

Brock J. LaMeres

Lab Exercises

(DE0-CV Version)

Introduction to Logic Circuits & Logic Design with VHDL

Second Edition

 Springer

INTRODUCTION TO LOGIC CIRCUITS & LOGIC DESIGN

WITH VHDL

2ND EDITION

Brock J. LaMeres, Ph.D.

LAB EXERCISES

(DE0-CV Version)

Last updated June 29, 2019

Table of Contents

TABLE OF CONTENTS	III
INTRODUCTION	1
PARTS LIST	3
CHAPTER 1: ANALOG VS. DIGITAL	7
LAB 1.1: INTRODUCTION TO LAB EQUIPMENT & BLINKING AN LED WITH AN AWG.....	7
1.1.1 <i>Objective</i>	7
1.1.2 <i>Learning Outcomes</i>	7
1.1.3 <i>Parts Needed</i>	7
1.1.4 <i>Deliverables</i>	7
1.1.5 <i>Lab Work & Demonstration</i>	7
1.1.5.1 Using the AWG to make an LED Blink	7
1.1.5.2 Measure the Logic Signals in the LED Circuit	13
CHAPTER 2: NUMBER SYSTEMS	17
LAB 2.1: 2-BIT COUNTER FROM THE AWG AND INTRODUCTION TO LOGIC ANALYSIS	17
2.1.1 <i>Objective</i>	17
2.1.2 <i>Learning Outcomes</i>	17
2.1.3 <i>Parts Needed</i>	17
2.1.4 <i>Deliverables</i>	17
2.1.5 <i>Lab Work & Demonstration</i>	17
2.1.5.1 Using an AWG to Create a 2-bit Binary Counter Pattern.....	17
2.1.5.2 Logic Analyzer Measurement of the Counter Pattern.....	20
CHAPTER 3: DIGITAL CIRCUITRY & INTERFACING	25
LAB 3.1: DIGITAL CIRCUIT OPERATION	25
3.1.1 <i>Objective</i>	25
3.1.2 <i>Learning Outcomes</i>	25
3.1.3 <i>Parts Needed</i>	25
3.1.4 <i>Deliverables</i>	25
3.1.5 <i>Lab Work & Demonstration</i>	26
3.1.5.1 Determine the Operating Specifications of a 74HC Logic Gate from its Data Sheet	26
3.1.5.2 DC Operation of 74HC Logic Gates	26
3.1.5.3 AC Operation of 74HC Logic Gates	31
3.1.5.4 Observing the Impact of an Input Signal that Doesn't Meet the Minimum Specifications.	31
3.1.5.5 Measuring the AC Characteristics of the Inverter	32
CHAPTER 4: COMBINATIONAL LOGIC DESIGN	35
LAB 4.1: 3-INPUT PRIME NUMBER DETECTOR USING CANONICAL FORMS.....	35
4.1.1 <i>Objective</i>	35
4.1.2 <i>Learning Outcomes</i>	35
4.1.3 <i>Parts Needed</i>	35
4.1.4 <i>Deliverables</i>	35
4.1.5 <i>Lab Work & Demonstration</i>	36
4.1.5.1 Implement an LED Driver Circuit.....	36
4.1.5.2 Design a 3-Input Prime Number Detector using a Canonical SOP Form	37

4.1.5.3 Implement a 3-Input Prime Number Detector using a Canonical SOP Form.....	39
4.1.5.4 Design a 3-Input Prime Number Detector using a Canonical POS Form	41
4.1.5.5 Implement a 3-Input Prime Number Detector using a Canonical POS Form.....	42
LAB 4.2: 3-INPUT PRIME NUMBER DETECTOR USING MINIMIZED FORMS	43
4.2.1 Objective.....	43
4.2.2 Learning Outcomes	43
4.2.3 Parts Needed.....	43
4.2.4 Deliverables.....	43
4.2.5 Lab Work & Demonstration	43
4.2.5.1 Design a 3-Input Prime Number Detector using a Minimized SOP Form	43
4.2.5.2 Implement a 3-Input Prime Number Detector using a Minimized SOP Form	45
4.2.5.3 Design a 3-Input Prime Number Detector using a Minimized POS Form	45
4.2.5.4 Implement a 3-Input Prime Number Detector using a Minimized POS Form	46
4.2.5.5 A Buzzer Driving Circuit	46
LAB 4.3: 7-SEGMENT DISPLAY DECODER (DISCRETE)	49
4.3.1 Objective.....	49
4.3.2 Learning Outcomes	49
4.3.3 Parts Needed.....	49
4.3.4 Deliverables.....	49
4.3.5 Lab Work & Demonstration	49
4.3.5.1 Design the Logic for the 7-Segment Decoder	49
4.3.5.2 Use DeMorgan's Theorem to Manipulate at Least One of the Logic Expressions	53
4.3.5.3 Implement your 7-Segment Decoder System.....	53
CHAPTER 5: VHDL (PART 1)	55
LAB 5.1: 4-INPUT PRIME NUMBER DETECTOR IN VHDL	55
5.1.1 Objective.....	55
5.1.2 Learning Outcomes	55
5.1.3 Parts Needed.....	55
5.1.4 Deliverables.....	55
5.1.5 Lab Work & Demonstration	55
5.1.5.1 Implement a VHDL Design that will Drive the Switches to the LEDs on the DE0-CV Board	56
5.1.5.2 Implement a VHDL Design for a 4-Input Prime Number Detector	66
5.1.5.3 Save a Copy of your top.vhd for your Records.....	67
CHAPTER 6: MSI LOGIC.....	69
LAB 6.1: 4-INPUT, 7-SEGMENT DISPLAY DECODER (IN VHDL)	69
6.1.1 Objective.....	69
6.1.2 Learning Outcomes	69
6.1.3 Parts Needed.....	69
6.1.4 Deliverables.....	69
6.1.5 Lab Work & Demonstration	69
6.1.1.1 Implement a VHDL Design for a 7-Segment Decoder + 4-Input Prime Number Detector	69
6.1.1.2 Save a Copy of your top.vhd for your Records.....	75
CHAPTER 7: SEQUENTIAL LOGIC DESIGN.....	77
LAB 7.1: 4-BIT RIPPLE COUNTER & SWITCH DEBOUNCING	77
7.1.1 Objective.....	77
7.1.2 Learning Outcomes	77
7.1.3 Parts Needed.....	77

7.1.4	<i>Deliverables</i>	77
7.1.5	<i>Lab Work & Demonstration</i>	77
7.1.5.1	Implement a 4-Bit Ripple Counter using D-Flip-Flops	77
7.1.5.2	Take a Logic Analyzer Measurement of the 4-Bit Ripple Counter	79
7.1.5.3	Observing Issues with Mechanical Switches	79
7.1.5.4	Debouncing Mechanical Switches.....	81
LAB 7.2:	3-BIT BINARY UP/DOWN COUNTER	85
7.2.1	<i>Objective</i>	85
7.2.2	<i>Learning Outcomes</i>	85
7.2.3	<i>Parts Needed</i>	85
7.2.4	<i>Deliverables</i>	85
7.2.5	<i>Lab Work & Demonstration</i>	85
7.2.5.1	Design the 3-Bit Binary Up/Down Counter	85
7.2.5.2	Implement the 3-Bit Binary Up/Down Counter.....	89
LAB 7.3:	4-BIT BINARY UP/DOWN COUNTER FINITE STATE MACHINE (IN VHDL).....	91
7.3.1	<i>Objective</i>	91
7.3.2	<i>Learning Outcomes</i>	91
7.3.3	<i>Parts Needed</i>	91
7.3.4	<i>Deliverables</i>	91
7.3.5	<i>Lab Work & Demonstration</i>	91
7.3.5.1	Implement the FSM for the 4-Bit Binary Up/Down Counter	91
7.3.5.2	Save a Copy of your top.vhd for your Records.....	99
CHAPTER 8:	VHDL (PART 2)	101
LAB 8.1:	7-SEGMENT DISPLAY DECODER USING A PROCESS	101
8.1.1	<i>Objective</i>	101
8.1.2	<i>Learning Outcomes</i>	101
8.1.3	<i>Parts Needed</i>	101
8.1.4	<i>Deliverables</i>	101
8.1.5	<i>Lab Work & Demonstration</i>	101
8.1.5.1	Implement the 7-Segment Decoder in VHDL on the DE0-CV Board.....	101
8.1.5.2	Save a Copy of your top.vhd for your Records.....	113
LAB 8.2:	DESIGN RE-USE AND BINARY CHARACTERS ON THE 7-SEGMENT DISPLAYS	115
8.2.1	<i>Objective</i>	115
8.2.2	<i>Learning Outcomes</i>	115
8.2.3	<i>Parts Needed</i>	115
8.2.4	<i>Deliverables</i>	115
8.2.5	<i>Lab Work & Demonstration</i>	115
8.2.5.1	Creating a 7-Segment Decoder Subsystem to Drive all the Character Displays	115
8.2.5.2	Display Binary Characters on the 7-Segment Displays	120
8.2.5.3	Save a Copy of your top.vhd for your Records.....	121
CHAPTER 9:	BEHAVIORAL MODELING OF SEQUENTIAL LOGIC	123
LAB 9.1:	RIPPLE COUNTER AND THE CHARACTER DISPLAYS.....	123
9.1.1	<i>Objective</i>	123
9.1.2	<i>Learning Outcomes</i>	123
9.1.3	<i>Parts Needed</i>	123
9.1.4	<i>Deliverables</i>	123
9.1.5	<i>Lab Work & Demonstration</i>	123

9.1.5.1	Implement the Ripple Counter System in VHDL on the DE0-CV Board	125
9.1.5.2	Take a Logic Analyzer Measurement of your Counter	127
9.1.5.3	Save a Copy of your top.vhd for your Records.....	130
LAB 9.2:	A “WALKING 1” FINITE STATE MACHINE	131
9.2.1	<i>Objective</i>	131
9.2.2	<i>Learning Outcomes</i>	131
9.2.3	<i>Parts Needed</i>	131
9.2.4	<i>Deliverables</i>	131
9.2.5	<i>Lab Work & Demonstration</i>	131
9.2.5.1	Implement the Walking 1 FSM	133
9.2.5.2	Take a Logic Analyzer Measurement of your Walking 1 Pattern	134
9.2.5.3	Save a Copy of your top.vhd for your Records.....	135
LAB 9.3:	COUNTERS USING A SINGLE PROCESS AND A 2 ^N CLOCK DIVIDER	137
9.3.1	<i>Objective</i>	137
9.3.2	<i>Learning Outcomes</i>	137
9.3.3	<i>Parts Needed</i>	137
9.3.4	<i>Deliverables</i>	137
9.3.5	<i>Lab Work & Demonstration</i>	137
9.3.5.1	Design the 24-Bit Counter and Selectable 2 ⁿ Clock Divider	139
9.3.5.2	Take a Logic Analyzer Measurement of your Counter	141
9.3.5.3	Save a Copy of your top.vhd for your Records.....	142
LAB 9.4:	PRECISION CLOCK DIVIDER AND A BCD COUNTER	143
9.4.1	<i>Objective</i>	143
9.4.2	<i>Learning Outcomes</i>	143
9.4.3	<i>Parts Needed</i>	143
9.4.4	<i>Deliverables</i>	143
9.4.5	<i>Lab Work & Demonstration</i>	143
9.4.5.1	Implement the Precision Clock Divider	145
9.4.5.2	Implement the 6-Digit BCD Counter	148
9.4.5.3	Save a Copy of your top.vhd for your Records.....	149
CHAPTER 10:	MEMORY	151
LAB 10.1:	ROM MEMORY.....	151
10.1.1	<i>Objective</i>	151
10.1.2	<i>Learning Outcomes</i>	151
10.1.3	<i>Parts Needed</i>	151
10.1.4	<i>Deliverables</i>	151
10.1.5	<i>Lab Work & Demonstration</i>	151
10.1.5.1	Implement the ROM System	153
10.1.5.2	Take a Logic Analyzer Measurement of your ROM System	155
10.1.5.3	Save a Copy of your top.vhd for your Records.....	155
LAB 10.2:	READ/WRITE MEMORY	157
10.2.1	<i>Objective</i>	157
10.2.2	<i>Learning Outcomes</i>	157
10.2.3	<i>Parts Needed</i>	157
10.2.4	<i>Deliverables</i>	157
10.2.5	<i>Lab Work & Demonstration</i>	157
10.2.5.1	Implement the R/W Memory System.....	159
10.2.5.2	Take a Logic Analyzer Measurement of your R/W System During a “Store”	162
10.2.5.3	Take a Logic Analyzer Measurement of your R/W System During a “Read”	163

10.2.5.4	Save a Copy of your top.vhd for your Records.....	164
CHAPTER 11: PROGRAMMABLE LOGIC		165
LAB 11.1:	DETAILS OF AN FPGA	165
11.1.1	Objective.....	165
11.1.2	Learning Outcomes	165
11.1.3	Parts Needed.....	165
11.1.4	Deliverables.....	165
11.1.5	Lab Work & Demonstration	165
11.1.5.1	View the Device Utilization Report	165
11.1.5.2	View the Maximum Frequency Report	167
11.1.5.3	Run the RTL Viewer	168
11.1.5.4	Run the State Machine Viewer.....	170
11.1.5.5	Run the Chip Planner Tool	171
CHAPTER 12: ARITHMETIC CIRCUITS		175
LAB 12.1:	UNSIGNED ADDERS	175
12.1.1	Objective.....	175
12.1.2	Learning Outcomes	175
12.1.3	Parts Needed.....	175
12.1.4	Deliverables.....	175
12.1.5	Lab Work & Demonstration	175
12.1.5.1	Implement the Adder System	177
12.1.5.2	Save a Copy of your top.vhd for your Records.....	178
LAB 12.2:	SIGNED ADDERS	179
12.2.1	Objective.....	179
12.2.2	Learning Outcomes	179
12.2.3	Parts Needed.....	179
12.2.4	Deliverables.....	179
12.2.5	Lab Work & Demonstration	179
12.2.5.1	Implement the Adder System	181
12.2.5.2	Save a Copy of your top.vhd for your Records.....	182
CHAPTER 13: COMPUTER SYSTEM DESIGN		183
LAB 13.1:	8-BIT COMPUTER IMPLEMENTATION.....	183
13.1.1	Objective.....	183
13.1.2	Learning Outcomes	183
13.1.3	Parts Needed.....	183
13.1.4	Deliverables.....	183
13.1.5	Lab Work & Demonstration	183
13.1.5.1	VHDL Shell.....	183
13.1.5.2	Functional Simulation of the Four Basic Instructions.....	187
13.1.5.3	Implementation of Basic Instructions on the FPGA	197
13.1.5.4	Implement Additional Instructions for the Computer System.....	198

Introduction

This workbook contains a series of hands-on lab exercises with digital logic circuits that are designed to reinforce the material from the textbook. These exercises require a parts kit that contains the components listed in the next section. Chapters 1-4 use discrete logic gates and basic input/output devices to provide experience with simple logic circuit synthesis, operation, and interfacing. Chapter 5 introduces a Field Programmable Gate Array (FPGA) board that allows a digital design to be modeled in VHDL and implemented on an FPGA. This provides exposure to the modern digital design flow. Chapters 6 and 7 use a combination of discrete parts and the FPGA to provide further interfacing experience in addition to introducing sequential logic in discrete form. Chapters 8-13 use the FPGA board exclusively to implement larger designs to fully explore the concepts of digital logic.

Each lab begins with a listing of the objective, learning outcomes, supplies needed, and deliverables. Then in the *Lab Work & Demonstration* section, a step-by-step guide is provided to complete the exercise.

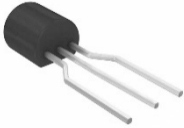





Parts List

The following parts are required to complete all of the lab exercises in this book. The breadboard and the discrete parts are only used on lab exercises for chapters 1-7. The FPGA board starts to be used in Chapter 5. Chapters 8-13 only use the FPGA board and the Analog Discovery 2.

Qty	Image	Description	Manufacturer (Mfn Part #)	Distributor (Distr. Part #)	Data Sheet
1		Solderless Breadboard	Digilent (340-002)	Digi-Key (1286-1062-ND)	
1		Wiring Kit for Solderless Breadboard	Global Specialties (WK-2)	Digi-Key (BKWK-2-ND)	
2		AND Gates, 2-Input, 4x per part	Texas Instruments (SN74HC08N)	Digi-Key (296-1570-5-ND)	Data Sheet
2		AND Gates, 3-Input, 3x per part	Texas Instruments (SN74HC11N)	Digi-Key (296-8217-5-ND)	Data Sheet
2		AND Gates, 4-Input, 2x per part	Texas Instruments (SN74HC21N)	Digi-Key (296-8266-5-ND)	Data Sheet
2		OR Gates, 2-Input, 4x per part	Texas Instruments (SN74HC32N)	Digi-Key (296-1589-5-ND)	Data Sheet
2		OR Gates, 3-Input, 3x per part	Texas Instruments (CD74HC4075E)	Digi-Key (296-33088-5-ND)	Data Sheet

2		Inverters, 6x per part	Texas Instruments (SN74HC04N)	Digi-Key (296-1566-5-ND)	Data Sheet
2		NAND Gates, 2-Input, 4x per part	Texas Instruments (SN74HC00N)	Digi-Key (296-1563-5-ND)	Data Sheet
2		NAND Gates, 3-Input, 3x per part	Texas Instruments (SN74HC10N)	Digi-Key (296-8214-5-ND)	Data Sheet
2		NAND Gates, 4-Input, 2x per part	Texas Instruments (SN74HC20N)	Digi-Key (296-12892-5-ND)	Data Sheet
2		NOR Gates, 2-Input, 4x per part	Texas Instruments (SN74HC02)	Digi-Key (296-1564-5-ND)	Data Sheet
2		NOR Gates, 3-Input, 3x per part	Texas Instruments (SN74HC27)	Digi-Key (296-12896-5-ND)	Data Sheet
2		NOR Gates, 4-Input, 2x per part	Texas Instruments (CD74HC4002)	Digi-Key (296-25987-5-ND)	Data Sheet
2		D-flip-flops, 2x per part	Texas Instruments (SN74HC74N)	Digi-Key (296-1602-5-ND)	Data Sheet
10		LED, Red, Discrete	Kingbright (WP710A10LSRD)	Digi-Key (754-1590-ND)	Data Sheet

1		LED, 7-Segment Display	Lumex Opto/Components Inc. (LDS-C416RI)	Digi-Key (67-1446-ND)	Data Sheet
1		Buzzer, Magnetic, DC, Single Tone	CUI Inc. (CEM-1205C)	Digi-Key (102-1124-ND)	Data Sheet
1		Switch, slider, SPST, 8-position	CTS Electrocomponents (208-8)	Digi-Key (CT2088-ND)	Data Sheet
1		Switch, push-button, SPDT	C&K (KS12R22CQD)	Digi-Key (CKN1595-ND)	Data Sheet
1		Resistor Network, 8x, DIP, 330 Ohm, Isolated	Bourns Inc (4116R-1-331LF)	Digi-Key (4116R-1-331LF-ND)	Data Sheet
1		Resistor Network, 9x, SIP, 10k Ohm, Bussed	Bourns Inc (4610X-101-103LF)	Digi-Key (4610X-1-103LF-ND)	Data Sheet
2		Resistor, Axial, 1k Ohm, 1/4 W, 5%	Yageo (CFR-25JB-52-1K)	Digi-Key (1.0KQBK-ND)	Data Sheet
12		Resistor, Axial, 150 Ohm, 1/4 W, 5%	Yageo (CFR-25JB-52-150R)	Digi-Key (150QBK-ND)	Data Sheet
2		Resistor, Axial, 10k Ohm, 1/4 W, 5%	Yageo (CFR-25JB-52-10K)	Digi-Key (10KQBK-ND)	Data Sheet

1		NPN Transistor, 2N3904, 200 mA	ON Semiconductor (2N3904TFR)	Digi-Key (2N3904D26ZCT-ND)	Data Sheet
1		Diode, 1N4002, 1A	ON Semiconductor (1N4002)	Digi-Key (1N4002FSCT-ND)	Data Sheet
1		Jumper Wires, Female-to- Female, 10-Pack	MikroElektronika (MIKROE-511)	Digi-Key (1471-1230-ND)	
1		Pin Header, 0.1", Single-Strip, 10- pos	3M (929834-02-10- RK)	Digi-Key (929834E-02-10-ND)	Data Sheet
1		DE0-CV, Cyclone V, FPGA Board	Terasic (P0192)	Digi-Key (P0192-ND) or Terasic (P0192)	User Manual
1		Analog Discovery 2 - Portable Oscilloscope/Logic Analyzer/Power Supply	Digilent (410-321)	Digi-Key (1286-1117-ND) or Digilent (410-321)	Reference Manual

Chapter 1: Analog vs. Digital

Lab 1.1: Introduction to Lab Equipment & Blinking an LED with an AWG

1.1.1 Objective

The objective of this lab is to become familiar with the equipment we will be using for the rest of the lab exercises and also gain experience with the on/off nature of digital signals. We will use the Arbitrary Waveform Generator (AWG) of the *Analog Discovery* to drive a simple LED circuit to turn it on and off. We will also use the oscilloscope function of the Analog Discovery to view the AWG input and the voltage across the LED.

1.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Breadboard a simple LED-Resistor circuit.
- Use an AWG to output a signal with a specified type, amplitude, offset, and frequency.
- Use an oscilloscope to display a waveform on the screen.
- Describe the on/off behavior of a digital signal.
- Describe the impact that increasing the frequency has on a digital signal.

1.1.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 1x red LED, discrete.
- 1x 150 Ω axial resistor.

1.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate the use of the AWG to make an LED blink at 2 Hz (50% of exercise).
2. Produce an oscilloscope measurement of the LED circuit running at 5 MHz (50% of exercise).

1.1.5 Lab Work & Demonstration

1.1.5.1 Using the AWG to make an LED Blink

Breadboard a Simple LED-Resistor Circuit

You are going to build the LED-Resistor circuit in [Figure 1.1](#). The resistor and LED will be connected on your breadboard. The square wave voltage will come from the Analog Discovery AWG output. For the first deliverable, you will breadboard this circuit and make the LED blink on and off.

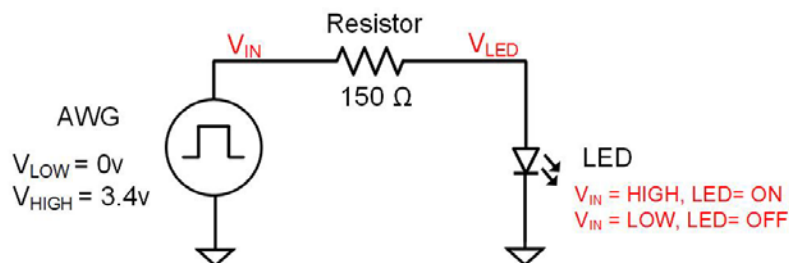


Figure 1.1
Simple LED-Resistor Circuit

A breadboard provides a way to electrically connect circuit components without solder. The breadboard consists of a series of holes designed to accept pins of standard digital logic parts and various switches and LEDs. Within the breadboard are internal connections that allow components to be electrically connected by inserting pins within the same hole set. [Figure 1.2](#) shows an overview of the main connection scheme of a breadboard.

