed, the device will proceed to the TST2 state. This state performs the same operations as the TST3 state, only it operates on the second most significant bit of the digital output. Each of the following states operates on a successively less significant bit, until all four bits for this four-bit A/D converter have been appropriately set.

3. Preliminary Design

This laboratory experiment will make use of both the R-2R resistor ladder network, and the voltage comparator circuits from the previous laboratory. These circuits will be constructed and tied together in the same manner as for testing the sequential estimation A/D converter circuit. Like the previous laboratory experiment, all the digital functionality of the successive approximation A/D converter will be implemented using the Spartan-II FPGA on the Spartan-3E Starter Kit board. The following figure shows a block diagram of the successive approximation A/D converter.

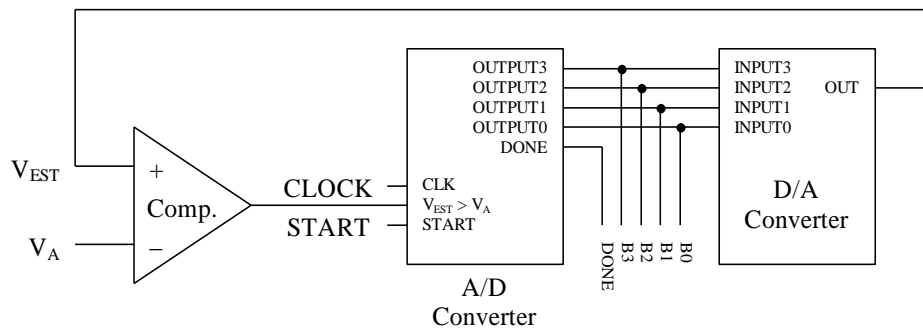


Figure 3 – Block diagram of a successive approximation A/D converter.

Before coming to lab, you should:

1. Design a VHDL implementation of the four-bit successive approximation A/D converter described in the ASM diagram of Figure 2. Be sure to include a RESET signal in your design that, when asserted, will put the device in the WAIT state. In the laboratory, you will need to use one of the clock divider modules supplied in Appendix B of this laboratory manual. In order to test your design, however, the module you chose to employ must be removed; therefore, you may decide to include the module now or in the laboratory.

2. Draw a schematic diagram of your complete successive approximation A/D converter, including the comparator circuit, the FPGA implementation of the converter itself, and the D/A resistor ladder converter. Be sure to include pin numbers for all circuit components.
3. Draw a timing diagram for the analog-to-digital conversion shown in Figure 1, in the Background section of this laboratory. Be sure to show the state of the device and output bit values as the conversion process proceeds.
4. As was discussed in the Background section of this laboratory, the analog conversion of the final digital output from one of these circuits may not match the analog input voltage exactly. Instead, there is usually some error between the output voltage estimate, V_{EST} , and the actual analog input, V_A . Let

$$E = V_A - V_{EST}$$

Write an equation for the maximum error, E , in terms of the total conversion voltage range V_r , and the number of bits, n , in the digital output. Using this equation, calculate the maximum error, in volts, when $V_r = 5V$, and $n = 4$. Repeat this exercise with $V_r = 5V$ and $n = 8$.

4. In the Lab

1. Enter your design for the four-bit successive approximation A/D converter circuit into the Xilinx ISE development environment.
2. Simulate your design to ensure the proper functionality of the converter. If you included the clock division module in your design from the pre-laboratory, you must remove it from your circuit temporarily, to allow for a manageable simulation. If your design does not simulate properly, correct all inconsistencies before continuing to the next section of this laboratory procedure.
3. Construct the complete successive approximation A/D converter circuit you designed in Pre-Laboratory Exercise 2, and verify its functionality. To prevent damage to the Spartan-3E Starter Kit board, test the output of your comparator circuit **before** connecting it to an input pin of the Spartan-3E FPGA. Be sure that the logic high output of the comparator does not exceed 3.3 volts, and that the logic low output does not drop below 0.0 volts. If either of these voltage thresholds is exceeded, adjust the power supply voltages to the operational amplifier to get the output voltages within the correct range.
4. Create a table and record the input voltage ranges corresponding to each output bit pattern (0 to 15) for your circuit. Observe these input voltage ranges and note any variations in regularity in your laboratory report. When you are sure that your converter circuit is functioning properly, demonstrate its operation to your laboratory instructor.