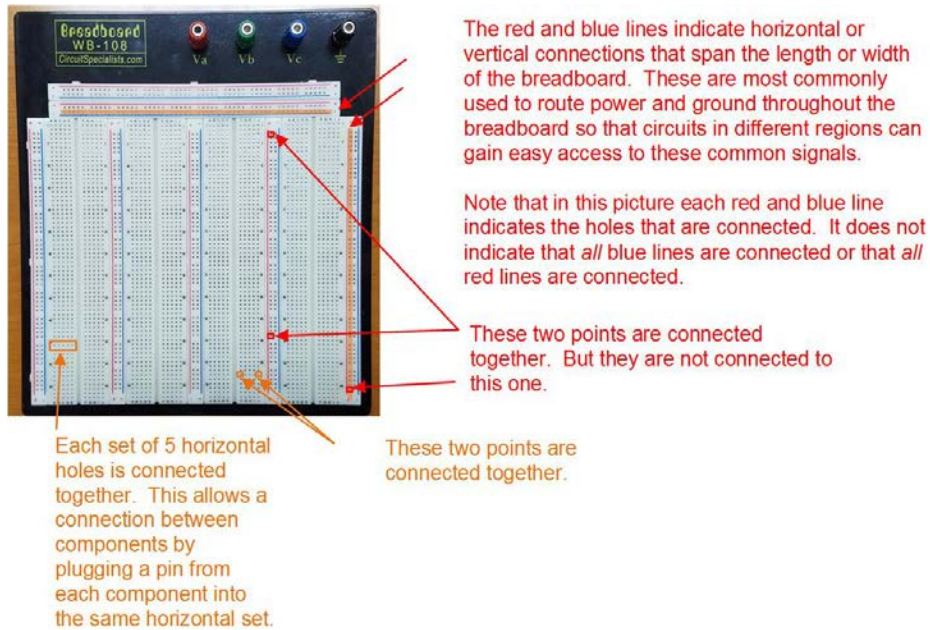


Figure 1.2
Overview of a Breadboard

For these lab exercises, it will be helpful if all of the vertical red lines are connected together and all of the vertical blue lines are connected together. This can be accomplished by wiring each red vertical connection into one of the red horizontal strips along the top and the same for the blue vertical connections. This will allow power to be provided to the entire board by connecting it into any of the holes on red vertical strips and ground to be provided to the entire board by connecting it to any of the holes on the blue vertical strips. Enter wires into your breadboard to electrically connect the vertical red/blue strips together as shown in [Figure 1.3](#).

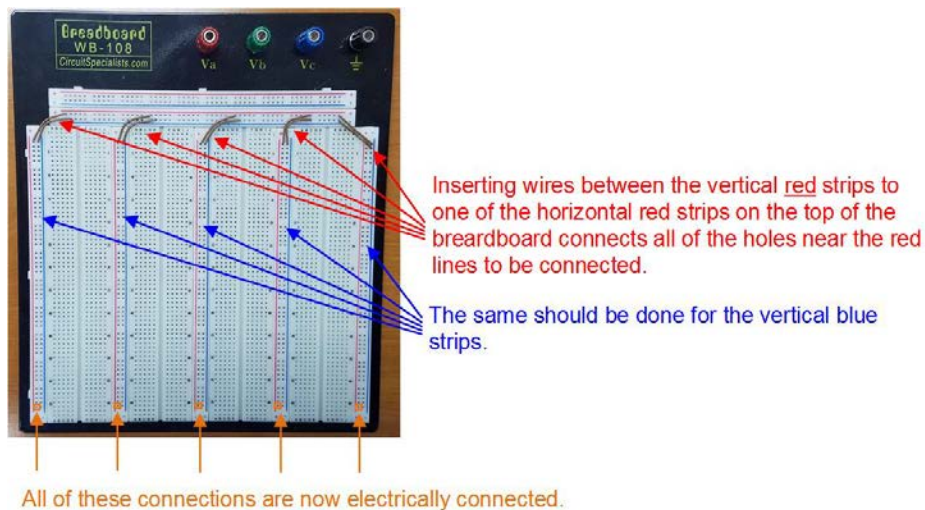


Figure 1.3
Setting up the Power/Ground Strips of the Breadboard

The breadboard is now ready to implement the LED-resistor circuit. The first step is to locate a $150\ \Omega$, axial resistor from your parts kit. The term *axial* means that the resistor resides on the same axis as the leads that are used to connect to it. The colors on the resistor's ceramic body indicate the value and tolerance of the resistor. Figure 1.4 shows an image of a $150\ \Omega$ axial resistor and a standard color code chart. The code uses one color for the first digit of the resistor value and a second color for the second digit of the resistor value. This is then followed by a color representing a *multiplier* to find the final value of the resistor. A $150\ \Omega$ resistor has a code of brown (1) – green (5) – brown (10^1). This can be thought of as forming the number “15” using the first two colors and then multiplying it by “10” to get 150. The final color on the resistor is the tolerance. All of the axial resistors in the parts kit are $\pm 5\%$, so all tolerances are indicated with the color gold.

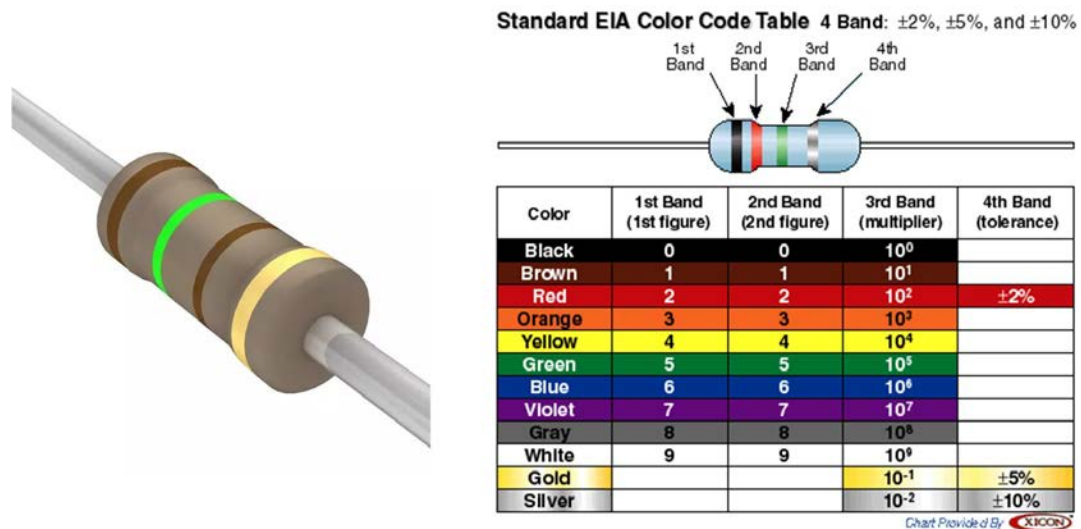
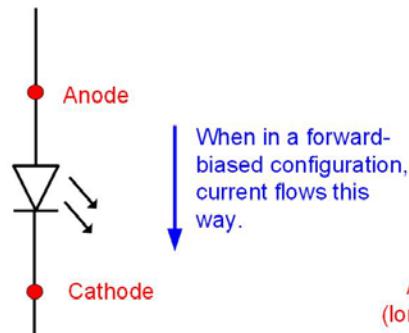


Figure 1.4
Determining the Value of an Axial Resistor (150 ohm example)

The next step is to locate the red LED from the parts kit and understand its *polarity*. The term polarity means that the pins of a part have different functions and that the orientation of the part matters when it is placed into the breadboard. An example of a part that is *not polarized* is the $150\ \Omega$ resistor discussed above. This part can be placed in a circuit in any orientation and it behaves the same. This is not the case with an LED. For all of the circuits in this lab manual, we will use LEDs in a *forward-biased* configuration. This simply means that the current will flow from the **anode** to the **cathode** of the LED. When current flows in this direction, the LED will turn on. Current will not flow in the opposite manner. What is important when using a discrete LED is to determine which pin is the anode and which is the cathode. In the discrete LED in our parts kit, the anode is always the longer pin while the cathode is the shorter pin. Figure 1.5 shows a graphical depiction of the polarity of discrete LED.

The symbol for an LED indicates which pin is the anode and which is the cathode.



The lengths of the pins on an LED part indicate which pin is the anode and which is the cathode.

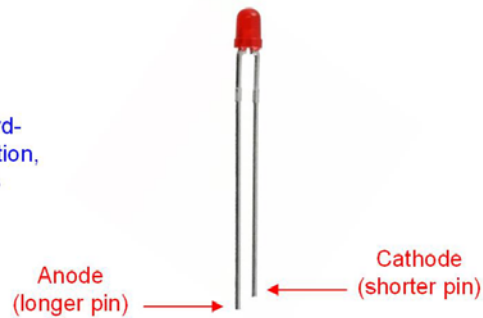


Figure 1.5
Determining the Anode and Cathode of the Discrete LEDs in our Parts Kit

Now that we have the two components for our circuit, we can insert them into the breadboard to form their electrical connections. [Figure 1.6](#) shows an example of how to breadboard this circuit.

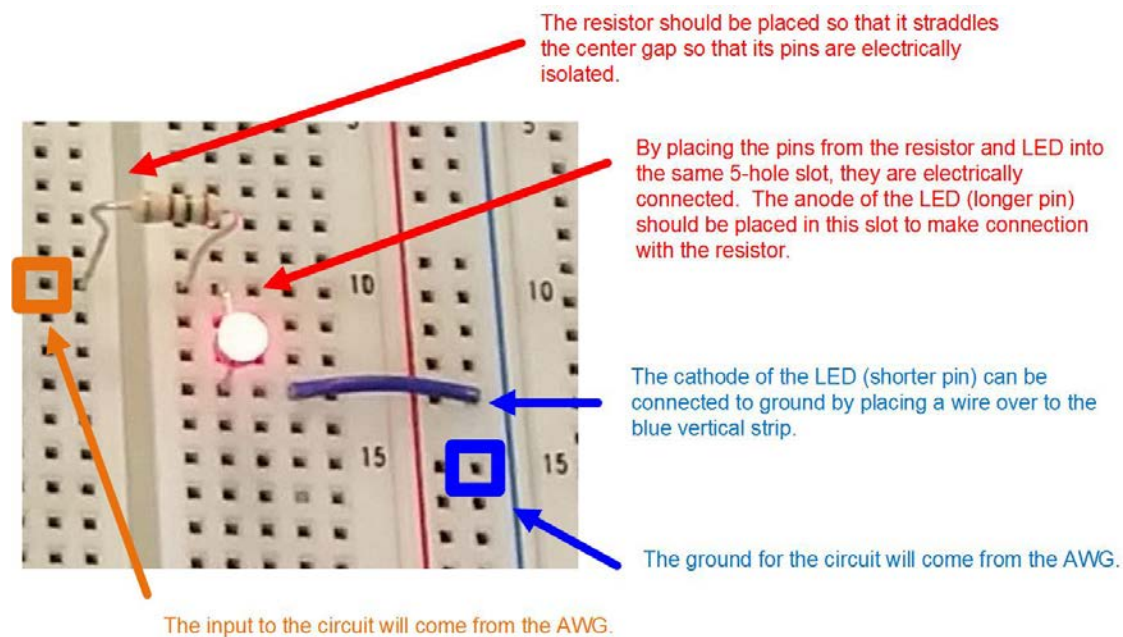


Figure 1.6
Wiring the LED-Resistor Circuit

Now that the resistor and LED are in place, we can connect the AWG of the Analog Discovery. An AWG provides an input signal to a circuit that can have a user-defined shape (i.e., square, sinusoid, triangle, etc.). The amplitude and offset of the waveform can also be defined. The Analog Discovery has two AWG outputs (W1 and W2). We will be using W1 in this exercise. The signal labels for the Analog Discovery are located on the plastic body near the main 30-pin connector. This is shown in [Figure 1.7](#).

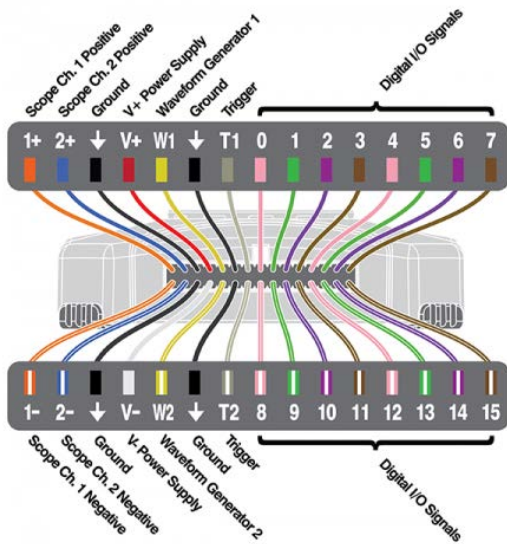


Figure 1.7
Pinout for the Analog Discovery 2

A wire kit is provided with the Analog Discovery that allows easy connections to external circuits. On one side of the wire kit is the mating connector that plugs into the 30-pin connector on the Analog Discovery’s main body. On the other end, each signal is brought out to a 0.1” square, female receptacle. This receptacle is ideal for interfacing to a breadboard using either 0.1” header pins or simple breadboard wires. At this point, plug channel W1 into the breadboard using a 0.1” header pin to make the input connection shown in Figure 1.1. All electrical circuits need a *ground*, or a return path for the circuit to flow back to the source. The Analog Discovery has four ground signals in its wire kit indicated with the ↓ symbol. At this point, also connect one of the grounds from the Analog Discovery to the ground of the breadboard. Figure 1.8 shows the connection of the Analog Discovery to the breadboard containing the LED-resistor circuit. After these connections are made, we are ready to configure the AWG output to drive the LED circuit to make it turn on and off.

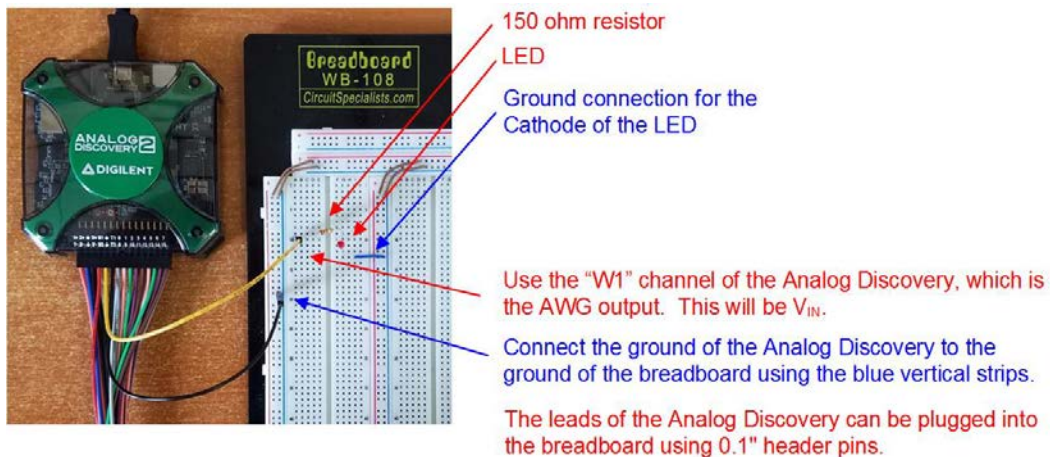


Figure 1.8
Connection between the Analog Discovery and the LED-Resistor Circuit on the Breadboard

[Install the Waveforms 2015 Software to Control the Analog Discovery](#)

Now we want to configure the AWG to drive a square wave that goes from 0v to +3.4v at a frequency of 2 Hz. The Analog Discovery is controlled using a free application called *Waveforms*, which can be downloaded from Digilent.com (<http://store.digilentinc.com/>). Once on this website, on the left select “Scopes, Instruments, & Circuits”, and then on the “Waveforms 2015 (Download Only)” product. On the next screen, select “Download Here”. On the next screen you will find “Latest Downloads” where you can choose “Windows”. The 65MB download will then commence. Once downloaded, run the *.exe file and the software will be installed.

Once installed, connect your Analog Discovery to your computer using the USB cable provided in the box. The USB drivers will install automatically. Wait until the drivers are installed before starting the program.

Launch Waveforms (Start –Digilent – Waveforms). The software will automatically recognize the Analog Discovery and connect. The Waveforms startup window shown in [Figure 1.9](#) will appear.

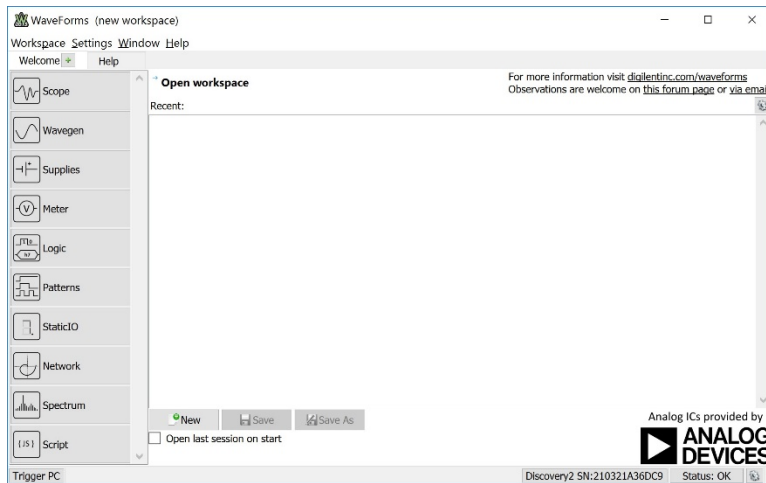


Figure 1.9
Waveforms 2015 Startup Window

[Configure the AWG Output of the Analog Discovery](#)

Click on the “Wavegen” tool on the left of the Waveforms window. A new tab will appear in the Waveforms window called “Wavegen 1”. You will see a graphical image of the waveform that will be created. At this point we want to configure the AWG to output a square wave with a minimum voltage of 0v, a maximum voltage of +3.4v, at a frequency of 2 Hz. The values for frequency, amplitude, and offset can either be typed in or entered using the drop-down menus.

The term “amplitude” refers to the amount that a signal swings above and below an “offset”. This means in order to produce a signal that goes from 0v to +3.4v, we want to set the amplitude to +1.7v with an offset of +1.7v. This can be thought of as a signal centered at 1.7v that will go above this center by +1.7v (i.e., $1.7v + 1.7v = 3.4v$) and go below this center by +1.7v (i.e., $1.7v - 1.7v = 0v$). Set the following values for the Wavegen.

- Type = Square
- Frequency = 2 Hz
- Amplitude = 1.7 V
- Offset = 1.7 V
- Symmetry = 50%
- Phase = 0°

Your AWG setup should look like [Figure 1.10](#). Note that the background can be changed from *Dark* to *Light* using the setup gear button in the upper right corner of the waveform. You can also change the thickness of the line. Making the background light and the line thicker makes taking and printing screenshots easier.

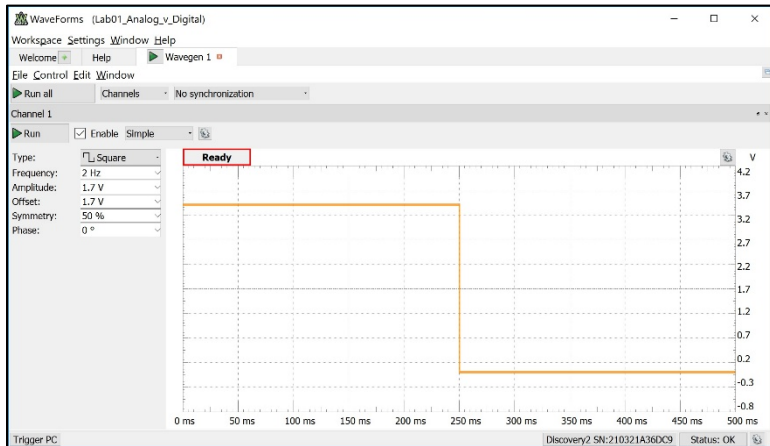


Figure 1.10
Waveforms AWG Setup Window

At this point, you should save your workspace to your computer. Click on the menu “Workspace – Save”. Browse to a location on your computer you wish to save your lab work and click “Save”. You can save the workspace continually as you configure settings throughout this exercise.

[Run the AWG to Make the LED Blink](#)

To run the AWG, you simply press the “Run” button in the Waveforms application. There is a “Run” and “Run All”. The “Run All” is used when there are multiple tools configured. In this case, there is only one tool setup so both buttons will do the same thing. The LED should now be blinking on and off at a rate of 2 Hz, or 2 times per second. Take a short video (<5 s) showing the operation of your circuit with the blinking LED. **This video satisfies the requirements for deliverable #1.**

1.1.5.2 [Measure the Logic Signals in the LED Circuit](#)

[Connect the Oscilloscope to Measure the Input and Output Voltages of the Circuit](#)

An *oscilloscope* is an instrument that displays an electrical signal graphically. It is one of the most commonly used instruments to debug electrical circuits. The Analog Discovery contains two oscilloscope channels. Each channel has two wires that must be connected (1+/1- and 2+/2-). The 1- and 2- channels will always be connected to ground for the exercises in this manual. We want to measure the input and output signals of the LED-resistor circuit. [Figure 1.11](#) shows the connection points for the oscilloscope measurement.

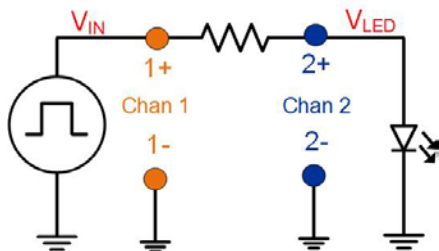


Figure 1.11
Connection Points for the Oscilloscope Measurement

In order to make these connections, use 0.1” header strips to plug 1+ into the same breadboard strip as the AWG input and 2+ to the same breadboard strip as the Anode of the LED. The 1- and 2- should be connected to the ground of the breadboard. [Figure 1.12](#) shows how the connection should look.

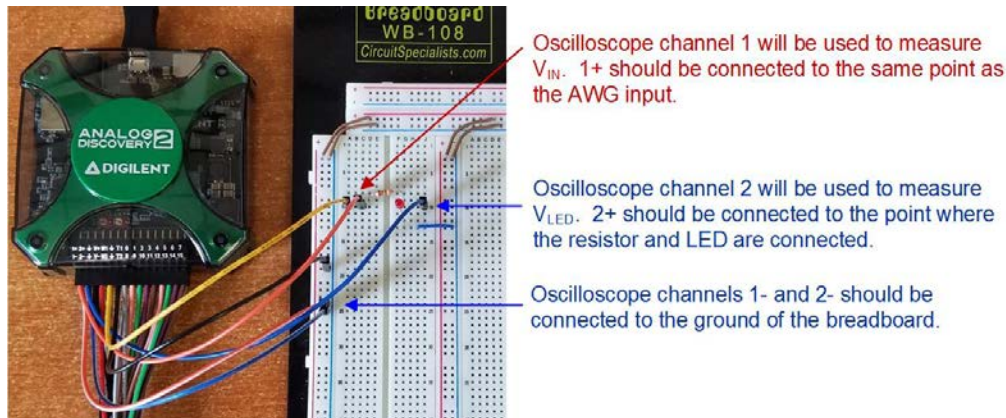


Figure 1.12
Connection between the Analog Discovery and the LED-Resistor Circuit on the Breadboard

Setup the Oscilloscope of the Analog Discovery

In the Waveforms application, stop the AWG by clicking the *Stop* button. Click on the “Welcome” tab to go back to the tool listing. Click on the “Scope” tool. A new tab will appear called “Scope 1”. Go to this tab and you’ll see a measurement screen. Both channels of the oscilloscope will be displayed on this screen by default.

On the right side of the screen you’ll see the zoom controls. An oscilloscope sets the zoom using *divisions*. The vertical axis is always voltage, and is measured in *volts/division*, or *V/div*. Each line on the measurement screen is a division. The offset of the measurement can also be configured. For both channel 1 and 2, set the zoom controls to:

- Offset = 0 V
- Range = 1 V/div

On the right side of the screen you’ll also see zoom controls for *time*. Again, both a scaling per division is given (base) and a horizontal offset (position). Oscilloscope are usually used for signals that are fast enough that they can’t be observed with the human eye. In this part, we will take a measurement on the LED-resistor circuit when it is running at a faster frequency (1 kHz) so we need to configure the time zoom accordingly. Configure the time control to:

- Position = 0 s
- Base = 1 ms/div

When signals are too fast to be seen with the human eye, simply displaying what the oscilloscope is measuring on the screen would result in the entire screen being lit up. To handle displaying fast repetitive signals, an oscilloscope uses a *trigger*. A trigger represents an event that occurs on the incoming signal, such as a rising edge passing a certain voltage level. When this occurs, the oscilloscope positions all of its recorded data on the screen with the trigger moment located at $\text{time}=0\text{s}$. As the oscilloscope continues to run, it will continually trigger and overwrite the data on the screen with the new set of data positioned with the trigger at $\text{time}=0\text{s}$. If the signal is repetitive, the resulting screen will show a steady waveform in which the characteristics of the signal can be determined. We want to setup the trigger so that every time Channel 1 has a rising transitions that passes through 2v, the oscilloscope will trigger. Along the top of the measurement screen there are a variety of trigger settings. Configure these as follows:

- Mode = Auto
- Source = Channel 1
- Condition = Rising
- Level = 2 V

Run the Oscilloscope to Measure the Input and Output Voltages of the LED-Resistor Circuit

Now we are ready to take an oscilloscope measurement. First, we need to set the frequency of the AWG to 1 kHz and turn it back on. Go back to the “Wavegen 1” tab, change the frequency to 1 kHz, and press the run button. Now

go back to the “Scope 1” tab and press run. You will see the waveform in Figure 1.13. Again, the background and line thickness of the oscilloscope measurement screen can be changed using the gear setup button in the upper right corner of the screen.

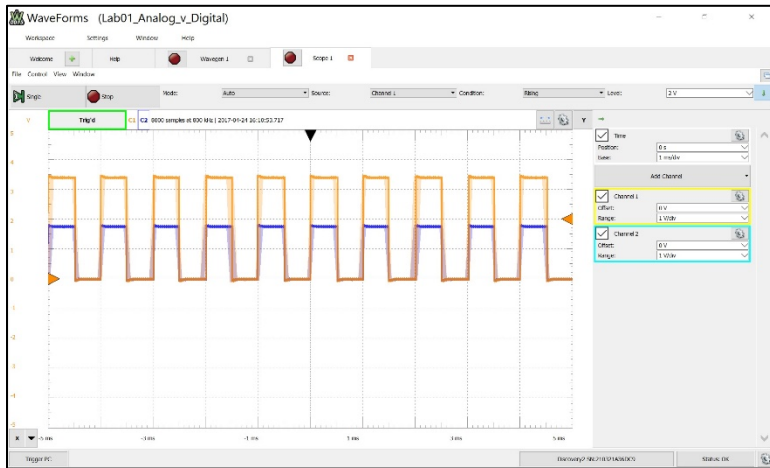


Figure 1.13
Oscilloscope Measurement of 1 kHz Signals

Observe the Impact of Increasing the Frequency of the Input on V_{LED}

Begin increasing the frequency of the input square wave. This can be done in real time without stopping the AWG or oscilloscope. As you increase the frequency of the AWG, you'll need to decrease the time/div setting on the oscilloscope. As an example, increase the frequency of the AWG to 5 kHz. In the oscilloscope, change the time/div (upper right of oscilloscope screen) to 0.2 ms/div so that the square waves are visible.

At 5 kHz, V_{IN} and V_{LED} still look like nice square waves. Increase the AWG frequency to 500 kHz and decrease the oscilloscope time base to 1 us/div. Notice that the first signs of V_{LED} distortion are beginning. Now increase the AWG frequency to 5 MHz with an oscilloscope time base of 0.2 us/div. At 5 MHz the signals are severely distorted. This illustrates that a digital signal can't have its frequency increased indefinitely while still maintaining its pure on/off signal shape.

Take a screenshot of your 5 MHz oscilloscope measurement for your records. In Windows you can do this using either the *Snipping Tool* or by using the *Print Screen* button on your keyboard to place the screen on your clipboard and then pasting it into Paint. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

CONCEPT CHECK

Lab 1.1 After completing this lab exercise, can you:

- Breadboard a simple LED-Resistor circuit?
- Use an arbitrary waveform generator to output a signal with a specified type, amplitude, offset, and frequency?
- Use an oscilloscope to display a waveform on the screen?
- Describe the on/off behavior of a digital signal?
- Describe the impact that increasing the frequency has on a digital signal?

Chapter 2: Number Systems

Lab 2.1: 2-Bit Counter from the AWG and Introduction to Logic Analysis

2.1.1 Objective

The objective of this lab is to demonstrate how logic signals can be interpreted as numbers. This lab will also allow you to gain more familiarity with the arbitrary waveform generator and introduce logic analysis. You will drive two square waves from the AWG into two LED circuits on your breadboard. You will configure the AWG outputs to drive a 2-bit binary counter pattern by adjusting the phase of the signals. You will then measure the LED signals using the logic analyzer within the Analog Discovery. You will then view the 2-bit binary information in both binary and decimal formats within the logic analyzer waveform.

2.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Use an arbitrary waveform generator to output a binary counter pattern by adjusting the phase of the outputs.
- Use the logic analyzer to measure the digital values of a set of signals.
- View the logic analyzer measurement in different bases to see how the logic signals are interpreted as numbers.

2.1.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 2x red LEDs, discrete.
- 2x 150 Ω axial resistors.

2.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate the use of the AWG to create a 2-bit binary counter displayed on two LEDs (50% of exercise).
2. Produce a logic analyzer measurement displaying the counter value in both binary and decimal (50% of exercise).

2.1.5 Lab Work & Demonstration

2.1.5.1 Using an AWG to Create a 2-bit Binary Counter Pattern

[Breadboard Two LED-Resistor Circuits](#)

You are going to build the LED-resistor circuits shown in [Figure 2.1](#). Locate 2x red, discrete LEDs and 2x, 150 Ω axial resistors from your parts kit and breadboard the circuit. The square wave voltages will come from the two AWG channels (W1 and W2) from the Analog Discovery. Connect the W1 and W2 outputs of the Analog Discovery to your circuits using either 0.1" header pins or breadboard wires.

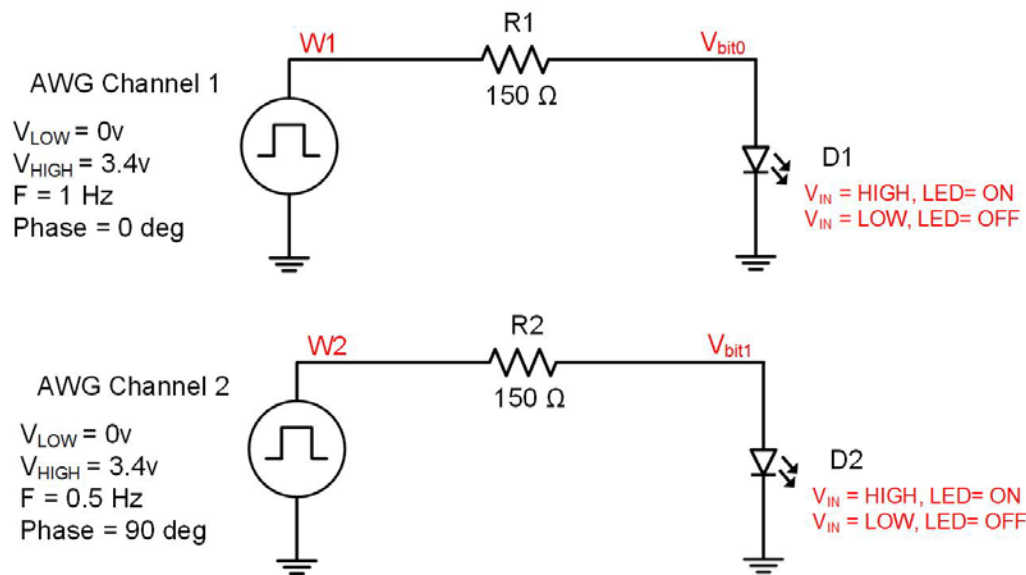


Figure 2.1
LED-Resistor Circuits to Display Binary Counter on LEDs

Figure 2.2 shows how the circuit will look after it is wired.

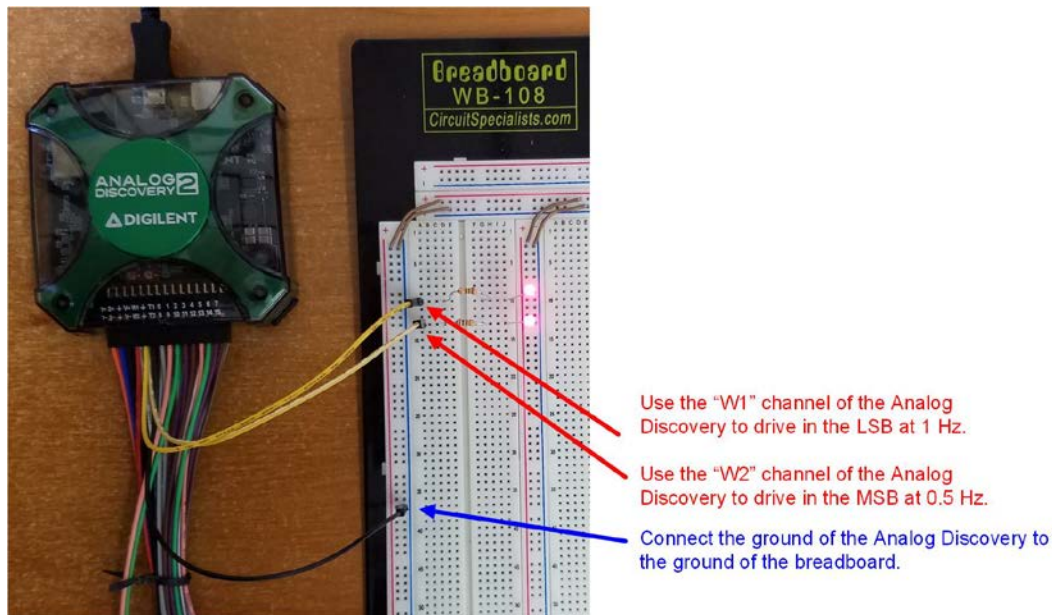


Figure 2.2
Wiring the Two LED-Resistor Circuits

Setup the AWG to Output a 2-Bit Binary Pattern

The Analog Discovery's AWG can output two voltages with different frequencies. By configuring these channels as square waves with one output exactly twice the frequency of the other, a 2-bit binary counter pattern can be created. The only caveat is that the least significant bit (LSB) needs to be phase shifted so that the pattern counts 00 – 01 – 10 – 11 – 00

Launch the Waveforms application. Click on the Wavegen button to bring up the AWG control window. In this exercise we will be using both AWG channels of the Analog Discovery. By default, only Channel 1 appears when the Wavegen tool is launched. In order to turn on Channel 2, use the “Channels” dropdown menu and select “2”. Once this is selected, two waveform plots will appear. To the right of the “Channels” dropdown menu there is another menu that controls whether the two signals are synchronized or unsynchronized. Choose “Synchronized” in this dropdown menu. Now configure **both** waveforms with the following settings.

- Type = Square
- Amplitude = 1.7 V
- Offset = 1.7 V
- Symmetry = 50%
- Phase = 0°

We will use channel 1 (W1) to drive the LSB of the counter pattern. Set the frequency of Channel 1 to **1 Hz**. We will use channel 2 (W2) to drive the MSB of the counter pattern. Set the frequency of Channel 2 to **0.5 Hz**.

The time zoom and time offset for both waveform images can be configured independent of each other. In order to visualize the phase relationship between the two waveforms, we want to set both waveforms to be on the same time scale. In the upper right corner of each waveform window, there are gear setup buttons. Use these buttons to configure the time settings for both plots to the following.

- Scale = Manual
- Length = 0.2 s/div
- Start = 0.5 s

At this point we have the AWG configured nearly correct; however, the pattern that will be driven does not follow a 00-01-10-11 counting pattern. Instead, you will see the pattern in [Figure 2.3](#)

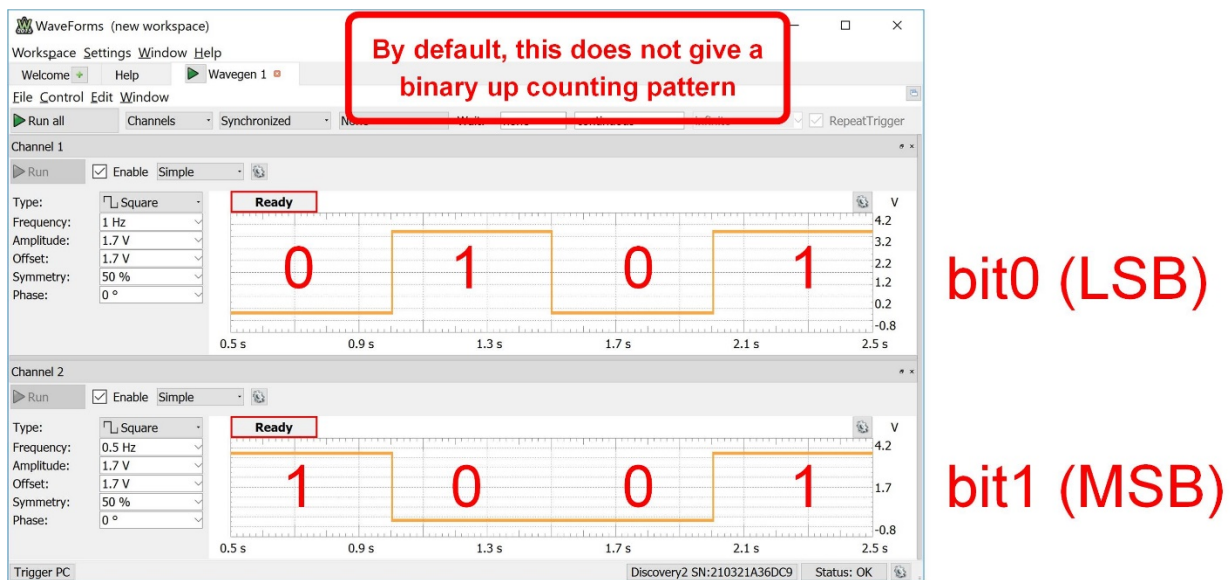


Figure 2.3
AWG Setup to Drive a 2-Bit Counter, but with the Incorrect Phase Relationship

In order to get the correct pattern, we need to configure the phase of channel 2 to 90°. Make this change and you will now see the pattern in [Figure 2.4](#).

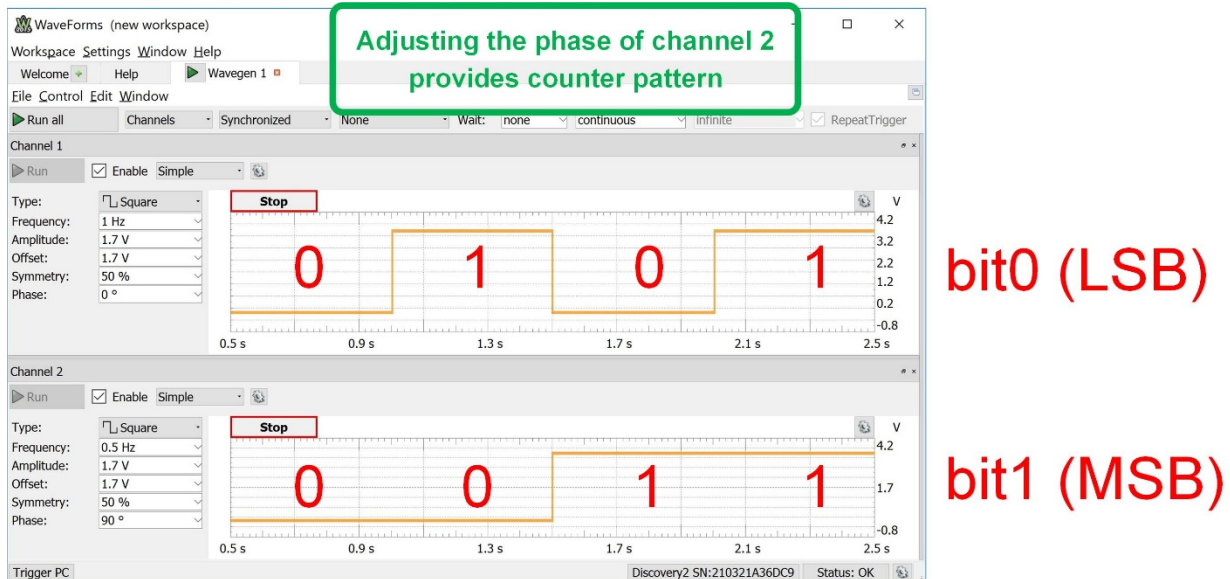


Figure 2.4

AWG Setup to Drive a 2-Bit Counter, but with the Correct Phase Relationship

Run the AWG to Output a 2-Bit Binary Pattern

Now press the “Run All” button. You should see a 2-bit binary pattern on the LEDs of your breadboard. Take a short video (<5 s) showing the operation of your circuit with the blinking LEDs. **This video satisfies the requirements for deliverable #1.**

2.1.5.2 Logic Analyzer Measurement of the Counter Pattern

Connect the Logic Analyzer to your Circuit

A logic analyzer is an instrument similar to an oscilloscope in that it graphically displays the signal being measured. While an oscilloscope displays the detailed analog wave shape of the signal being measured, a logic analyzer only displays the digital value of the signal. Since only 1’s and 0’s are stored in a logic analyzer, the circuitry to implement the instrument is much simpler than an oscilloscope. This allows the logic analyzer to have more channels in the same amount of area as an oscilloscope. The Analog Discovery has a 16-channel logic analyzer. A logic analyzer also uses a trigger to control how the data being measured is displayed similar to an oscilloscope. The difference is that the logic analyzer can trigger on logic values across all signals being measured.

We want to measure the logic values of the 2-bit counter that was implemented on the breadboard. The logic channels of the Analog Discovery are labeled 0-15. Connect channels 0 and 1 of the logic analyzer to the LED-resistor circuit as shown in [Figure 2.5](#).

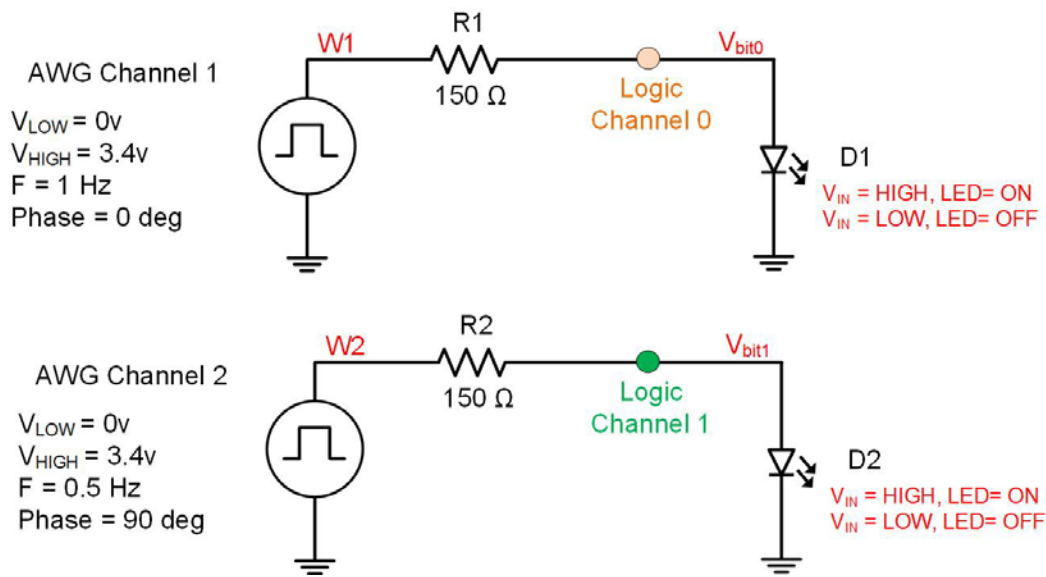


Figure 2.5
 Connection Points for the Logic Analyzer to Measure the 2-Bit Counter

Figure 2.6 shows how the breadboard will look after connecting the logic analyzer.

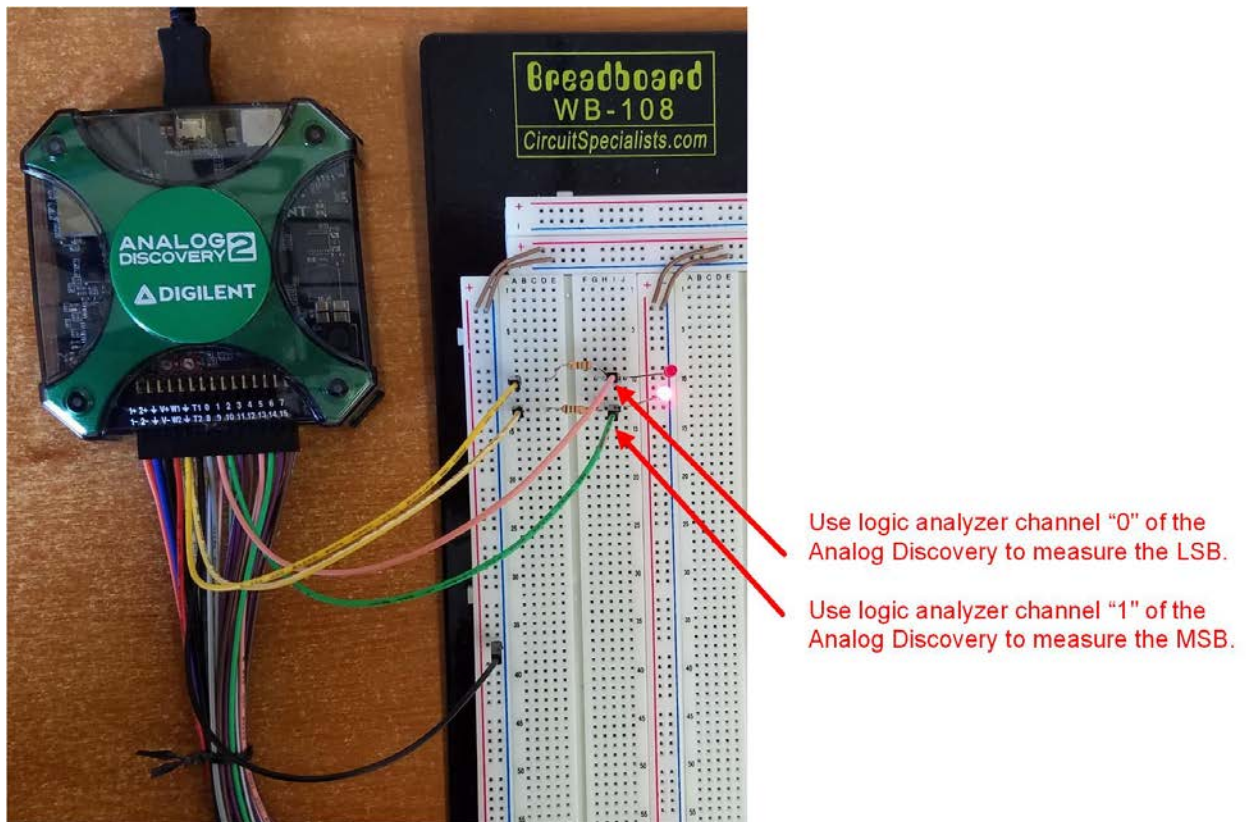


Figure 2.6
 Connecting the Logic Analyzer to the LED-Resistor Circuits

Configure the Logic Analyzer to Measure the 2-Bit Counter

In Waveforms, click on the “Welcome” button to go back to the main workspace screen. Start the logic analyzer by clicking on the “Logic” button on the left. The logic analyzer screen will appear. The first thing to do is define the signals that we are measuring. On the left of the screen there is a section that shows all of the signals in the measurement. Select the “Click to Add channels” button and then choose “Bus”. A *bus* is the term used to describe a group of signals. For digital signals, we can think of a bus as either data or a number formed by individual bits. In this lab exercise, the two bits being driven to the LEDs by the AWG will form a 2-bit number, or a 2-bit *bus*.