5. In your laboratory report, discuss some of the functional differences between the sequential estimation and successive approximation A/D converters. Be sure to point out some advantages and disadvantages of each design. Is there ever an instance where the sequential estimation A/D converter will be faster than the successive approximation A/D converter?

5. References

For further information on digital logic design, consult:

1. Mano, Morris M. Digital Design. New Jersey: Prentice Hall.
2. Wakerly, John F. (2001). Digital Design Principles & Practices Third Edition. New Jersey: Prentice Hall.

Appendix A: Spartan-3E Starter Kit board Pin Mappings

Board Designator	Spartan-3e Pin Number	
SW0	L13	Board Switches
SW1	L14	
SW2	H18	
SW3	N17	
J1-SW0	B4	J1 PMOD Switches
J1-SW1	A4	
J1-SW2	D5	
J1-SW3	C5	
J2-SW0	A6	J2 PMOD Switches
J2-SW1	B6	
J2-SW2	E7	
J2-SW3	F7	
BTN_NORTH	V4	Push Buttons
BTN_EAST	H13	
BTN_SOUTH	K17	
BTN_WEST	D18	
ROT_A	K18	Rotary Dial
ROT_B	G18	
ROT_CENTER	V16	
LED0	F12	Board LEDs
LED1	E12	
LED2	E11	
LED3	F11	
LED4	C11	
LED5	D11	
LED6	E9	
LED7	F9	
LCD_DB_4	R15	Character LCD
LCD_DB_5	R16	
LCD_DB_6	P17	
LCD_DB_7	M15	
LCD_E	M18	
LCD_RS	L18	
LCD_RW	L17	

Possible Clock Sources	
Spartan-3e Pin Number	Description
C9	Main board 50 MHz crystal oscillator.
B8	Auxiliary clock oscillator socket.
A10	SMA clock connector.

Appendix B: VHDL 50-MHz Clock Divider Modules

1. 1 Hz Clock Generator

This first code module may be used to generate a clock rate of approximately 1 Hz using a 50 MHz crystal oscillator provided on the Spartan-3E Starter Kit board. In order to produce the correct output from this module, the input clock signal must be mapped to pin C9 of the Spartan-3e FPGA. This pin is tied directly to the output of the 50 MHz crystal oscillator, and is divided down by this code module to produce a bounce-free, 1 Hz clock signal.

```
-- One hertz clock divider code.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity one_hz_clock is
    Port ( clk : in std_logic;
          one_hz : out std_logic);
end one_hz_clock;

architecture Behavioral of one_hz_clock is

begin

    process (clk)
        variable count : integer := 0;
    begin
        if clk = '1' and clk'event then
            count := count + 1;

            if count = 50000000 then
                count := 0;
            end if;

            if count >= 0 and count <= 25000000 then
                one_hz <= '1';
            else
                one_hz <= '0';
            end if;
        end if;
    end process;

end Behavioral;
```


2. Selectable Output Frequency Clock Divider

This second code module also achieves division of the 50 MHz signal from the crystal oscillator, but this module allows the output clock rate to be selected from four values based upon two extra input signal values. The following table shows the resulting output clock rate for each input combination possible:

s1	s0	Output Clock Frequency
0	0	1/10 Hz
0	1	1 Hz
1	0	10 Hz
1	1	1 KHz

Table 1 – Mapping of selection inputs to output clock frequencies.