In the options window that appears, name the bus *Count*. On the left side of the window there is a list of available logic analyzer channels to include in the measurement. Highlight the two channels “DIO 1” and “DIO 0” by selecting them while holding down the Shift key. Once selected, click on the + sign to add them to the box on the right. These two signals will now form the 2-bit bus called *Count* in the measurement. Change the format to **binary**. Leave all other settings the same for now. Your settings should look like [Figure 2.7](#).

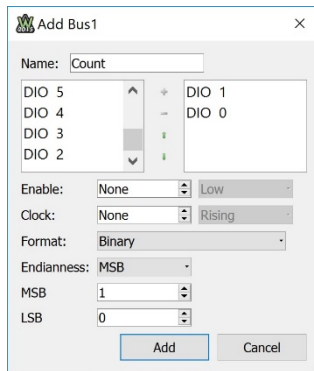


Figure 2.7
Logic Analyzer Bus Setup

Click the “Add” button to close the options window. In the logic waveform screen you’ll now see the *Count* bus in addition to the individual bits of the bus.

Just as with an oscilloscope, a logic analyzer is typically used to measure signals that are faster than the human eye can see. Go back to the Wavegen tab and increase the frequency of **Channel 1 to 1 kHz** and **Channel 2 to 500 Hz**. This will allow us to examine more data at once. Make sure the AWG is still running.

Run the Logic Analyzer to Measure the 2-Bit Counter Pattern

Go back to the Logic tab. Change the time *Base* zoom to 1ms/div (if not already set to this). Press the “Run” button. You should now see data scrolling on the screen. Press the “Stop” button to examine the results. You should see the same results as shown in [Figure 2.8](#). Note that the *Count* bus shows the two bits grouped together as a 2-bit number. The individual bits shown are identical to the individual channels of the AWG.

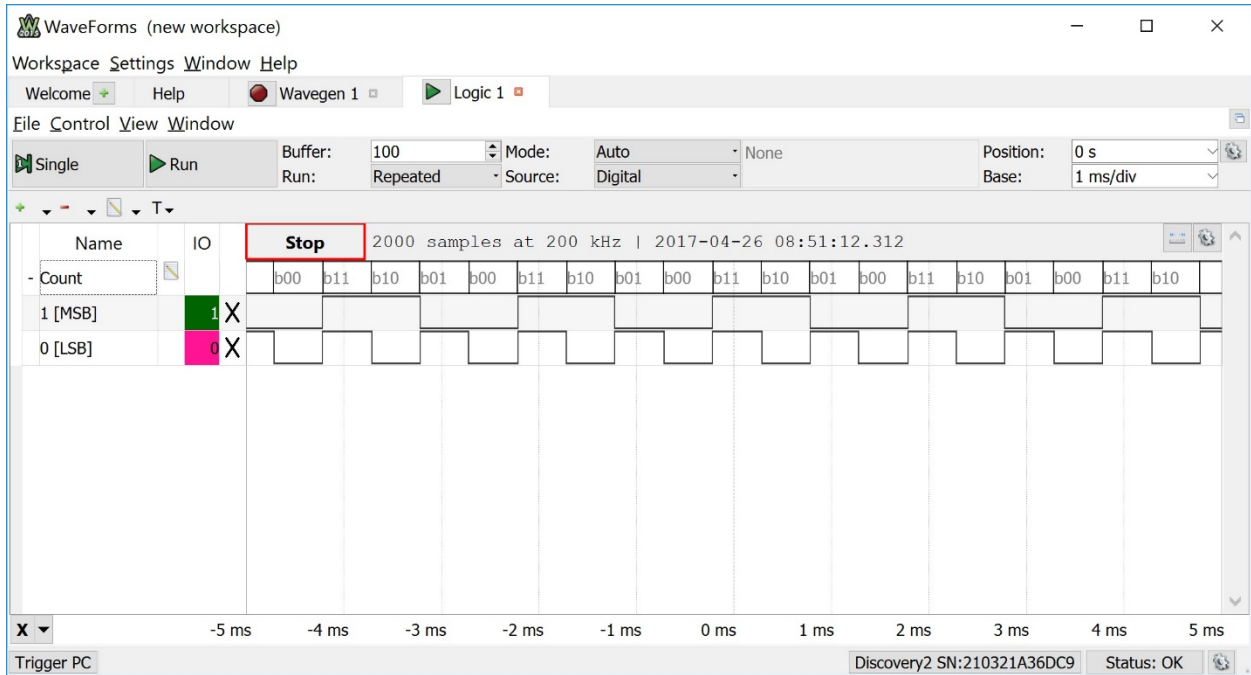


Figure 2.8
Logic Analyzer Measurement of 2-Bit Counter Displayed in Binary Format

Now change the format of the bus to interpret the 2-bits as a decimal number. Double click on the Count label and the settings dialog will appear again. Change the format to **decimal** and press “OK”. You will now see the results as shown in Figure 2.9.

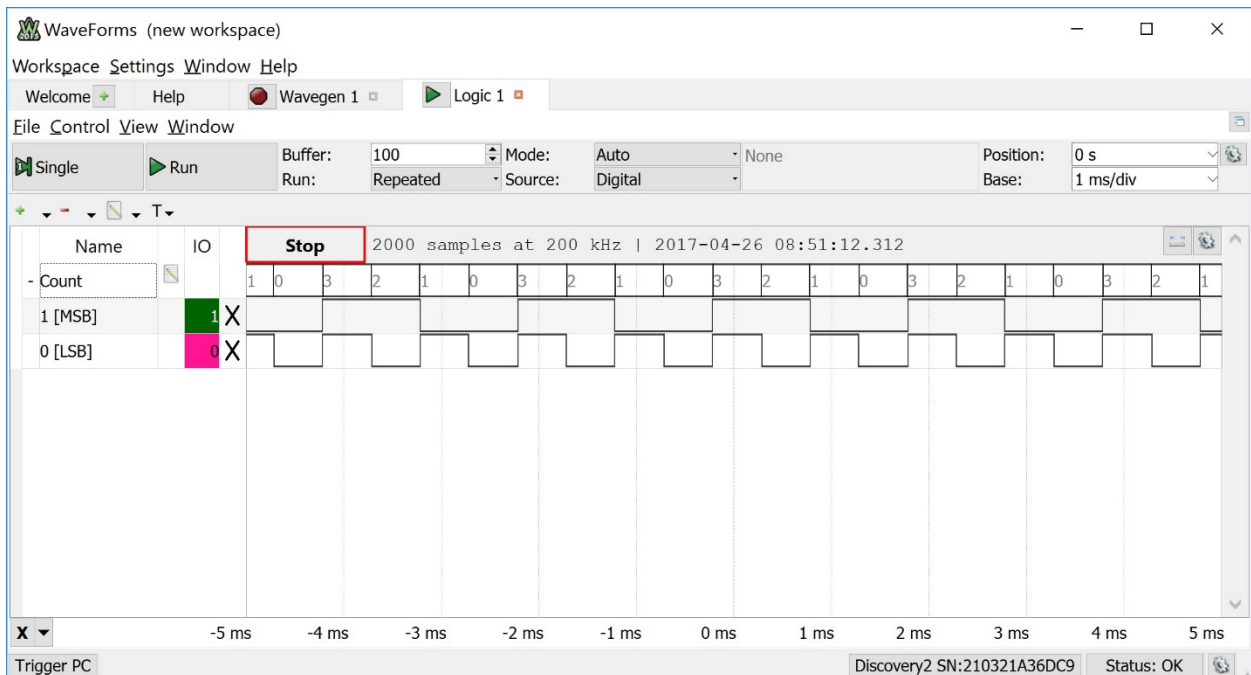


Figure 2.9
Logic Analyzer Measurement of 2-Bit Counter Displayed in Decimal Format

The logic analyzer displaying the 2-bits in decimal is an example of how digital signals are representations of data and as long as the receiving system knows how to interpret the representation, the data can be successfully transmitted. In this case, the logic analyzer is set to treat the 2-bits as a positive decimal number. This is just one example of how to interpret the data. Other coding schemes can be used to transmit information such as signed numbers, characters, or other forms of data.

Take a screenshot of the logic analyzer measurement displaying the 2-bits in decimal format. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

CONCEPT CHECK

Lab 2.1 After completing this lab exercise, can you:

- Use an AWG to create a 2-bit binary counting pattern?
- Use a logic analyzer to measure the digital values of signals in a circuit?
- View the logic analyzer results in different bases to see how logic signals are interpreted as numbers?

Chapter 3: Digital Circuitry & Interfacing

Lab 3.1: Digital Circuit Operation

3.1.1 Objective

The objective of this lab is to demonstrate the DC and AC operation of digital circuits. This lab will also introduce the power supply functionality of the Analog Discovery. You are going to breadboard three simple logic circuits from the 74HC logic family (an inverter, an AND gate, and an OR gate). You will provide power to each of the gates using the power supply output (+3.4v) of the Analog Discovery. You will then perform a series of activities to examine the operation of the logic gates. First, you will determine the input/output specifications for the 74HC logic family by reading a datasheet. Next, you will use the arbitrary waveform generator to drive in all possible input codes to the gates and observe the input/output behavior of the gates using the logic analyzer. Next, you will observe the analog input/output behavior of the inverter using the oscilloscope and examine the impact of an input signal that does not meet the DC specifications of the receiving gate. Finally, you will measure the propagation delay and transition time of the inverter using the oscilloscope.

3.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Read and understand the pin out, DC, and AC specifications of a basic gate from its data sheet.
- Breadboard basic gates from the 74HC logic family.
- Use the power supply output of the Analog Discovery to provide +3.4v to your breadboard.
- Measure the DC behavior of an inverter, AND gate, and OR gate using an AWG and logic analyzer.
- Measure the AC behavior of an inverter using the AWG and oscilloscope.
- Measure the impact of an input signal that does not meet the DC specifications of the receiving gate.
- Measure the switching characteristics of a basic gate using an AWG and an oscilloscope.

3.1.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 1x 74HC04 inverter IC.
- 1x 74HC08 2-input AND gate IC.
- 1x 74HC32 2-input OR gate IC.

3.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Provide the operating specifications for a 74HC logic gate by reading its data sheet (5% of exercise).
2. Demonstrate the DC operation of a 74HC inverter, AND gate, and OR gate using the AWG and logic analyzer (45% of exercise).
3. Demonstrate the AC operation of a 74HC inverter using the AWG and oscilloscope (15% of exercise).
4. Demonstrate the impact of an input signal not meeting the minimum input specifications of the 74HC inverter (15% of exercise).
5. Measure the AC characteristics of the 74HC inverter (transition time and propagation delay) using the AWG and oscilloscope (20% of exercise).

3.1.5 Lab Work & Demonstration

3.1.5.1 Determine the Operating Specifications of a 74HC Logic Gate from its Data Sheet

You are going to breadboard three basic gates from the 74HC logic family. Whenever you are going to use a new part, the first thing you do is examine its data sheet. Retrieve the data sheets for the 74HC04 inverter, 74HC08 AND gate, and the 74HC32 OR gate. We will be using the *dual inline package* (DIP). Examine the pinouts for each part. Notice that the pinouts for these parts includes V_{CC} and GND. Recall that logic gates are active devices, meaning that they require power to operate. In order for the logic gates perform correctly, you must provide a power supply and ground to each part. We will do this using the power supply feature of the Analog Discovery.

The next information you should find is the DC operating conditions for these gates. We will be providing $V_{CC}=+3.4v$. For this logic family, the best-case output (V_{OH-max} and V_{OL-min}) and input (V_{IH-max} and V_{IL-min}) conditions will be a HIGH= $V_{CC}=+3.4v$ and LOW=GND=0v. We are more interested in finding the worst-case specifications (V_{OH-min} , V_{OL-max} , V_{IH-min} and V_{IL-max}). Use the data sheets to determine the following DC specifications for each part. Note that these specifications are the same for all three parts since they are from the same logic family so you can find these for just the HC04 inverter. Also note that since these specifications are only provided for supply voltages of $V_{CC}=+2v$, $+4.5v$, and $+6v$, you will need to interpolate the information to estimate the values for $V_{CC}=+3.4v$ (see chapter 3 in the textbook). For the output specification, you can use the +/- 20uA test condition. Record the following specifications.

Worst Case Outputs	V_{OH-min}	= _____
	V_{OL-max}	= _____
Worst Case Inputs	V_{IH-min}	= _____
	V_{IL-max}	= _____

The next information you should find is the AC operating conditions for the parts. Go to the switching characteristics portion of the data sheet for the 74HC04 inverter. Record the maximum values for t_{pd} and t_t for our power supply voltage. Note that again these specifications are given for three different power supplies, none of which we are using. Instead of interpolating, you can infer what the values will be for a $V_{CC}=+3.4v$. Notice how the values for $V_{CC}=+6v$ and $V_{CC}=+4.5v$ are very close to each other while the values for $V_{CC}=+2v$ is much larger. This is showing how when $V_{CC}=+2v$, the part is on the edge of not operating properly. The value for $V_{CC}=+3.4v$ will be only slightly higher than for $V_{CC}=+4.5v$. Record your best guess of what the values will be based on the values given for $V_{CC}=+6v$ and $V_{CC}=+4.5v$. Note that these represent the maximum value that you should see, but the value depends greatly on what load is connected to it. We will measure these values for our load type in a later activity.

Maximum Propagation Delay	t_{pd}	= _____
Maximum Transition Time	t_t	= _____

Record your values from above electronically. If you printed this page and manually wrote in the values, you can either scan the page or take a photo of the page and save as a JPG. If you recorded the values electronically, take a screen shot of your values and save in JPG format. **This image satisfies the requirements for deliverable #1.**

3.1.5.2 DC Operation of 74HC Logic Gates

Breadboard the Three Base Gate Circuits

Now place the 74HC04, 74HC08, and 74HC32 parts on your breadboard. First connect their power supply and ground pins to the supply rails on your breadboard using jumper wires. The power supply for your breadboard will come from the power supply output of the Analog Discovery (V+). This supply should be wired into your power supply rails of your breadboard (red) using a 0.1" pin or jumper wire. Note that the Analog Discovery has two power supplies, V+ and V-. We will **not** be using the V- supply. The ground should be wired into your ground rails of your breadboard (blue) using a 0.1" pin or jumper wire. This will allow easy access to power and ground for each of the basic gates.

You are going drive in the input codes for each of the three gates using the AWG. Since there are only two AWG channels on the Analog Discovery, you will need to do this one gate at a time (INV, then AND, then OR). For the inverter, you will connect the AWG channel W1 to pin 1 of the 74HC04 part. For the AND gate and OR gate you will connect W1 to the A input of the gate and W2 to the B input of the gate. You will need to connect the output of the gate

being measured to the input of another gate on its same package to provide a typical load configuration (see the following figure). You will connect the logic analyzer to the input(s) and output of each gate in turn in order to observe the DC logic behavior. Figure 3.1 shows the breadboard setup and wiring diagram for the basic gate circuits.

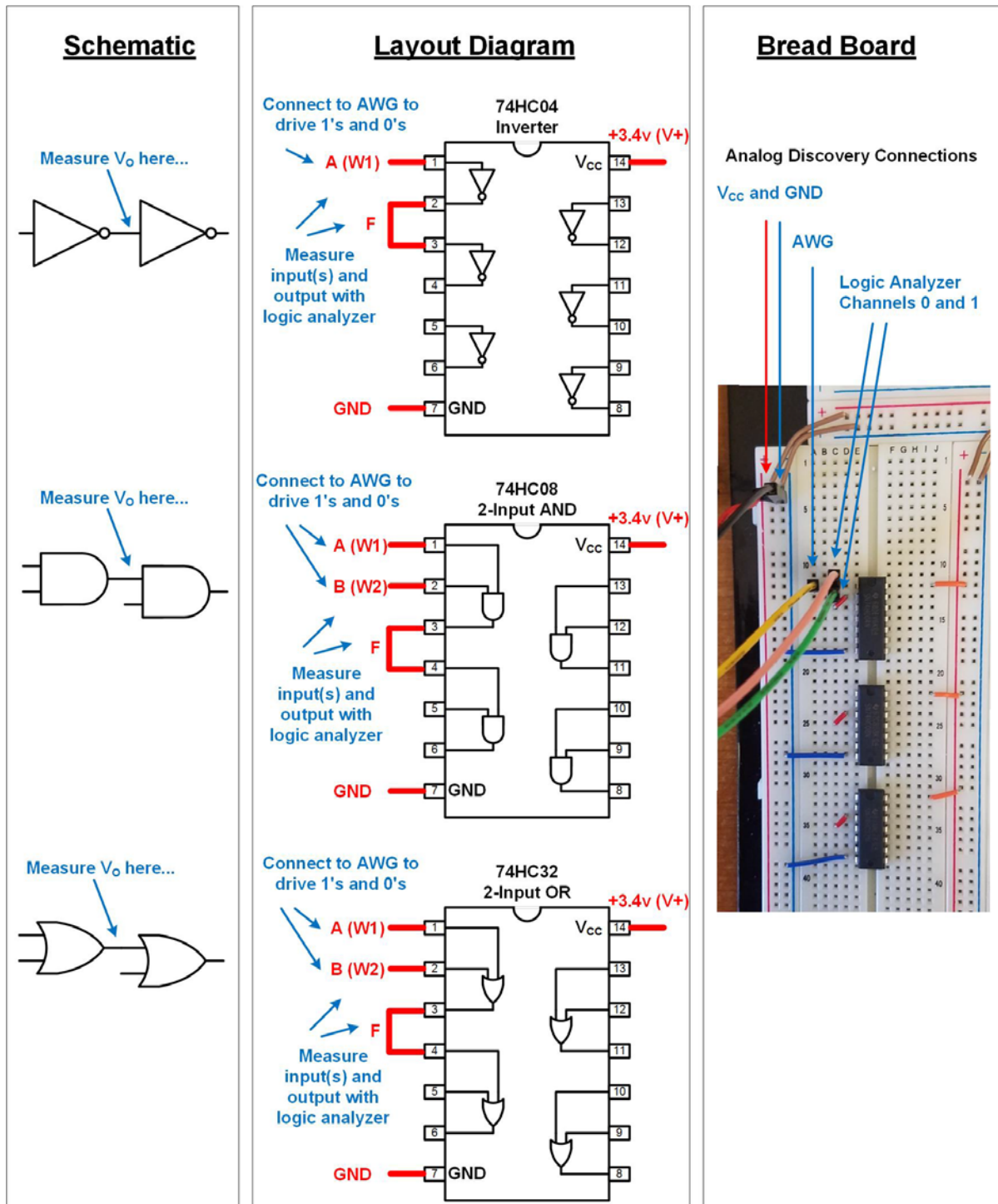


Figure 3.1
Wiring Diagram for Breadboarding the Basic Gate Circuits

Setup the Power Supply

You now need to setup the power supply for the gates. Launch the Waveforms 2015 software. In the main screen, click on the “Supplies” icon. This brings up the power supply window. The Analog Discovery has two power supply channels (V+ and V-). V+ can take on positive supply values while V- can take on negative supply values. The channels can be individually enabled by clicking on the channel labels. To turn on the power supply, you click on the “Run” button. Configure the voltage for V+ to +3.4v by typing the value in on the right dialog of the window. Turn off V- by clicking on the label “Negative Suply (V-)”. Click on the “Run” button at the top to power your breadboard. You should see the window in [Figure 3.2](#) once configured. At this point your logic gates are all powered.

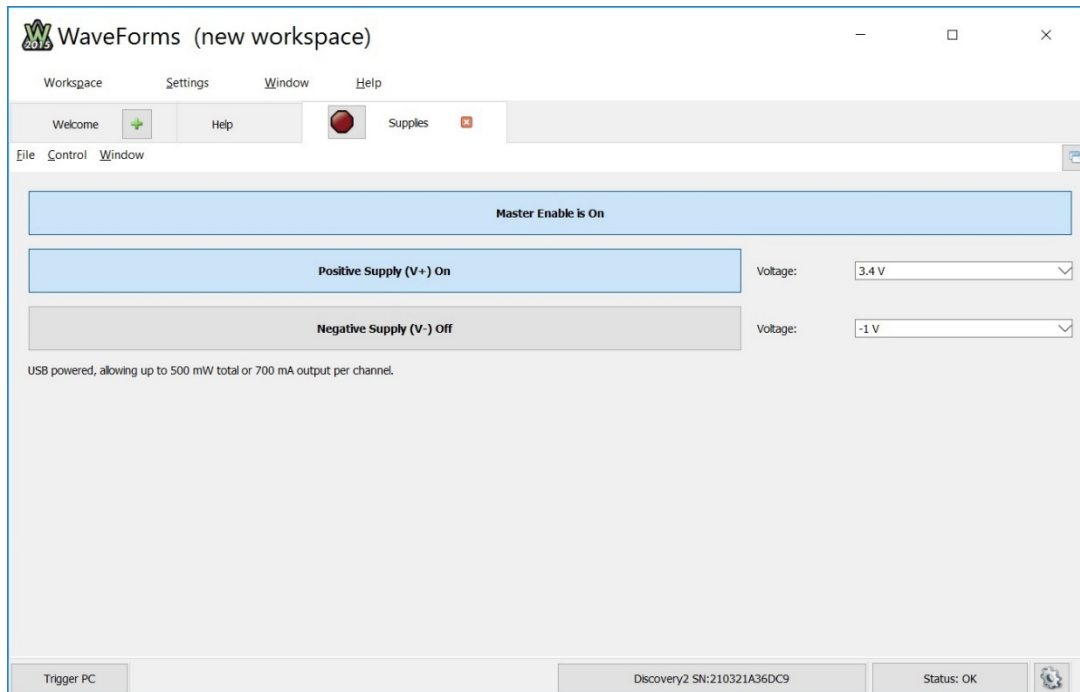


Figure 3.2
Power Supply Setup

Setup the AWG

We are now going to configure the AWG to output a square wave to drive the inputs to the logic gates. Recall that since the AWG of the Analog Discovery only has two channels, we will drive and measure each gate individually. We will first setup the inverter measurement completely and then repeat the process for the AND gate and OR gate.

In the Waveforms window, click on the “Welcome” button to go back to the main screen. Click on “WaveGen” and configure Channel 1 of the AWG as follows:

- Type = Square
- Frequency = 1 kHz
- Amplitude = 1.7 V
- Offset = 1.7 V
- Symmetry = 50%
- Phase = 0°

Press the “Run” button. You are now driving in a logic signal into the input of the inverter.

Setup the Logic Analyzer

We are now going to take a DC measurement on the inverter. The term “DC” refers to that we are not looking at the transition region of the signal but rather its value once it reaches steady state. So while technically the signals we are measuring are toggling between a logic HIGH and LOW, we call this measurement DC because we are really only concerned about the signal value once it reaches and stays at its final value.

In the Waveforms window, click on the “Welcome” button to go back to the main screen. Click on “Logic” and configure the logic analyzer to measure the input to the inverter on DIO 0 and the output of the inverter on DIO 1. When you add the channels to observe, you will click on the “+” button and then select “Signal”. This is as opposed to creating a Bus as in the prior lab. Add DIO 0 first and name it “A”. Add DIO 1 next and name it “F”. Change the Position to 0s and the Base to 0.5 ms/div.

Run the Logic Analyzer Measurement on the Inverter

Press the “Run” button on the logic analyzer. You should see the input and output of the inverter on the screen. Press the “Stop” button to analyze the results. Hint: You can also click the “Single” button to simply fill the screen with data instead of running and then stopping. You should see the waveforms as shown in [Figure 3.3](#).

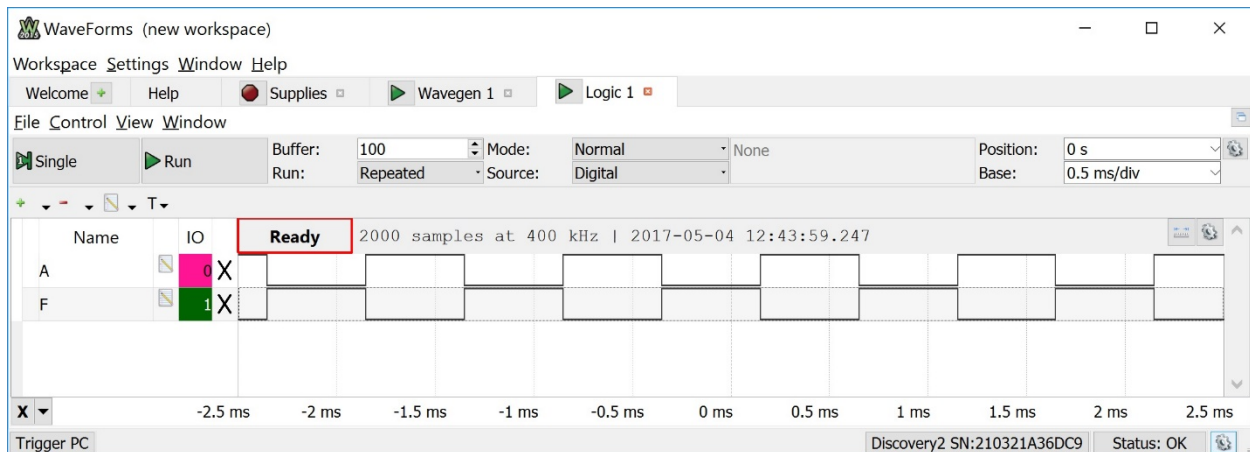


Figure 3.3
Logic Analyzer Measurement of the Inverter

At this point you have successfully wired an inverter, provided it power and ground, provided a logic input, and measured its proper operation by observing the input and output. Take a screenshot of the logic analyzer measurement for your records and save in a JPG format. **This image partially satisfies the requirements for deliverable #2.**

Setup and Run the Logic Analyzer Measurement on the AND Gate

You are now going to repeat this measurement on the AND gate. You first need to connect both channels of the AWG to the inputs of the AND gate on pins 1 and 2. You should then configure the AWG to output a 2-bit counting pattern (refer to lab 2.1 for how to setup the AWG to produce a counting pattern). You will next need to connect logic analyzer channels 1 and 2 to the inputs of the AND gate and channel 3 to the output of the AND gate. Configure the logic analyzer to have DIO 0 observe the channel 1 input, DIO 1 observe the channel 2 input, and DIO 2 observe the output of the AND gate. Label DIO 0 “A”, DIO 1 “B”, and DIO 2 “F”. Run the logic analyzer. You should see the measurement shown in [Figure 3.4](#).

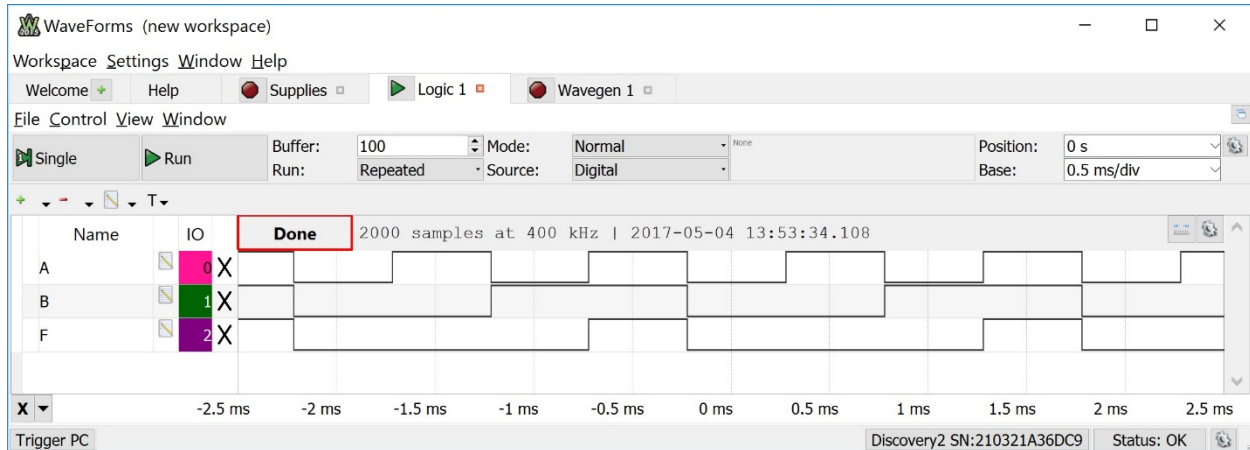


Figure 3.4
Logic Analyzer Measurement of the 2-Input AND Gate

At this point you have successfully wired a 2-input AND gate, provided it power and ground, provided logic inputs, and measured its proper operation by observing the inputs and output. Take a screenshot of the logic analyzer measurement for your records and save in a JPG format. **This image partially satisfies the requirements for deliverable #2.**

[Setup and Run the Logic Analyzer Measurement on the OR Gate](#)

You now are going to repeat this measurement on the OR gate. You will need to move the AWG channels to the inputs of the OR gate. You will also need to move the three channels of the logic analyzer to the OR gate. Since you have already setup the Analog Discovery, you should be able to just press “Single” on the logic analyzer. You should see the measurement shown in [Figure 3.5](#).

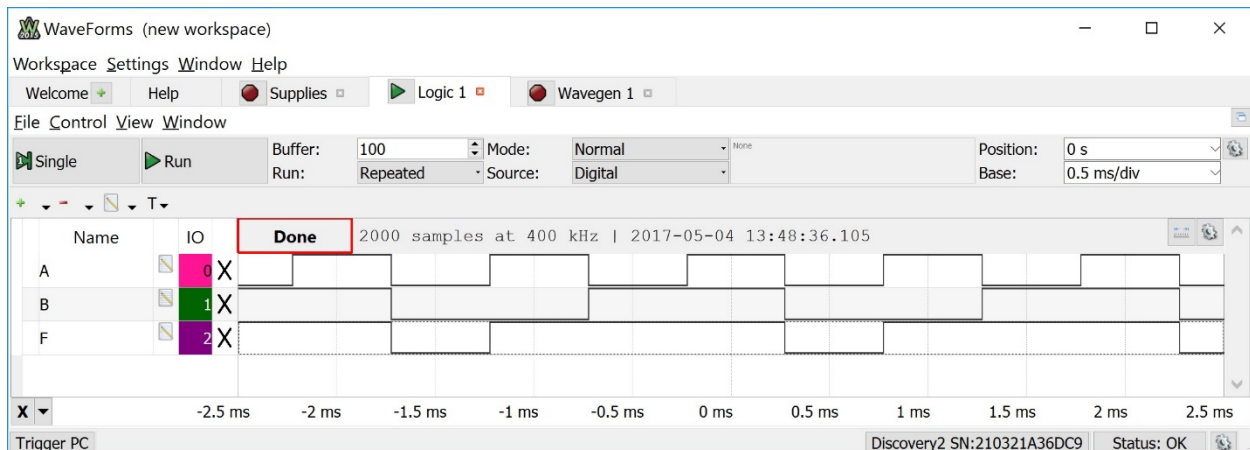


Figure 3.5
Logic Analyzer Measurement of the 2-Input OR Gate

At this point you have successfully wired a 2-input OR gate, provided it power and ground, provided logic inputs, and measured its proper operation by observing the inputs and output. Take a screenshot of the logic analyzer measurement for your records and save in a JPG format. **This image partially satisfies the requirements for deliverable #2.** To full satisfy deliverable #2 you will produce three images (INV, AND, OR).

3.1.5.3 AC Operation of 74HC Logic Gates

We now want to begin looking at the AC characteristics of a logic gate. The term “AC” refers to when we look at the signals as they transition between a logic 1 and 0. We do this type of measurement with an oscilloscope. Connect channel 1 of the AWG back to the input of the inverter. Connect the 1+ oscilloscope channel of the Analog Discovery to the input of the inverter and the 2+ channel to the output of the inverter. Connect the oscilloscope references 1- and 2- to the ground of your breadboard.

In Waveforms, click on the Welcome button to go back to the main screen. Click on the “Scope” button. Setup the trigger for the oscilloscope as follows:

- Mode: Auto
- Source: Channel 1
- Rising
- Level: 1.7 V

Press the “Run” button on the oscilloscope. You should now see the input and output signals for the inverter. Zoom in horizontally by setting the time base to 0.2 ms/div. Zoom vertically by setting the range of both channels to 1 V/div. On the left of the screen, you can drag/drop the triangles for each channel to shift them up or down. Position the waveforms so you can clearly see the input and output. You should see a measurement similar to [Figure 3.6](#).

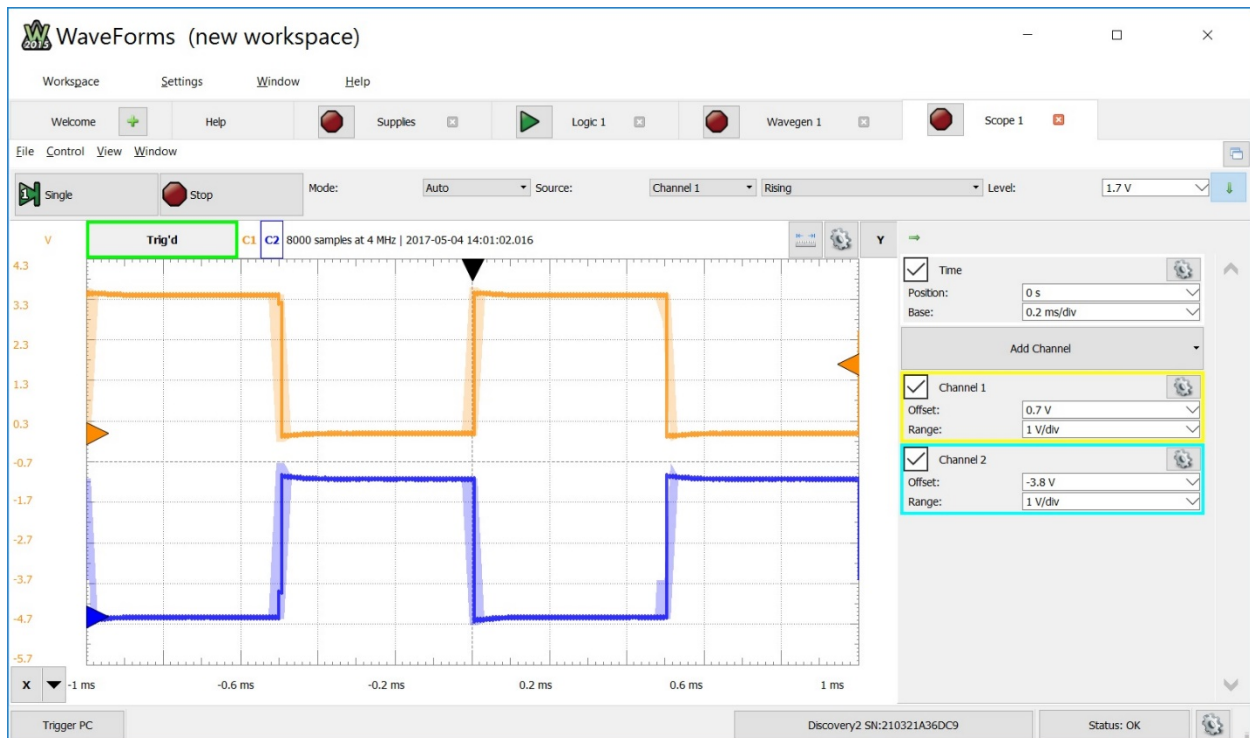


Figure 3.6
Oscilloscope Measurement of the Input and Output of the Inverter

This measurement shows the analog nature of the behavior of the inverter. Take a screenshot of the oscilloscope measurement for your records and save in a JPG format. **This image satisfies the requirements for deliverable #3.**

3.1.5.4 Observing the Impact of an Input Signal that Doesn't Meet the Minimum Specifications.

Now let's observe the impact when the input does not meet the minimum input specifications that you determined for the first deliverable of this lab. Let's keep the input centered at +1.7v but begin shrinking its amplitude. Keep the

oscilloscope measurement running while you reduce the amplitude of channel 1 of the AWG. Note that you can undock a tool window in Waveforms by clicking the icon in the upper right corner of any window.

As you reduce the input signal amplitude, you'll first notice that the output of the inverter remains a quality digital signal for quite a while. It is only once the amplitude gets down to ~ 0.3 V does the output begin to show signs of failure. You will begin to see the output pulses have different magnitudes and behave erratically. This can be seen in Figure 3.7. When the input amplitude is reduced further the output ultimately go to 0v.

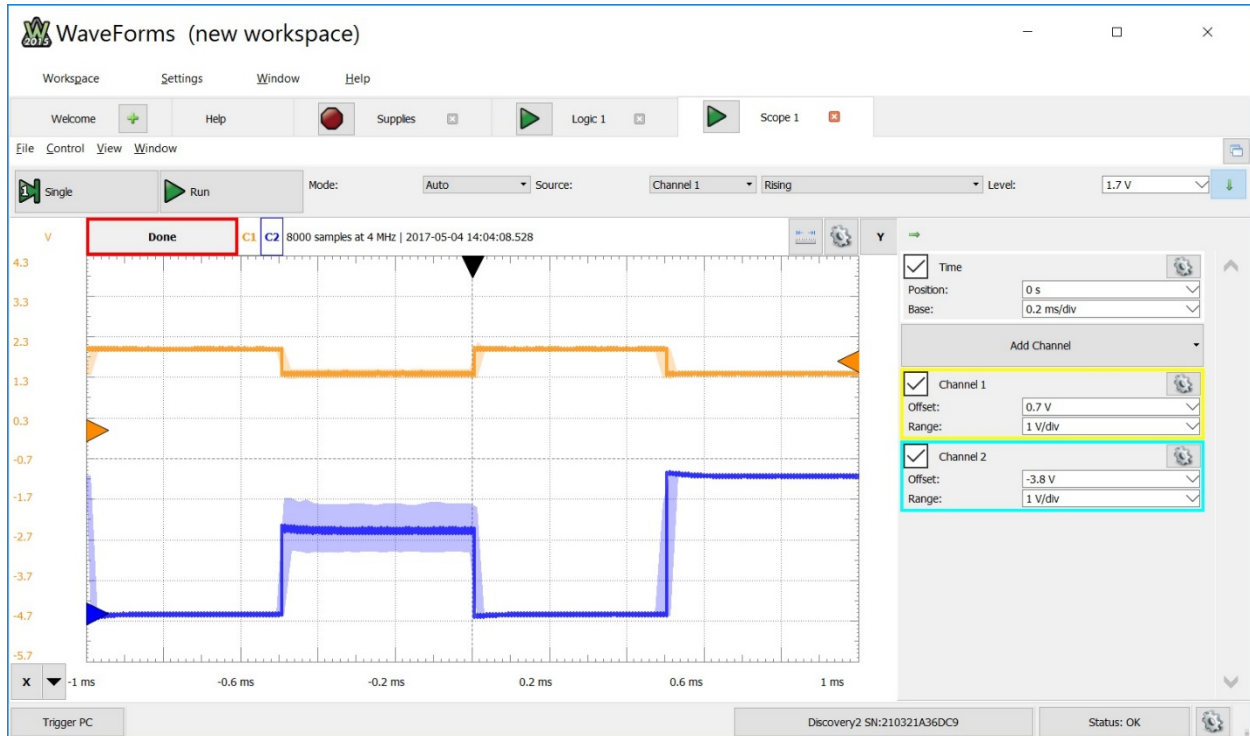


Figure 3.7
Impact of an Input Signal Not Meeting the Minimum Specifications

Set the AWG amplitude back to the point where the output is exhibiting erratic behavior. Take a screenshot of the measurement for your records and save in a JPG format. **This image satisfies the requirements for deliverable #4.**

3.1.5.5 Measuring the AC Characteristics of the Inverter

Finally, let's look at the propagation delay and transition time of the inverter. In the oscilloscope, configure the zoom so that the waveforms are larger by setting the vertical range to 0.5 V/div. Drag the waveforms vertically so that they are on top of each other and in the center of the screen. Zoom in horizontally by setting the time **Base = 0.05 us/div**. To get the trigger point in the center of the measurement by setting the time **Position = 0s**. You should see a measurement similar to Figure 3.8.

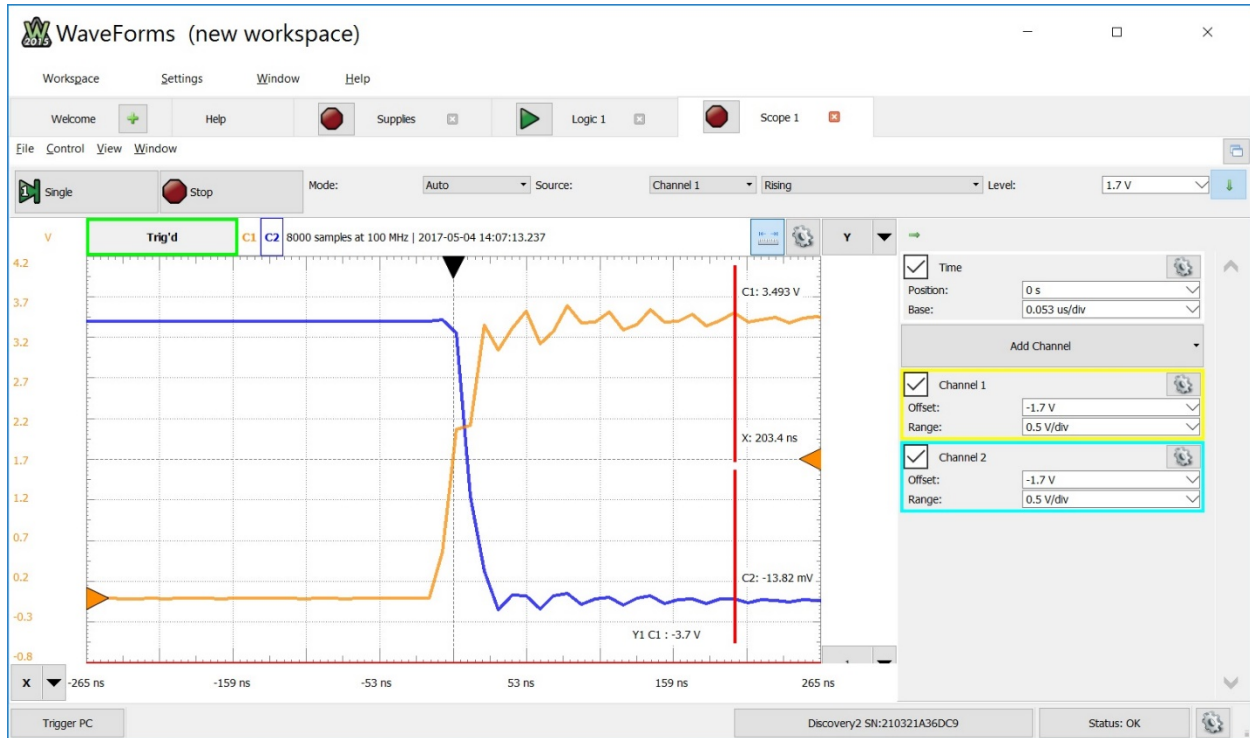


Figure 3.8
Measurement of the AC Characteristics of the Inverter

The oscilloscope tool has a feature called “HotTrack”. This provides a cursor that automatically measures time and voltage of the waveforms on the screen. To turn on HotTrack, click on the icon in the upper right corner of the measurement section. The icon looks like a ruler. Once on, you’ll see a cursor that you can drag left and right and view various time and voltage information about your measurement. Use HotTrack to determine the propagation delay (i.e., the time difference between when the input and output cross the 50% point of their transition) and the transition time (i.e., the time it takes the output to go from 10% to 90% of its logic swing). Note that HotTrack doesn’t automatically produces these results. You will need to manually move the cursor to the specific points on the waveform, record the time manually, and then perform a subtraction.

Take a screenshot of your oscilloscope measurement with HotTrack on for your records and save in a JPG format. **This image satisfies the requirements for deliverable #5.**

CONCEPT CHECK

Lab 3.1 After completing this lab exercise, can you:

- Determine the pinout, DC, and AC specification of a logic gate from its data sheet?
- Use a power supply, AWG, and logic analyzer to verify the DC operation of a basic gate?
- Measure the AC operation of a basic gate with an oscilloscope?
- Describe the impact of an input signal that doesn’t meet the minimum input specifications?
- Use an oscilloscope to measure the propagation delay and transition time of a logic gate?

Chapter 4: Combinational Logic Design

Lab 4.1: 3-Input Prime Number Detector using Canonical Forms

4.1.1 Objective

The objective of this lab is to gain experience with logic synthesis using canonical sum-of-products (SOP) and canonical product-of-sums (POS) forms. You will design a 3-input prime number detector using both forms and demonstrate the proper (and equivalent) operation. You will also gain experience with addressing fan-in constraints of real logic circuits. You will also build an LED driver circuit that will allow you to drive in different codes into your detector using switches and observe both the input and output logic values on LEDs.

4.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Implement an LED driver circuit that will be used to provide the inputs of your logic circuits.
- Design, breadboard, and test 3-input combinational logic circuits using a canonical SOP form?
- Design, breadboard, and test 3-input combinational logic circuits using a canonical POS form?
- Understand how fan-in constraints can be addressed in real-world circuits.

4.1.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 1x, 8-position SPST slider switch.
- 1x, resistor network, 9-position, 10-pin SIP, 10 k Ω , bussed.
- 1x, resistor network, 8-position, 16-pin DIP, 330 Ω , isolated.
- 5x, red LEDs, discrete.
- 2x, 74HC04 inverter ICs (1x used for LED driver, 1x used for SOP/POS circuits).
- 2x, 74HC4075 3-input OR ICs.
- 1x, 74HC32 2-input OR gate IC.
- 2x, 74HC11 3-input AND gate ICs.
- 1x, 74HC21 4-input AND gate IC.

4.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate the proper operation of an LED driver circuit (30% of exercise).
2. Provide your design steps to synthesize a canonical SOP logic circuit to implement a 3-input prime number detector (10% of exercise).
3. Demonstrate the proper operation of a 3-input prime number detector implemented with a canonical SOP form (25% of exercise).
4. Provide your design steps to synthesize a canonical POS logic circuit to implement a 3-input prime number detector (10% of exercise).
5. Demonstrate the proper operation of a 3-input prime number detector implemented with a canonical POS form (25% of exercise).

4.1.5 Lab Work & Demonstration

4.1.5.1 Implement an LED Driver Circuit

Breadboard an LED Circuit

In this exercise we need a way to create all possible input codes for a 3-input combinational logic circuit in order to verify proper operation of the design. We can drive the inputs using the 8-position DIP switch provided in the lab kit. We would also like to be able to observe the input codes on LEDs. We would also like to observe the output of our combinational logic circuit on an LED. In order to accomplish this, you are going to build the following LED driver circuit. This circuit allows up to 4 input codes to be created and displayed on LEDs. This circuit uses a dedicated inverter to drive the LEDs so that the inputs to the circuit-under-test are not loaded excessively. This circuit will be used on future labs so plan on leaving it on your breadboard. A schematic of the LED driver circuit is shown in Figure 4.1.