It may be desirable to utilize this clock divider circuit in the event that a sequential circuit design allows for two or more unused board toggle switches. The behavior of the design may then be observed at a variety of different rates, including one that most likely may only be viewed using a logic analyzer.

```
-- Selectable output frequency clock divider code.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity selectable_clock is
    Port ( clk : in std_logic;
          s0 : in std_logic;
          s1 : in std_logic;
          out_clk : out std_logic);
end selectable_clock;

-- If s1 and s0 are both low, the output clock rate is 1/10 Hz.
-- If s1 is low and s0 is high, the output clock rate is 1 Hz.
-- If s1 is high and s0 is low, the output clock rate is 10 Hz.
-- If s1 and s0 are both high, the output clock rate is 1 KHz.

architecture Behavioral of selectable_clock is
begin

    process (clk, s0, s1)
        -- Start a process.
        variable count : integer := 0;
        -- Variable declaration.
        begin
            if clk = '1' and clk'event then
                -- Rising edge detection.
                count := count + 1;

                -- Code to create the 1/10 Hz clock.
                if s0 = '0' and s1 = '0' then
                    if count >= 500000000 then
                        -- Taken off a 50MHz clock.
                        count := 0;
                        -- Reset count for next cycle.
                    end if;
                end if;
            end if;
        end process;
    end architecture;
```

```

        if count >= 0 and count <= 250000000 then
            out_clk <= '1';          -- High portion of 1/10 HZ clock.
        else
            out_clk <= '0';          -- Low portion of 1/10 HZ clock.
        end if;
    end if;

-- Code to create the 1 Hz clock.
if s0 = '1' and s1 = '0' then
    if count >= 50000000 then        -- Taken off a 50MHz clock.
        count := 0;                 -- Reset count for next cycle.
    end if;

    if count >= 0 and count <= 25000000 then
        out_clk <= '1';          -- High portion of 1 HZ clock.
    else
        out_clk <= '0';          -- Low portion of 1 HZ clock.
    end if;
end if;

-- Code to create the 10 Hz clock.
if s0 = '0' and s1 = '1' then
    if count >= 5000000 then        -- Taken off a 50MHz clock.
        count := 0;                 -- Reset count for next cycle.
    end if;

    if count >= 0 and count <= 2500000 then
        out_clk <= '1';          -- High portion of 10 HZ clock.
    else
        out_clk <= '0';          -- Low portion of 10 HZ clock.
    end if;
end if;

-- Code to create the 1 KHz clock.
if s0 = '1' and s1 = '1' then
    if count >= 50000 then          -- Taken off a 50MHz clock.
        count := 0;                 -- Reset count for next cycle.
    end if;

    if count >= 0 and count <= 25000 then
        out_clk <= '1';          -- High portion of 1 KHz clock.
    else
        out_clk <= '0';          -- Low portion of 1 KHz clock.
    end if;
end if;

end if;
end process;

end Behavioral;

```

Appendix C:

Adding Additional Logic to the Spartan-3E Starter Kit board

1. Motivation

As the designs you wish to download to the Spartan-3E Starter Kit board become more complex, you may find that the main board does not contain enough logic switches to manipulate all the inputs of your circuit. In this event, it may be desirable to add several logic switches to the test board, thereby increasing the input capabilities of the board. This task is relatively simple with the correct supplies and a little time.

2. Supply List

- FX2 expansion board
- DIP-Switch Package
- 4.7 K Ω Resistors

3. Connection Procedure

First, it should be noted that while it is possible to add DIP switches to the Spartan-3E Starter Kit board utilizing a separate, user provided breadboard, the use of only the FX2-BB board is **highly recommended**. This breadboard will minimize the possibility of incorrect connections that may permanently damage the test board.

The Xilinx XC3S500E Spartan-3E FPGA that the Spartan-3E Starter Kit board is designed around uses a 3.3 V LVTTL/LVCMOS voltage level for its inputs and outputs. This means that while a logic low input is still represented by 0.0 V, a logic high input is represented by only 3.3 V. This is a **critical** note when building any interface hardware for the Spartan-3E Starter Kit board, because if standard TTL voltage levels are interfaced to the Spartan-II FPGA, the chip may be **permanently damaged**.

In order to add a logic switch to the Spartan-3E Starter Kit board, two resistors and some jumper wires are needed. The following schematic shows how to create one such logic switch.

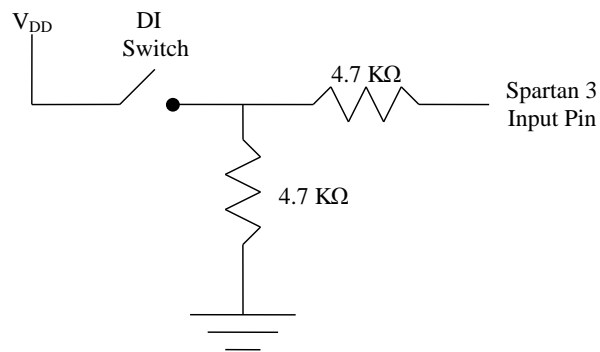


Figure 1 – Schematic for an additional logic switch.

One resistor in the schematic presented in Figure 1 is used to pull down the logic level of the Spartan-3E input pin when the DIP-switch is open, producing a logic low. The other resistor is responsible for providing input protection to the Spartan-3E input pin. When the switch is closed, the Spartan-3E input pin is presented with a logic high, and the resistor is then used to limit the current flow in the system, preventing a short of V_{DD} to ground.