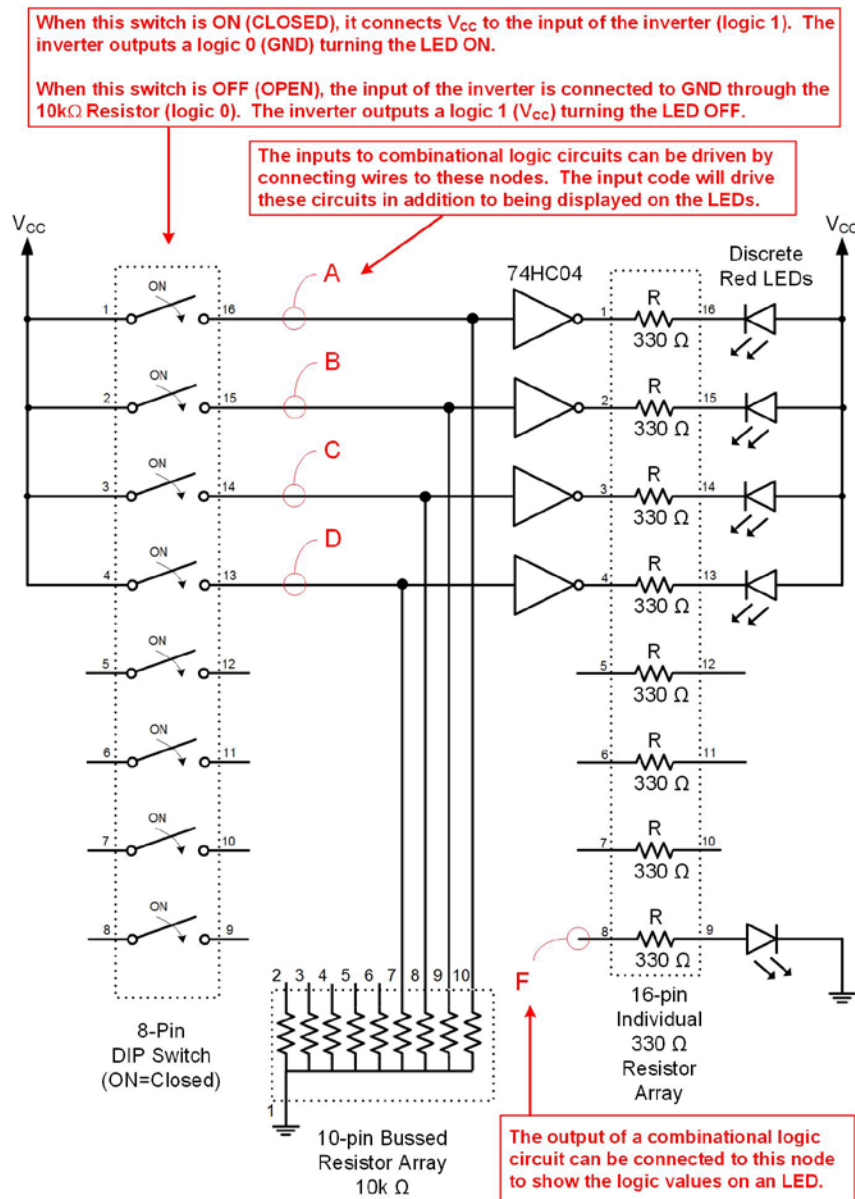


Figure 4.1
LED Driver Circuit Schematic

Figure 4.2 shows an example layout on your breadboard.

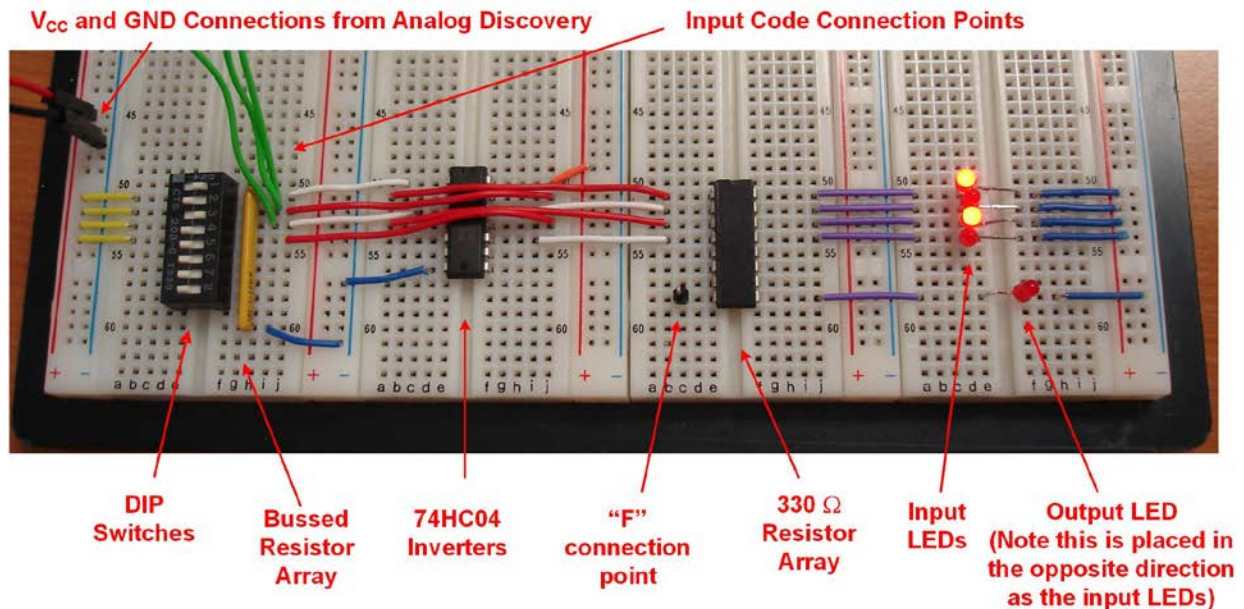


Figure 4.2
LED Driver Circuit Layout

Test your LED Driver Circuit

Test your LED driver circuit. You will need connect V_{CC} and GND from the Analog Discovery to your breadboard. You will then need to launch Waveforms and configure the power supply output to generate +3.4v for your breadboard. Once powered, you should be able to toggle DIP switches 1-4 and see the corresponding ON/OFF pattern on the LEDs. You won't be able to test the output LED until you connect the output of a logic circuit to it. This will be done in the next sections.

Take a short video (<5 s) showing the proper operation of your LED driver circuit. **This video satisfies the requirements for deliverable #1.**

4.1.5.2 Design a 3-Input Prime Number Detector using a Canonical SOP Form

You are now going to design a logic circuit that will assert its output when the decimal value of the 3-bit input is a prime number. A 3-bit input can represent numbers from 0_{10} to 7_{10} . Within this range, the values 2, 3, 5, and 7 are considered prime numbers. Notice that in a canonical SOP form, you will need a 3-input AND gate for each input row with an output of 1. That means you will need 4x, 3-input AND gates. These are provided in your lab kit. You will feed the corresponding outputs of the 4x AND gates into a 4-input OR operation; however, your lab kit only provides 2-input and 3-input OR gates. This means you have encountered a **fan-in constraint**. You need to figure out a way to overcome this issue by manipulating the circuit. Consider using the associative property on the OR operation. You are going to design the SOP prime number detector by hand. The steps in this design process are as follows:

- Create a 3-input truth table with the desired output for each input code.
- Create a minterm list from the truth table.
- Create a canonical SOP logic expression from the truth table / minterm list.
- Draw the logic diagram of the SOP logic expression.
- Map the logic operations into the available ICs in your parts kit.
- Address any fan-in issues you may have (HINT: You will have a fan-in issue when implementing the 4-input OR operation).
- Redraw the final logic diagram that will be implemented on your breadboard.

Create the Truth Table

In the space provided below, draw the truth table for the 3-input prime number detector.

Create the Minterm List

In the space provided below, enter the minterm list for the 3-input prime number detector.

Create the Canonical SOP Logic Expression

In the space provided below, enter the canonical SOP logic expression for the 3-input prime number detector.

Draw the Logic Diagram for the Canonical SOP Logic Expression

In the space provided below, draw the logic diagram for the canonical SOP logic expression for the 3-input prime number detector. Your circuit should contain three inverters to generate A' , B' , and C' . Your circuit should contain 4x, 3-input AND operations to generate the 4x minterms in this circuit. Your circuit should contain 1x, 4-input OR operation to generate the output of the circuit.

Map the Logic Diagram for your SOP Circuit into Available ICs in your Kit

At this point you are now ready to begin thinking about how to implement your logic diagram in real circuitry. Each of the gates in the above logic circuit will be implemented using real circuits from your parts kit. In this step you need to think about which ICs will implement the logic operations. You are essentially *mapping* the operations into real circuitry. This can be done by drawing rectangles around various operations that can be grouped into specific ICs.

First, you need 3x inversion operations. Note that your parts kit contains an HC04 IC, which has 6x inverters on it. This means you can implement all three inversion operations on 1x HC04. Draw a rectangle around your 3x inverters in your logic diagram and label the rectangle "HC04". This rectangle represents how you will implement the inverters on our breadboard.

The next step is to map the 4x, 3-input AND operations into ICs in your parts kit. Note that your parts kit contains a 3-input AND gate in the form of the HC11 IC. The HC11 contains 3x, 3-input AND operations. This means you will need 2x of the HC11 IC in order to implement all four of the AND operations in your circuit. Draw one rectangle around the upper three AND operations and another rectangle around the lower AND operation. Label both rectangles "HC11". These rectangles represent how you will implement the AND operations on our breadboard.

Finally, you need to map the 4-input OR operation into ICs in your parts kit. Note that you do NOT have a 4-input OR gate IC. The largest OR gate in your kit has 3-inputs. You have encountered a **fan-in issue**.

Address Any Fan-In Issues You've Encountered

Fan-in issues are addressed by manipulating the logic operation into one that is equivalent, but that uses gates that exist in your parts kit. In this situation, you can apply the associative property to the 4-input OR operation in order to manipulate it into an equivalent expression that uses only 2-input operations. Consider the following:

$$F = (m_2 + m_3 + m_5 + m_7) = (m_2 + m_3) + (m_5 + m_7)$$

Notice that the manipulated expression gives the same logic operation, but uses 3x, 2-input OR operations instead of 1x, 4-OR operation. The 2-input OR operations can be implemented using the HC32. Since the HC32 contains 4x, 2-input OR gates, you can implement the all 3 operations on a single HC32 IC.

Back in your logic diagram, add an additional drawing next to the 4-input OR gate that shows the new logic operation using 3x, 2-input OR gates. Draw a rectangle around the three OR operations and label it "HC32". This rectangle represents how you will implement the OR operations on our breadboard.

Record your SOP design process from above electronically. You should have all of your work on one page. If you printed the page with your design steps and manually wrote in your steps, you can either scan the page or take a photo of the page and save as a JPG. If you created the design electronically, take a screen shot of your design steps and save in JPG format. **This image satisfies the requirements for deliverable #2.**

4.1.5.3 Implement a 3-Input Prime Number Detector using a Canonical SOP Form

Breadboard your Canonical SOP Circuit for the Prime Number Detector

You are now going to breadboard your canonical SOP circuit. You will drive in the 3-bit input codes using your LED driver circuit. You will display your detector's final output on the "F" LED of the driver circuit. When building your circuit it is useful to label your wires to avoid confusion. Also, it is good practice to test each interim computation as you go. For example, once you wire up the first minterm (m_2), test that it only asserts for the appropriate input code before moving onto bread boarding the next minterm. This will avoid the situation where you wire the entire circuit and then discover that it doesn't work and you have to start debugging back at the beginning. You can test the interim operations by wiring the result to the F LED of the driver circuit. Position your SOP circuit so that there is room for the POS circuit in the next section. Consider the breadboard layout shown in [Figure 4.3](#). NOTE: Don't try to simply copy the wiring in this figure. It is provided for reference on how to layout your breadboard. You should wire your own breadboard according to your own logic diagram.

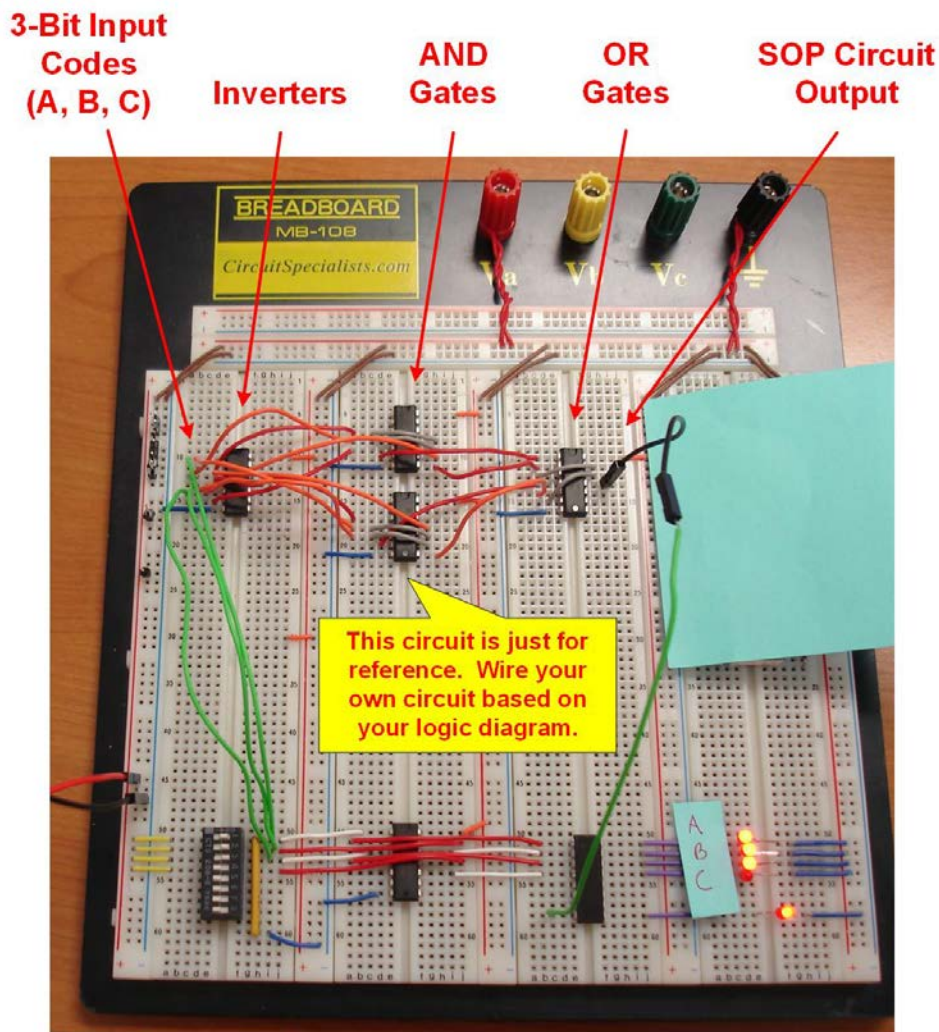


Figure 4.3
Prime Number Detector SOP Breadboard Layout

Test your Canonical SOP Prime Number Detector Circuit

Take a short video (<5 s) showing the proper operation of your canonical SOP prime number detector. You should cycle through each of the 8 possible input codes and show that the F LED only asserts for prime numbers. **This video satisfies the requirements for deliverable #3.**

4.1.5.4 Design a 3-Input Prime Number Detector using a Canonical POS Form

You are now going to design a functionally equivalent circuit to the SOP you just created, but now using a canonical POS form.

Create the Truth Table

In the space provided below, redraw the truth table for the 3-input prime number detector. This is the same table from the SOP design, you are just redrawing it here so you have it readily available.

Create the Maxterm List

In the space provided below, enter the Maxterm list for the 3-input prime number detector.

Create the Canonical POS Logic Expression

In the space provided below, enter the canonical POS logic expression for the 3-input prime number detector.

Draw the Logic Diagram for the Canonical POS Logic Expression

In the space provided below, draw the logic diagram for the canonical POS logic expression for the 3-input prime number detector. Your circuit should contain three inverters to generate A' , B' , and C' . Your circuit should contain 4x, 3-input OR operations to generate the 4x maxterms in this circuit. Your circuit should contain 1x, 4-input AND operation to generate the output of the circuit.

Map the Logic Diagram for your POS Circuit into Available ICs in your Kit

Now map the logic operations in your diagram into the parts that are available in your kit. For your inverters, you can reuse the inverter IC from your SOP circuit. You will need to use 2x, HC4075 ICs to implement the 4x, 3-input OR operations. For your final 4-input AND operation, your lab kit contains a 4-input AND IC (HC21) so you don't have any fan-in issues in your POS circuit. As before, draw rectangles around the logic operations to represent the ICs that will be used to implement the gates. Label each rectangle with the IC part number.

Record your POS design process from above electronically. You should have all of your work on one page. If you printed the page with your design steps and manually wrote in your steps, you can either scan the page or take a photo of the page and save as a JPG. If you created the design electronically, take a screen shot of your design steps and save in JPG format. **This image satisfies the requirements for deliverable #4.**

4.1.5.5 Implement a 3-Input Prime Number Detector using a Canonical POS Form

Breadboard your Canonical POS Circuit for the Prime Number Detector

Now breadboard your canonical POS circuit. You can reuse the HC04 inverter IC from the prior section. Note that you can't drive the F LED with two signals. You will need to disconnect your SOP output and instead connect the POS output to F.

Test your Canonical POS Prime Number Detector Circuit

Take a short video (<5 s) showing the proper operation of your canonical POS prime number detector. You should cycle through each of the 8 possible input codes and show that the F LED only asserts for prime numbers. **This video satisfies the requirements for deliverable #5.**

CONCEPT CHECK

Lab 4.1 After completing this lab exercise, can you:

- Describe the operation of the LED driver circuit that produces input codes to your logic circuit while simultaneously displays the input codes and output code on LEDs?
- Design & implement a canonical SOP circuit from a word description?
- Design & implement a canonical POS circuit from a word description?
- Explain what a fan-in issue is and how to address it?

Lab 4.2: 3-Input Prime Number Detector using Minimized Forms

4.2.1 Objective

The objective of this lab is to gain experience with logic synthesis using minimized sum-of-products (SOP) and minimized product-of-sums (POS) topologies. You will design a 3-input prime number detector using both forms and demonstrate their proper (and equivalent) operation. You will also gain experience with a circuit used to drive higher current loads than a basic gate can provide (i.e., a buzzer).

4.2.2 Learning Outcomes

After completing this lab, you will be able to:

- Design, breadboard, and test combinational logic circuits in both minimized SOP and POS forms.
- Implement a circuit to drive a buzzer with a logic signal as its control input.

4.2.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- LED driver circuit from prior lab (8-position slider switch, 10 k Ω resistor network, 330 k Ω resistor network, 5x red LEDs, 1x 74HC04 inverter IC).
- 1x, 74HC04 inverter IC.
- 2x, 74HC08 2-input AND ICs.
- 2x 74HC32 2-input OR gate ICs.
- 1x, Magnetic Buzzer.
- 1x, NPN Transistor, 2N3904.
- 1x, Diode, 1N4002.
- 1x, 10 k Ω axial resistor.

4.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Provide your design steps to synthesize a minimized SOP logic circuit (10% of exercise).
2. Demonstrate the proper operation of a 3-input prime number detector implemented with a minimized SOP form (30% of exercise).
3. Provide your design steps to synthesize a minimized POS logic circuit (10% of exercise).
4. Demonstrate the proper operation of a 3-input prime number detector implemented with a minimized POS form (30% of exercise).
5. Demonstrate the proper operation of a buzzer driving circuit (20% of exercise).

4.2.5 Lab Work & Demonstration

4.2.5.1 Design a 3-Input Prime Number Detector using a Minimized SOP Form

You are going to design a circuit that will assert when the decimal value of the 3-bit input is a prime number. A 3-bit input can represent numbers from 0₁₀ to 7₁₀. Within this range, the values 2, 3, 5, and 7 are considered prime numbers. You will do this using a minimized SOP approach. You will derive the minimized SOP logic expression using a K-map. The steps in this design process are as follows:

- Create a 3-input truth table with the desired output for each input code.
- Use a K-map to derive a minimized SOP logic expression.
- Draw the logic diagram of the SOP logic expression.
- Map the logic operations into the available ICs in your parts kit.

Create the Truth Table

In the space provided below, draw the truth table for the 3-input prime number detector.

Derive the Minimized SOP Logic Expression using a K-map

In the space provided below, derive the minimized SOP logic expression using a K-map for the 3-input prime number detector.

Draw the Logic Diagram for the Minimized SOP Logic Expression

In the space provided below, draw the logic diagram for the minimized SOP logic expression for the 3-input prime number detector.

Map the Logic Diagram for your SOP Circuit into Available ICs in your Parts Kit

In the above logic diagram, draw rectangles around the logic operations that can be implemented within a single logic IC from your parts kit. Write the part number next to the rectangle.

Record your SOP design process from above electronically. You should have all of your work on one page. If you printed the page with your design steps and manually wrote in your steps, you can either scan the page or take a photo of the page and save as a JPG. If you created the design electronically, take a screen shot of your design steps and save in JPG format. **This image satisfies the requirements for deliverable #1.**

4.2.5.2 Implement a 3-Input Prime Number Detector using a Minimized SOP Form

Breadboard your Minimized SOP Circuit for the Prime Number Detector

You are now going to breadboard your minimized SOP circuit. You will drive in the 3-bit input codes using your LED driver circuit. You will display your detector's final output on the "F" LED of the driver circuit.

Test your Minimized SOP Prime Number Detector Circuit

Provide power to your breadboard using the power supply from the Analog Discovery configured to output +3.4v. Take a short video (<5 s) showing the proper operation of your minimized SOP prime number detector. You should cycle through each of the 8 possible input codes and show that the F LED only asserts for prime numbers. **This video satisfies the requirements for deliverable #2.**

4.2.5.3 Design a 3-Input Prime Number Detector using a Minimized POS Form

You are now going to design a circuit that will implement the 3-input prime number detector, but this time using a minimized POS approach. You will derive the minimized POS logic expression using a K-map. The steps in this design process are as follows:

- Create a 3-input truth table with the desired output for each input code.
- Use a K-map to derive a minimized POS logic expression.
- Draw the logic diagram of the POS logic expression.
- Map the logic operations into the available ICs in your parts kit.

Create the Truth Table

In the space provided below, draw the truth table for the 3-input prime number detector. This is the same table as above, it is just being redrawn here so it is readily available.

Derive the Minimized POS Logic Expression using a K-map

In the space provided below, derive the minimized POS logic expression using a K-map for the 3-input prime number detector.

[Draw the Logic Diagram for the Minimized POS Logic Expression](#)

In the space provided below, draw the logic diagram for the minimized POS logic expression for the 3-input prime number detector.

[Map the Logic Diagram for your POS Circuit into Available ICs in your Parts Kit](#)

In the above logic diagram, draw rectangles around the logic operations that can be implemented within a single logic IC from your parts kit. Write the part number next to the rectangle.

Record your POS design process from above electronically. You should have all of your work on one page. If you printed the page with your design steps and manually wrote in your steps, you can either scan the page or take a photo of the page and save as a JPG. If you created the design electronically, take a screen shot of your design steps and save in JPG format. **This image satisfies the requirements for deliverable #3.**

[4.2.5.4 Implement a 3-Input Prime Number Detector using a Minimized POS Form](#)

[Breadboard your Minimized POS Circuit for the Prime Number Detector](#)

You are now going to breadboard your minimized POS circuit. You will drive in the 3-bit input codes using your LED driver circuit. You will display your detector's final output on the "F" LED of the driver circuit.

[Test your Minimized POS Prime Number Detector Circuit](#)

Take a short video (<5 s) showing the proper operation of your minimized POS prime number detector. You should cycle through each of the 8 possible input codes and show that the F LED only asserts for prime numbers. **This video satisfies the requirements for deliverable #4.**

[4.2.5.5 A Buzzer Driving Circuit](#)

[Breadboard an Interfacing Circuit for a Magnetic Buzzer](#)

There are often times when a digital circuit needs to interface with a device that requires more current than a logic gate output can provide. One example of this is a magnetic buzzer. The buzzer in your parts kit will make sound when V_{cc} is provided across its terminals; however, it will also draw 35 mA when it is on. This is above what our 74HC logic family can provide. In order to interface our logic signals to the buzzer, we will use the circuit in [Figure 4.4](#).

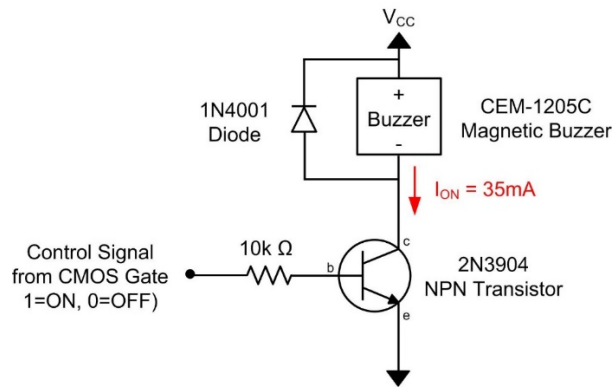


Figure 4.4
Schematic of Interfacing Circuit for a Magnetic Buzzer

The 2N3904 is an NPN transistor. This transistor can be thought of as a switch with a control signal. When we provide a high voltage to its control input (the BASE), it will close the switch, allowing current to flow between the other two terminals (the COLLECTOR & EMITTER terminals). The control signal takes very little current so we can drive it directly from a 74HC gate. The advantage of using this transistor is that the current between the collector and emitter terminals can be very high (200mA max). This allows us to use a 74HC gate to drive a device that pulls more current than the 74HC can provide. The diode in this circuit is used to sink the remaining current in the buzzer when it is turned off. Since the buzzer is magnetic, it acts like an inductor. When we turn it off, the current can't stop flowing instantaneously. The diode provides a path for the current to flow when it switches off instead of forcing it into the NPN transistor, which will cause damage. The 10kohm resistor is used to limit the current that flows into the NPN transistor. Keep the protective tape on the buzzer or it will be LOUD!

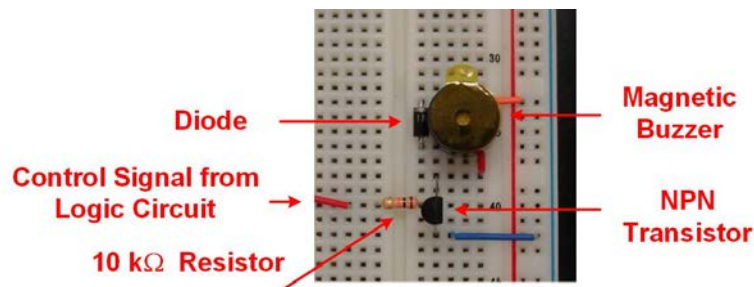


Figure 4.5
Layout of Interfacing Circuit for a Magnetic Buzzer

Test your Buzzer Circuit

Connect the output of one of your prime number detector circuits to both the F LED and the input to the buzzer. Whenever the output of the circuit is asserted, you should hear the buzzer sound. Take a short video (<5 s) showing the proper operation of your interfacing circuit for the buzzer. You should cycle through a few of the input codes that produce a logic 1 on the output to turn the buzzer on. **This video satisfies the requirements for deliverable #5.**

CONCEPT CHECK

Lab 4.2 After completing this lab exercise, can you:

- Design & implement a minimized SOP circuit from a word description?
- Design & implement a minimized POS circuit from a word description?
- Describe the operation of an interfacing circuit for a magnetic buzzer circuit?

Lab 4.3: 7-Segment Display Decoder (Discrete)

4.3.1 Objective

The objective of this lab is to gain additional experience with logic synthesis using minimized SOP/POS forms and logic manipulation using DeMorgan's Theorem. You will be designing a 3-input, 7-segment display decoder and demonstrating its proper operation.

4.3.2 Learning Outcomes

After completing this lab, you will be able to:

- Design, breadboard, and test a 3-input 7-segment display decoder.
- Use DeMorgan's theorem to manipulate logic expressions to use NAND/NOR gates.

4.3.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- LED driver circuit from prior lab (8-position slider switch, 10 k Ω resistor network, 330 k Ω resistor network, 5x red LEDs, 1x 74HC04 inverter IC).
- 7-Segment character display.
- 7x, 150 Ω axial resistors.
- Nearly all of your discrete basic gates in your parts kits (AND, OR, INV, NAND, and NOR).

4.3.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Provide your design steps to synthesize the minimized SOP/POS logic expressions for the 7-segment decoder circuitry (10% of exercise).
2. Provide your steps to manipulate at least one logic expression using DeMorgan's Theorem to implement the expression using only NAND or NOR gates (10% of exercise).
3. Demonstrate the proper operation of your 7-segment decoder circuit implemented on your breadboard (80% of exercise).

4.3.5 Lab Work & Demonstration

4.3.5.1 Design the Logic for the 7-Segment Decoder

Complete the Truth Table for the 7-Segment Decoder

You are going to design a 7-segment display decoder. A 7-segment display contains 7 separate LEDs that can be turned on/off to create symbols that represent decimal numbers. They are used in many applications when decimal numbers need to be displayed in a simple manner. One of the most common applications that you may be familiar with is a digital clock. Your lab kit contains a 7-segment display that you can examine as you read through this exercise. You are going to build a decoder that takes in a 3-bit binary number and displays the equivalent decimal symbol on the 7-segment display (e.g., 0, 1, 2, 3, 4, 5, 6 or 7). Note that your decoder will consist of 7 different logic circuits, one for each of the LEDs within the 7-segment display.

The 7-segment display in your lab kit is manufactured in a *common-cathode* configuration. This part contains seven separate input pins for each LED segment and then a single cathode for all LEDs. This minimizes the number of pins on the part. The common-cathode configuration allows us to design our decoder using positive logic (e.g., a logic 1 corresponds to the LED being ON). We can connect the single cathode of the display to ground. [Figure 4.6](#) shows how the display is wired and an example of how it is used. The example shows a 3-bit code that is displayed using our LED driver from prior labs and the corresponding decimal symbol on the 7-segment display.

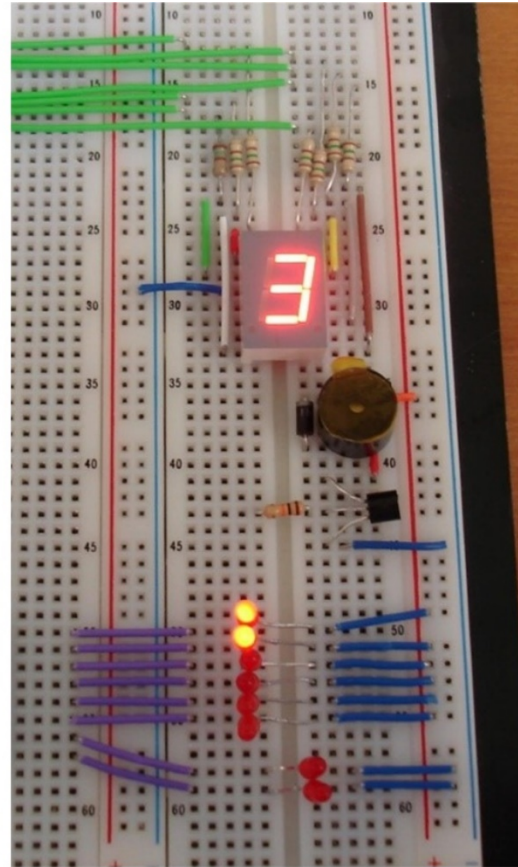
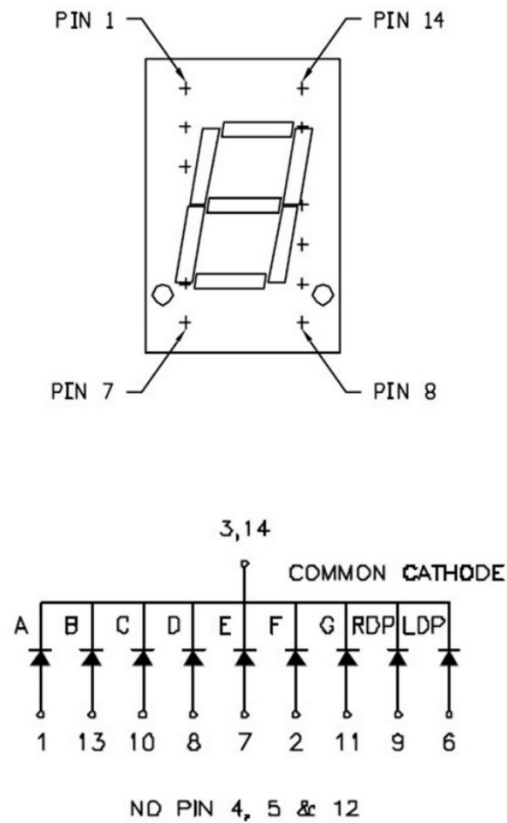
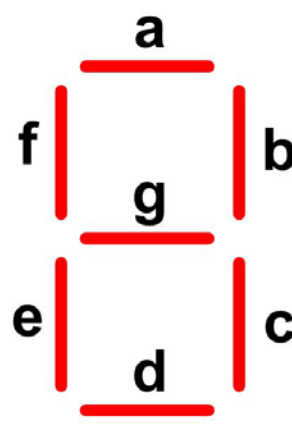


Figure 4.6
7-Segment Display Pinout, Schematic, and Use-Model

The first step in designing the decoder logic is to build a truth table for each of the seven circuits that will drive the display. [Figure 4.7](#) illustrates how to derive the truth tables for the decoder. Each of the seven LEDs are labeled a, b, c, d, e, f and g. For each input code, you will enter a 1 in the row of the following table if the individual LED needs to be ON to display the corresponding decimal character. For example, the input code $A=0, B=0, C=0$ corresponds to a display character of "0". To create this character, you turn on all LEDs within the display except for "g". In the table you would go to the **row** corresponding to input codes $A=0, B=0, C=0$ and enter $F_a=1, F_b=1, F_c=1, F_d=1, F_e=1, F_{de}=1,$ and $F_g=0$. Once you have completed the table for all input codes, logic expressions can be derived for each **column**. The column values for F_a have been provided to get you started. The first part of deliverable #1 is to complete the truth table in [Figure 4.7](#).

LED Labels



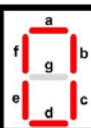
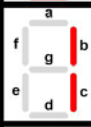
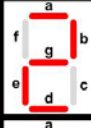
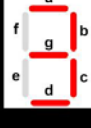
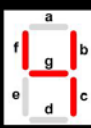
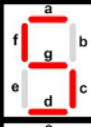
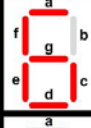
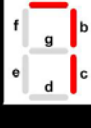
A	B	C		F _a	F _b	F _c	F _d	F _e	F _f	F _g
0	0	0		1						
0	0	1		0						
0	1	0		1						
0	1	1		1						
1	0	0		0						
1	0	1		1						
1	1	0		1						
1	1	1		1						

Figure 4.7
Truth Table for 7-Segment Display Decoder

Derive the Minimized Logic Expressions for the 7-Segment Decoder

Now you are going to create a logic expression for each of the segments in the display (i.e., F_a, F_b, F_c, ..., F_g). You will do this by creating 3-input K-maps for each function and then deriving a minimal logic expression. You will need **seven k-maps**. The input codes for each K-map are A, B, and C. The output values within each K-map come from the columns associated with each segment. In the space below, derive the minimized logic expressions. Note that depending on the K-map, a POS form might be easier to implement than a SOP.

- **Logic Expression for F_a :**

- **Logic Expression for F_b :**

- **Logic Expression for F_c :**

- **Logic Expression for F_d :**

- **Logic Expression for F_e :**

- **Logic Expression for F_f :**

- **Logic Expression for F_g :**

Record your table and logic expressions from above electronically. This should consist of two pages, one for the truth table and one with your seven logic expressions. If you printed the pages and completed them manually, you can either scan the pages or take photos of the pages and save as a JPG. If you completed the design electronically, take a screen shot of your design steps and save in JPG format. **These image satisfies the requirements for deliverable #1.**

4.3.5.2 Use DeMorgan's Theorem to Manipulate at Least One of the Logic Expressions

Now you are going to begin mapping the logic expressions into the logic gates in your parts kit. As you go through this process, you will quickly realize that you do not have enough AND gates and OR gates to implement all seven logic expressions in their derived SOP/POS forms. You will need to manipulate some (or all) of your logic expressions using DeMorgan's Theorem into forms that use NAND gates and NOR gates. Your lab kits *does* contain a variety of n-input NAND/NOR gates that will allow you to complete the implementation. For this deliverable, show your use of DeMorgan's to manipulate at least one of your logic expressions from above. Record your logic manipulation electronically. **This image satisfies the requirements for deliverable #2.**

4.3.5.3 Implement your 7-Segment Decoder System

Breadboard your 7-Segment Decoder Circuit

Now breadboard your 7 logic circuits and your 7-segment display. The 3-bit input will come from the slider switches in your LED driver circuit. When you are finished, you will be able to cycle through each of the 8 possible input codes on the slider switches and see the corresponding decimal symbol on the character display. When breadboarding your character display, a resistor is needed in series with each LED. You will need to place 150Ω resistors in series with each input pin of the display in order to set the forward current to its recommended value. You will leave your 7-segment display on your breadboard for the next few lab exercises, so consider an organized approach to the wiring as shown in [Figure 4.6](#).

This is a large circuit to implement. Consider bread boarding and testing one circuit at a time. For example, implement the circuit for segment F_a . Then cycle through the inputs codes 000 to 111 and make sure that the LED asserts correctly. This will allow you to ensure each segment is working incrementally before moving to the next circuit. Another advantage of building the decoder one circuit at a time is that you can reuse terms that have been verified. For example, once you know that A' , B' , and C' are working, you can use them in other circuits without putting down more inverters. The same goes with common product or sum terms that occur in multiple circuits. Another tip is to label the wires of interim terms so that you can easily find them as you build subsequent circuits.

Test Your 7-Segment Decoder Circuit

Provide power to your breadboard using the power supply from the Analog Discovery configured to output +3.4v. Take a short video (<5 s) showing the proper operation of your 7-segment decoder. You should cycle through each of the 8 possible input codes and the corresponding decimal symbol should light up on the 7-segment display. **This video satisfies the requirements for deliverable #3.**

CONCEPT CHECK

Lab 4.3 After completing this lab exercise, can you:

- Describe the theory of operation of a 7-segment decoder?
- Derive minimized logic expressions for a 7-segment decoder using either SOP or POS forms?
- Use DeMorgan's Theorem to manipulate logic expressions into forms that use only NAND or NOR gates?
- Implement a 7-segment decoder system using discrete logic gates?

Chapter 5: VHDL (part 1)

Lab 5.1: 4-Input Prime Number Detector in VHDL

5.1.1 Objective

The objective of this lab is to gain experience designing combinational logic using a hardware description language and implementing the circuitry using the modern digital design flow. In this lab you will design a 4-input prime number detector in VHDL and then use the *Quartus* toolchain to synthesize and implement your circuit on a common programmable logic device, the field programmable gate array (FPGA). You will also gain experience on how to interface an FPGA board to your breadboard.

5.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Use the Quartus toolchain to synthesize, technology map, place/route and implement a VHDL model on an FPGA.
- Interface an external FPGA board to your breadboard.

5.1.3 Parts Needed

- Breadboard + wires.
- LED driver circuit from prior lab (8-position slider switch, 10 k Ω resistor network, 330 k Ω resistor network, 5x red LEDs, 1x 74HC04 inverter IC).
- Buzzer circuit from prior lab (magnetic buzzer, 2N3904 NPN transistor, 1N4002 diode, 10 k Ω axial resistor).
- DE0-CV FPGA board.
- 3x female-to-female jumper wires.

5.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a VHDL design on an FPGA that drives the DE0-CV slider switches to the DE0-CV LEDs (50% of exercise).
2. Demonstrate a 4-input prime number detector in VHDL on an FPGA (40% of exercise).
3. Provide your top.vhd design file (10% of exercise).

5.1.5 Lab Work & Demonstration

You are going to design another prime number detector, but this time the detector will take in numbers between 0_{10} and 15_{10} . This will require your detector to have four binary inputs (0000_2 to 1111_2). The detector will be designed in VHDL and implemented on an Altera Cyclone V FPGA (part no. 5CEBA4F23C7N), which resides on the DE0-CV board. This board is provided as part of your lab kit. Your VHDL will be synthesized using the Quartus toolchain and downloaded to the FPGA board for testing. The inputs of the detector will come from the 4x slider switches on the DE0-CV board. The output of the detector will be driven to a pin on the DE0-CV board and then a jumper wire will be used to connect the output to your breadboard, which will in turn drive your buzzer and an LED. Your breadboard will receive power (+3.4v) from the DE0-CV board for this lab. Note that we will not be using the Analog Discovery for this lab. Again, we will use the definition of a prime number as: "A Prime Number can be divided evenly only by 1 or itself and it must be a whole number greater than 1." By this definition, the prime numbers between 0_{10} and 15_{10} are: 2, 3, 5, 7, 11 and 13. As is good practice in all large designs, we will get incremental pieces of the design working before moving to the next part. First, we will create a Quartus project with VHDL to drive 4x slider switch inputs to 4x red LEDs on the DE0-CV board. Second, we will connect the DE0-CV board to our breadboard. Finally, we will create the VHDL to create a 4-input prime number detector with the inputs displayed on the red LEDs on the DE0-CV board and the

output displayed on an LED on your breadboard. We will also drive the buzzer on the breadboard to provide audible feedback. Figure 5.1 shows the block diagram for this lab exercise.

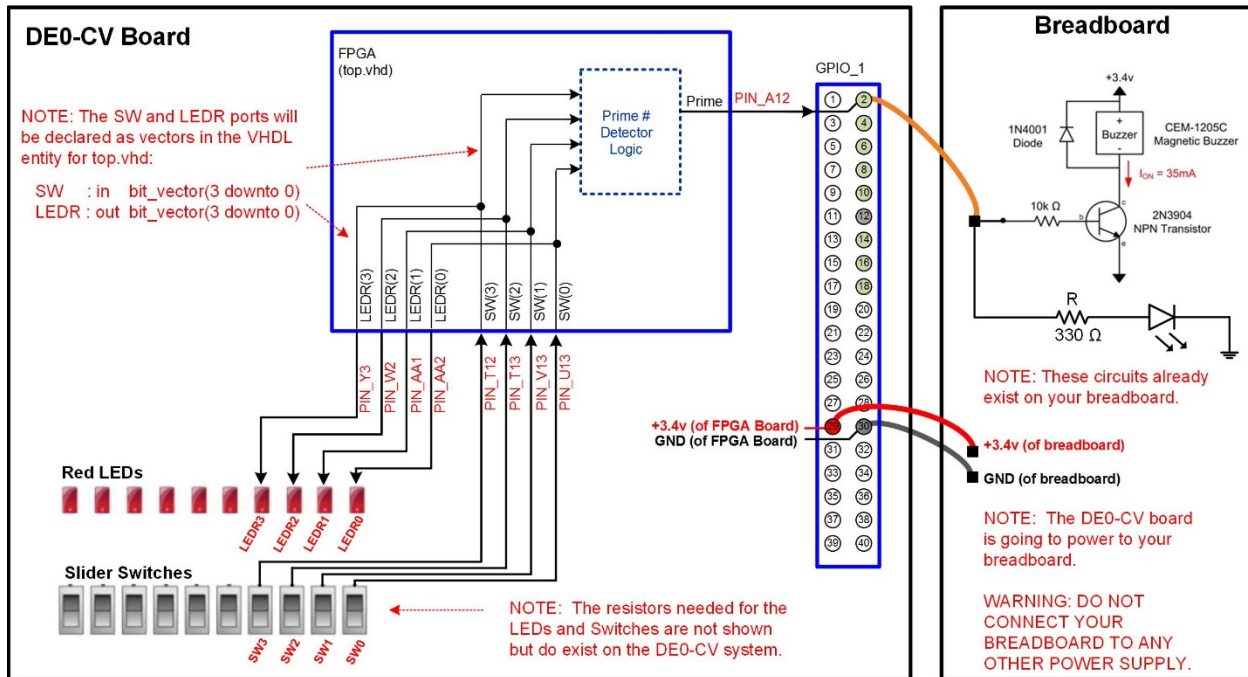


Figure 5.1
Block Diagram of the 4-Input Prime Number Detector System

Figure 5.2 shows the final implementation of the prime number detector. Note that the DE0-CV board is connected to the breadboard using jumper wires. The jumper wires are used to connect power (+3.4v) and ground between the boards and routes the output of the detector (Prime) to the breadboard LED and buzzer. **Don't connect the DE0-CV to your bread board yet.** You will first get the DE0-CV working and then make the connection in a later step.

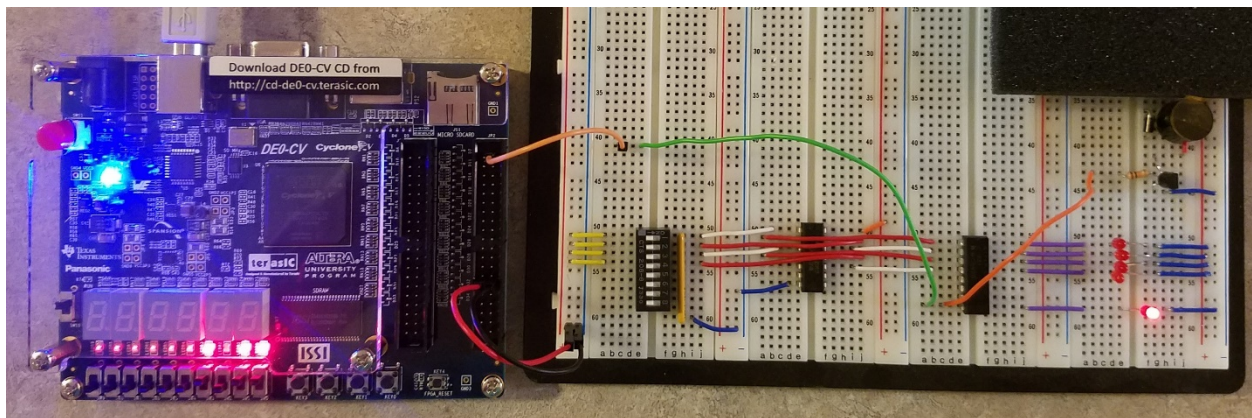


Figure 5.2
Picture of the 4-Input Prime Number Detector System on the DE0-CV Board + Breadboard Interface

5.1.5.1 Implement a VHDL Design that will Drive the Switches to the LEDs on the DE0-CV Board

We first want to create a simple design that will drive the lower four slider switches on the DE0-CV board (labeled SW3, SW2, SW1 and SW0) to the red LEDs on the DE0-CV board (labeled LEDR3, LEDR2, LEDR1 and LEDR0). In this portion of the lab exercise we will create a new Quartus project, create the VHDL to accomplish driving the switch

values to the LEDs, assign the pins of the FPGA to the I/O we are using, and synthesize the design. Once we download our design to the FPGA we will be able to test the first part of this exercise. This will ensure that the Quartus software is installed correctly, including the drivers for the DE0-CV board, and that the pins are assigned correctly before moving onto the next parts. You should not connect the DE0-CV board to your breadboard in this step.

[Install the Quartus Lite Software \(if not already installed on your computer\)](#)

The Quartus Lite software can be downloaded for free from www.altera.com. This will take you to an *intel* website, which purchased Altera Inc. in 2016. Once on this page, click on “Support” and then “Downloads”. You will need to create a free account. There you’ll be given the option of downloading various versions of Quartus. You want to download the “Lite” version. Click on the “Download” icon next to the Lite version. In the next screen you’ll be given options on what all should be downloaded. You want to select three items:

- Select edition: Lite
- Select release: “latest release, i.e., the highest number”
- Operating System: windows (assuming you are on windows)
Download Method: Direct download

Then click on the “Individual Files” tab. Select the following files to download:

- Quartus Prime (includes Nios II EDS)
- ModelSim-Intel FPGA Edition (includes Starter Edition)
- Cyclone V device support

Then click “Download Selected Files”. It will ask you where to download the files. Choose your Desktop and select “OK”. It will then proceed to download three files. The files are large so it will take a while. Once they are downloaded to your desktop, double click on the Quartus install file (named something similar to “QuartusLiteSetup-17.0.0.595-windows.exe”). This will automatically install the ModelSim software and the Cyclone V drivers that you downloaded as long as they are in the same location (i.e., the Desktop). Accept the defaults for all options in the install.

When complete, it will ask if you want to install desktop icons, launch Quartus, and/or launch the driver install tool. Select the option to launch the driver install tool. Follow the instructions to install the drivers for the DE0-CV board. When you plug in the DE0-CV board in the next few steps, the driver installation will complete.