The schematic presented in Figure 1 should be replicated for each additional logic switch to be added to the Spartan-3E Starter Kit board. In order to utilize the logic switches that are added to the test board, they must be connected to input pins of the Spartan-3E FPGA that are not already connected to other I/O devices. If an added DIP-switch were to be connected to a Spartan-3E pin already in use, this could result in a short that could damage the Spartan-3E Starter Kit board. It should be mentioned that IO1 to IO20 are shared with LEDs, J1, J2 and J4 of Spartan-3E Starter Kit board.

4. List of Available Spartan-3E I/O Pins

The following table lists the Spartan-3E I/O pins that may be assigned to additional logic switches added to the Spartan-3E Starter Kit board.

Starter KIT Expansion Connection (J3)	FX2 Expansion Connection (J12)	Spartan-3E Pin Number
A6	IO1	B4
A7	IO2	A4
A8	IO3	D5
A9	IO4	C5
A10	IO5	A6
A11	IO6	B6
A12	IO7	E7
A13	IO8	F7
A14	IO9	D7
A15	IO10	C7
A16	IO11	F8
A17	IO12	E8
A18	IO13	F9
A19	IO14	E9
A20	IO15	D11
A21	IO16	C11
A22	IO17	F11
A23	IO18	E11
A24	IO19	E12
A25	IO20	F12
A26	IO21	A13
A27	IO22	B13
A28	IO23	A14
A29	IO24	B14
A30	IO25	C14
A31	IO26	D14
A32	IO27	A16
A33	IO28	B16
A34	IO29	E13
A35	IO30	C4
A36	IO31	B11

A37	IO32	A11
A38	IO33	A8
A39	IO34	G9
A40	IO35	D12
A41	IO36	C12
A42	IO37	A15
A43	IO38	B15
A44	IO39	C3
A45	IO40	C15

Table 1 – List of Spartan-3E I/O pins and their connector mappings.

5. How to connect the FX2-BB Expansion Board

The board can be seen in Figure 2 below.

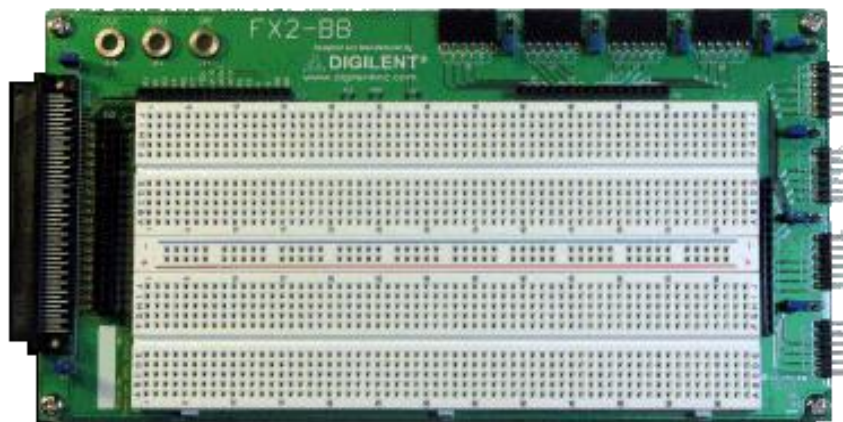


Figure 2 – Picture of FX2-BB Expansion Board

The 100-pin connector at the left is connected to the Spartan-3E. Make sure that the jumpers above and below the connector are in place, ensuring that power is supplied to the V_U (5.0V) and V_{CC} (3.3V) buses on the board. The 40 I/O pins can be found to the right of the connector. GND, V_U , and V_{CC} can be found just to the right of I/O pin 1. Make sure that the ground in your circuit is tied to the ground of the Spartan, as well as to the grounds of any additional instruments you are using.

Appendix D:

Pushbutton Switch De-bouncer Module

This code module may be used to de-bounce the pushbutton switches on the Spartan-3E Starter Kit board. The input to this module may be attached to any of the on-board pushbutton switches, and the clock must be tied to pin C9 of the Spartan-3E FPGA. In this configuration, when the pushbutton switch is high for 1,000,000 continuous clock cycles (approximately 20 ms), the output of the module will go to high for the remainder of the duration for which the switch is activated.

```
-- Pushbutton debouncer code module.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity debouncer is
    Port ( input : in std_logic;
          clk : in std_logic;
          output : out std_logic);
end debouncer;

architecture Behavioral of debouncer is

begin

    process (clk, input)
        variable count : integer := 0;
    begin
        -- Start a process.
        -- Variable declaration.
        if clk = '1' and clk'event then
            -- Rising edge detection.
            if input = '1' then
                -- Input is high at clock.
                count := count + 1;
                -- Increment count.
            else
                -- Input is low at clock.
                count := 0;
                -- Reset count.
            end if;

            if count > 1000000 then
                -- Input high long enough to
                -- output.
                output <= '1';
                -- Output high.
            else
                -- Input not high long enough.
                output <= '0';
                -- Output low.
            end if;
        end if;
    end process;

end Behavioral;
```

Appendix E: ASCII-I and ASCII-II Tables

The following tables contain the ASCII-I and ASCII-II characters along with their decimal and hexadecimal equivalents.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of Heading	33	21	!	65	41	A	97	61	a
2	02	Start of Text	34	22	“	66	42	B	98	62	b
3	03	End of Text	35	23	#	67	43	C	99	63	c
4	04	End of Transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible Bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal Tab	41	29)	73	49	I	105	69	i
10	0A	Line Feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical Tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form Feed	44	2C	‘	76	4C	L	108	6C	l
13	0D	Carriage Return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift Out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift In	47	2F	/	79	4F	O	111	6F	o
16	10	Data Link Escape	48	30	0	80	50	P	112	70	p
17	11	Device Control 1	49	31	1	81	51	Q	113	71	q
18	12	Device Control 2	50	32	2	82	52	R	114	72	r
19	13	Device Control 3	51	33	3	83	53	S	115	73	s
20	14	Device Control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. Acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous Idle	54	36	6	86	56	V	118	76	v
23	17	End Trans. Block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of Medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[123	7B	{
28	1C	File Separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group Separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record Separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit Separator	63	3F	?	95	5F	_	127	7F	□

Table 1 – ASCII-I Values.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	Ł	224	E0	α
129	81	ü	161	A1	í	193	C1	⊥	225	E1	β
130	82	é	162	A2	ó	194	C2	⌈	226	E2	Γ
131	83	â	163	A3	ú	195	C3	⌋	227	E3	π
132	84	ä	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	⊢	229	E5	σ
134	86	â	166	A6	ª	198	C6	⌌	230	E6	μ
135	87	ç	167	A7	º	199	C7	⌍	231	E7	τ
136	88	ê	168	A8	¿	200	C8	⌎	232	E8	Φ
137	89	ë	169	A9	¬	201	C9	⌏	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	⌐	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	⌑	235	EB	δ
140	8C	î	172	AC	¼	204	CC	⌒	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	⌓	237	ED	∅
142	8E	Ä	174	AE	«	206	CE	⌔	238	EE	ε
143	8F	Å	175	AF	»	207	CF	⌕	239	EF	∩
144	90	É	176	B0	⋮	208	D0	⌖	240	F0	≡
145	91	æ	177	B1	⋮	209	D1	⌗	241	F1	±
146	92	Æ	178	B2	⋮	210	D2	⌘	242	F2	≥
147	93	ô	179	B3	⋮	211	D3	⌙	243	F3	≤
148	94	ö	180	B4	⋮	212	D4	⌚	244	F4	⌈
149	95	ò	181	B5	⋮	213	D5	⌛	245	F5	⌋
150	96	û	182	B6	⋮	214	D6	⌜	246	F6	÷
151	97	ù	183	B7	⋮	215	D7	⌝	247	F7	~
152	98	ÿ	184	B8	⋮	216	D8	⌞	248	F8	°
153	99	Ö	185	B9	⋮	217	D9	⌟	249	F9	·
154	9A	Ü	186	BA	⋮	218	DA	⌠	250	FA	·
155	9B	ç	187	BB	⋮	219	DB	⌡	251	FB	√
156	9C	£	288	BC	⋮	220	DC	⌢	252	FC	ⁿ
157	9D	¥	189	BD	⋮	221	DD	⌣	253	FD	†
158	9E	₣	190	BE	⋮	222	DE	⌤	254	FE	■
159	9F	ƒ	191	BF	⋮	223	DF	⌥	255	FF	□

Table 2 – ASCII-II Values.