[Create a Folder for to Hold All of your VHDL Projects \(if not already created\)](#)

For each Quartus project, you will manually create a folder and then direct Quartus to put all design files into that directory. We want to first create a main folder that will hold all of the project folders that will be created throughout this manual to help us stay organized. We will create a folder on the Desktop called “Logic_Lab”. Right click on the Desktop of your computer and select “New → Folder”. Give it the name “Logic_Lab”.

[Create a Folder for this Exercise](#)

Now we need to create the folder that will contain all of the Quartus files for this project. We want to name this project something descriptive. Browse to your Logic_Lab folder that you created in the prior section and manually create a folder called “Lab_05p1_PrimeNumDet”.

[Launch Quartus](#)

On a windows 10 or equivalent machine, the Quartus application can be launched at: Start → Intel FPGA *version* Lite Edition → Quartus (Quartus Prime *version*). [Figure 5.3](#) shows the Quartus startup window that will appear.

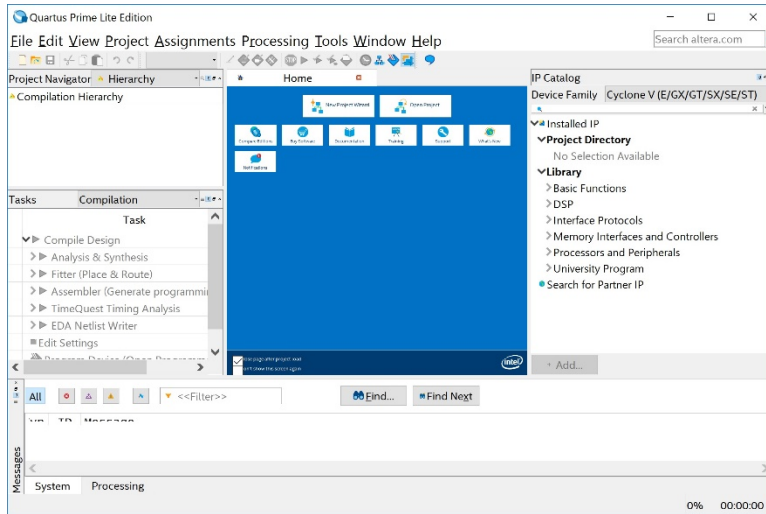


Figure 5.3
Quartus Startup Window

[Create a New Project using the “New Project Wizard”](#)

Click on the “New Project Wizard” button in the *Home* pane of the Quartus window. The *Introduction* window shown in Figure 5.4 will appear.

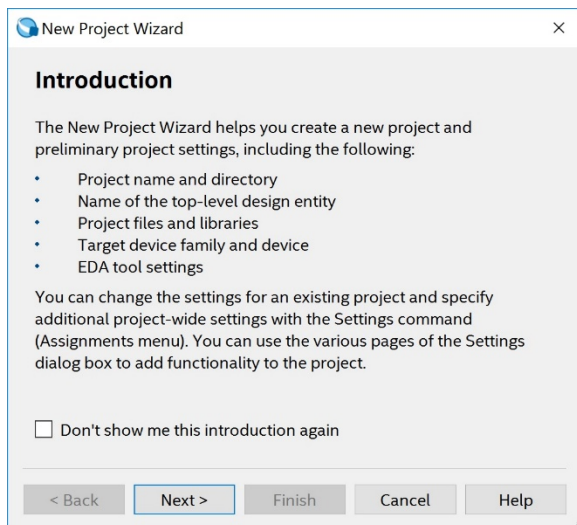


Figure 5.4
Quartus New Project Wizard (Introduction)

Click on “Next” to go the next window. If you don’t want the introduction window to show up next time you run New Project Window, you can check the box. The *Directory, Name, Top-Level Entity* window shown in Figure 5.5 will appear.

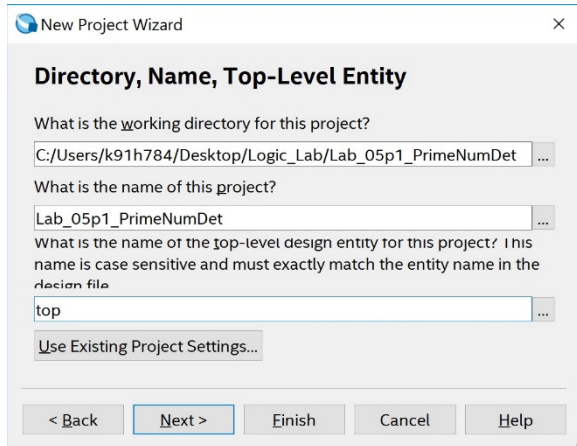


Figure 5.5
Quartus New Project Wizard (Directory, Name, Top-Level Entity)

For the working directory, click on the “...” button and browse to the “Logic_Lab/Lab_05p1_PrimeNumDet” folder you created on your desktop. This project folder will contain all of the Quartus design files.

For the name of the project, enter “Lab_05p1_PrimeNumDet”. Note that as you type this name, Quartus automatically enters the same name for the top-level entity. **You do not want to use this name as your top-level entity.**

For the top-level design entity, enter “top”. The term *top* is the standard naming convention for the VHDL file that is at the highest level of hierarchy in the system. At the top level, the ports of the entity will be the physical pins of the device. For this exercise, we will only have one top-level file. We will create this file (top.vhd) in a later step.

Click on the “Next” button. The *Project Type* window shown in [Figure 5.6](#) will appear.

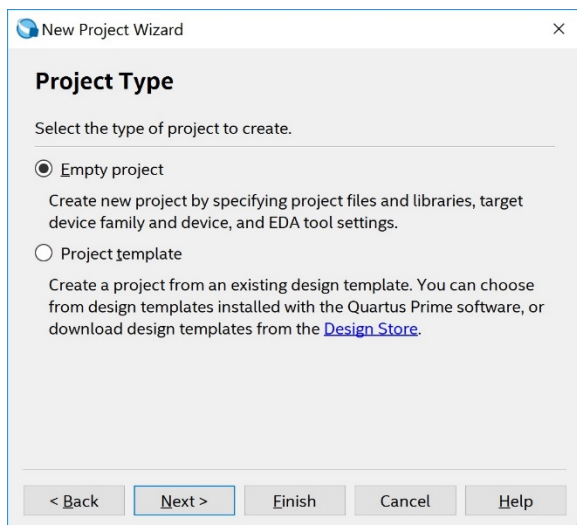


Figure 5.6
Quartus New Project Wizard (Project Type)

Leave the project type as “Empty” and click “Next”. The *Add Files* window shown in [Figure 5.7](#) will appear.

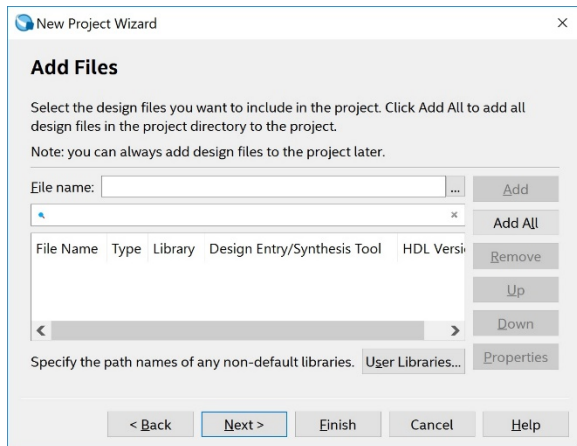


Figure 5.7
Quartus New Project Wizard (Add Files)

This window is where we can add existing design files. For this exercise we do not have any existing design files. Instead, we will be creating a new file called `top.vhd`. We will do this later outside of the New Project Wizard. Do not do anything in this window except click “Next”. The *Family, Device & Board Settings* window shown in Figure 5.8 will appear.

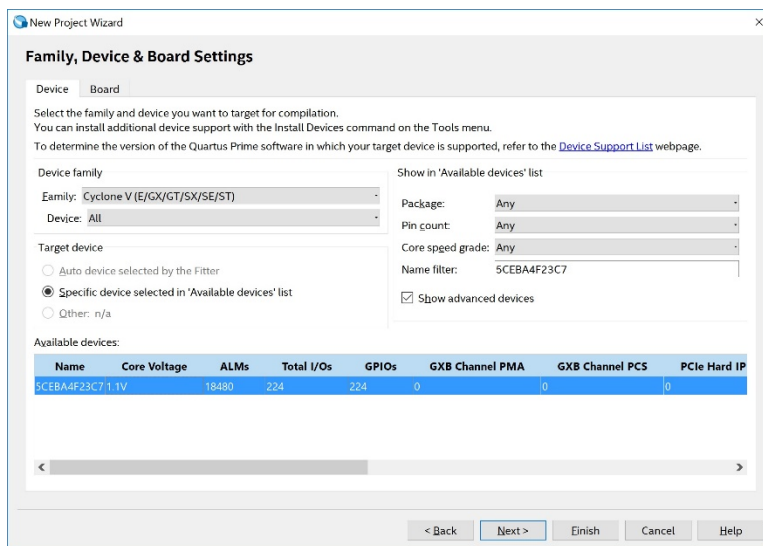


Figure 5.8
Quartus New Project Wizard (Family, Device & Board Settings)

In this window we tell the Quartus synthesizer which device we will be targeting. Since we only installed the Cyclone V FPGA family, only those devices will be shown. Notice that there are many different Cyclone V devices that can be selected. We want to select the FPGA device that is on our DE0-CV board. We can begin filtering down the selection by typing in the “Name filter” field on the right side of the pane. Begin typing `5CEBA4F23C7`. After each character is typed, it will reduce the number of devices that are available. Once you type the last character, there will only be one device available. Highlight this device and click “Next”. The *EDA Tool Settings* window shown in Figure 5.9 will appear.

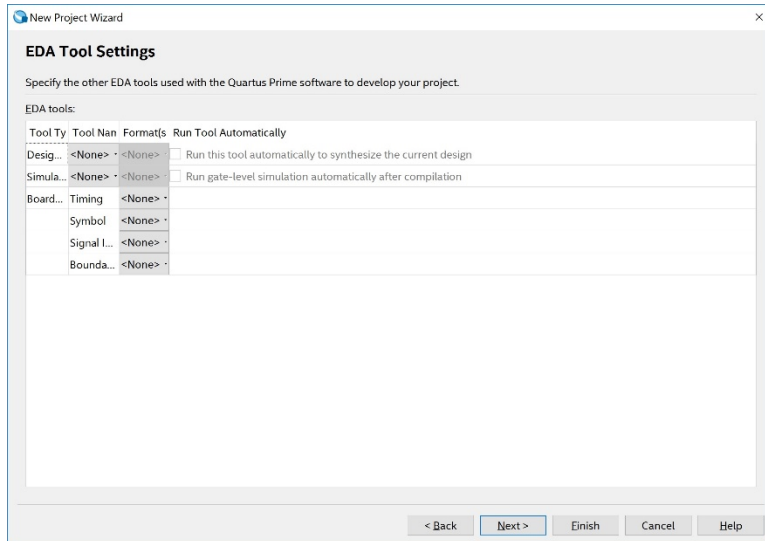


Figure 5.9
Quartus New Project Wizard (EDA Tool Settings)

In this window you can direct Quartus to read files from various CAD tools from other vendors. For this exercise, we will be using the Quartus tool by itself. Leave all the settings at *<None>* and click “Next”. The *Summary* window shown in [Figure 5.10](#) will appear.

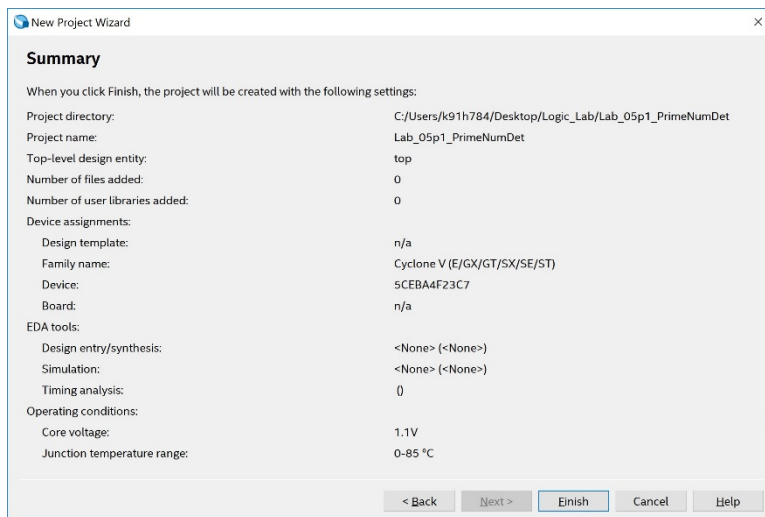


Figure 5.10
Quartus New Project Wizard (Summary)

Review the settings in this window and if correct, click “Finish”. If any of the settings are incorrect, you can click the “Back” button to go back and change them. You will now see a new blank project window with all of your settings as shown in [Figure 5.11](#).

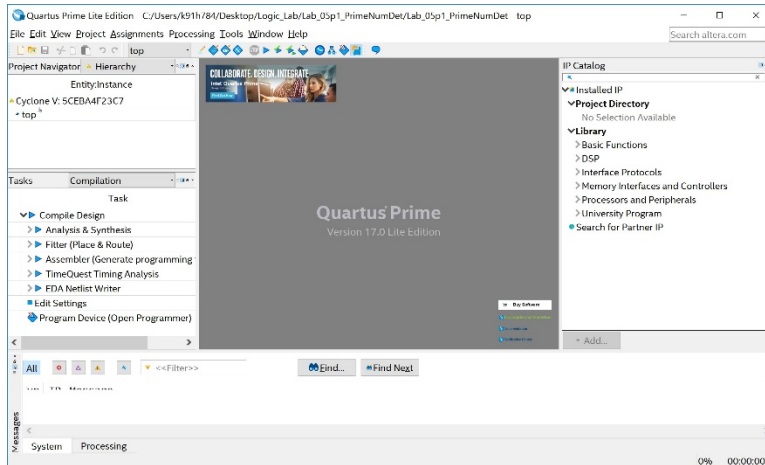


Figure 5.11
New Blank Project for the Cyclone V on the DE0-CV FPGA Board

[Create the top.vhd File for this Project](#)

Now we are ready to create the top.vhd file. In the Quartus window, use the pull-down menus to select: File → New. In the *New* window that appears, select “VHDL File” under the *Design Files* group. Click “OK”.

Now we need to save the file and name it accordingly. In the Quartus window, use the pull-down menus to select: File → Save As. By default, the name of the file should be called top.vhd and be located in your project directory. Verify that these settings are correct (if not, fix them) and click “Save”. The top.vhd is now open for editing. If you close this file it can be reopened by double clicking on it in the *Project Navigator* pane of the Quartus window.

[Enter the VHDL Entity](#)

We are now ready to enter the **VHDL entity** for the design. The entity contains all of the ports for the system. Based on the block diagram provided in [Figure 5.1](#), the ports are:

- **SW (3 downto 0)** This is the 4-bit input vector for the 4x slider switches on the DE0-CV board.
- **LEDR (3 downto 0)** This is the 4-bit output vector for the 4x red LEDs on the DE0-CV board.
- **Prime** This is the output will go to pin 2 on the GPIO_1 connector and jumper-wired to your breadboard.

Enter the following VHDL entity into your top.vhd. Notice that the entity name matches the file name in addition to the name of the top-level of the design in Quartus. This tells Quartus that any ports that are declared will be connected to pins on the FPGA board.

```
entity top is
  port (SW          : in  bit_vector (3 downto 0);
        LEDR       : out bit_vector (3 downto 0);
        Prime      : out bit);
end entity;
```

At this point your project should look like [Figure 5.12](#). Notice that Quartus recognizes the VHDL syntax and will color code based on the construct type.

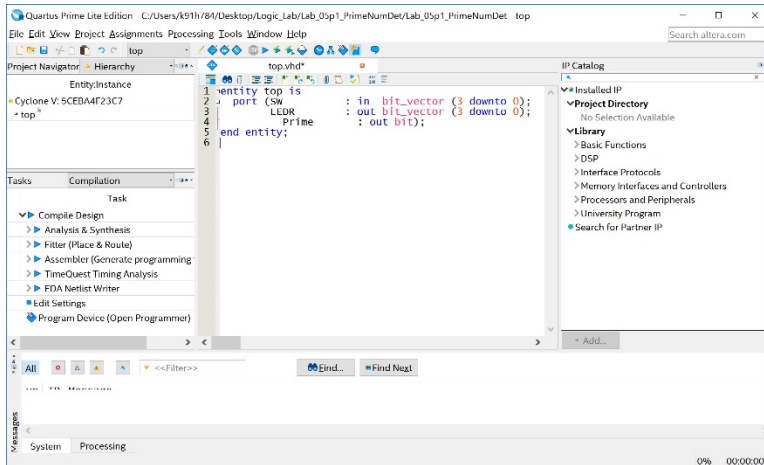


Figure 5.12
VHDL Entity for Prime Number Detector

[Enter the VHDL Architecture for the Switch-to-LED Circuit](#)

Now we are ready to enter the **VHDL architecture**. The functionality we are going to model for this part is to simply drive the slider switch values to the red LEDs. This can be done using a single concurrent signal assignment of SW to LEDR. Enter the following VHDL architecture into your top.vhd. Notice that since both LEDR and SW are both 4-bit vectors, a single signal assignment will handle assigning bit 0 to bit 0, bit 1 to bit 1, etc.

```
architecture top_arch of top is
begin
    LEDR <= SW;
end architecture;
```

Save your file using the pull-down menus File → Save. At this point, your project should look like [Figure 5.13](#).

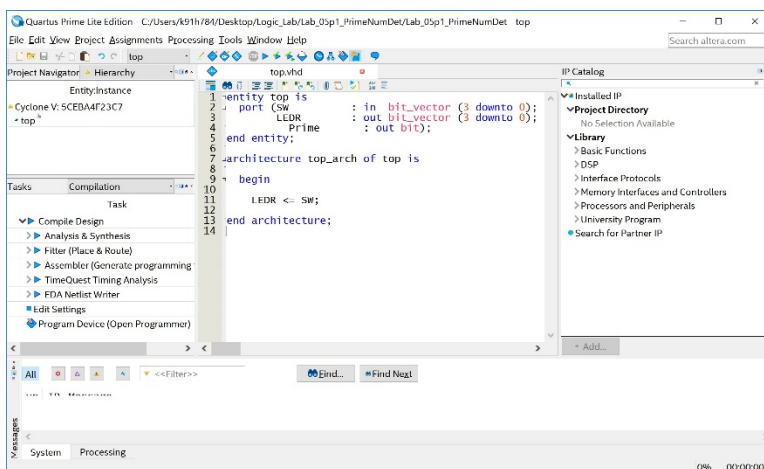


Figure 5.13
VHDL Architecture for Switch-to-LED Circuit

[Compile your Design](#)

Now we are going to compile the top.vhd file and fix any errors. To launch a task in Quartus, you double click on the desired task name within the *Task* pane on the left hand side of the window. Notice that underneath the task

“Compile Design” there are a handful of other tasks (i.e., Analysis & Synthesis, Fitter, etc.). When you click on “Compile Design”, Quartus will run all of the lower-level tasks. If it encounters an error, it will stop at the task where the error was found. If the task completes successfully, it will turn green. We typically allow Quartus to run as many tasks as it can when we compile to attempt full synthesis of the design. To launch the compiler and synthesizer, double click on “Compile Design”. This will take longer than a simple syntax check as Quartus is performing synthesis. The status of the task will appear in the *Messages* window at the bottom of the pane. You will also see a status bar for each of the tasks being performed. Once you have fixed any errors that have occurred and successfully completed the Compile Design tasks, you will see the status in [Figure 5.14](#).

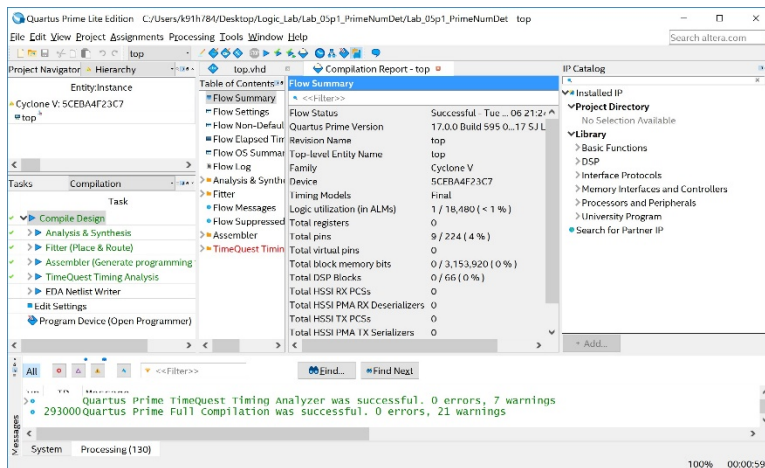


Figure 5.14
Quartus Window After Successful Compile and Synthesis

[Assign the Ports to the Pins of the FPGA](#)

We will now use a tool called *Pin Planner* to assign the ports of our top-level entity to the pins of the FPGA. Launch Pin Planner using the pull-down menus: Assignments → Pin Planner. You will see a graphical depiction of the FPGA with fields at the bottom to assign locations to any port that was in the entity during the most recent compile. At the bottom of Pin Planner, enter the pin locations for the 9x I/O of the Prime Number detector. The pin numbers will go in the “Location” column. Notice that there is a column called “Fitter Location” that has values for pin locations. When pins aren’t assigned manually during the first compile, Quartus will make automatic pin assignments for you. We **do not** want to use these locations. We want locations that correspond to the switches, LEDs, and GPIO_1 pin we are using in this exercise. Enter the following pin locations in the “Location” column of Pin Planner. To enter a value you can either click in the location field and type in the exact pin name or you can double click in the field to access the drop-down menu of available pins. Assign the following pin locations for this exercise.

- LED[3] PIN_Y3
- LED[2] PIN_W2
- LED[1] PIN_AA1
- LED[0] PIN_AA2
- Prime PIN_A12
- SW[3] PIN_T12
- SW[2] PIN_T13
- SW[1] PIN_V13
- SW[0] PIN_U13

Also change the “I/O Standard” setting for the output ports to “3.3-LVCMOS” and the “Current Strength” setting to “Maximum” for Prime. Once you are complete, you should see the results in [Figure 5.15](#).

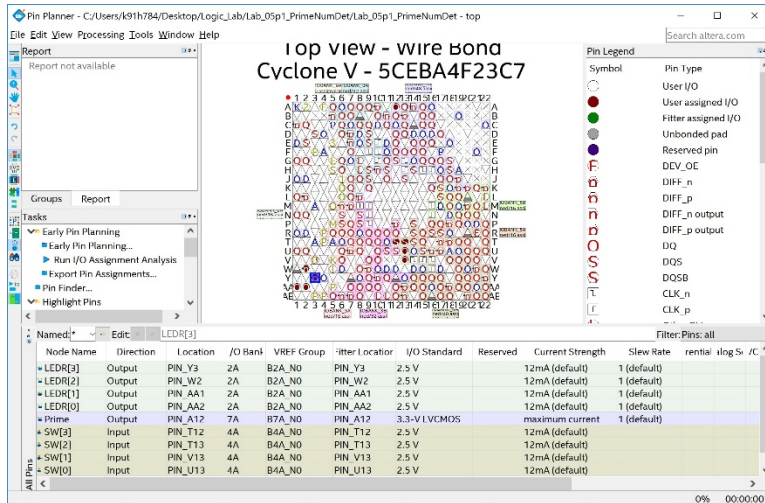


Figure 5.15
Quartus Pin Planner for the Prime Number Detector

Pin Planner does not have a save option. Instead, you simply close it using the drop-down menus: File – Close. Back in Quartus you'll notice that the tasks are no longer green. This indicates that the design needs to be compiled and synthesized again to take the new pin assignments into account. Double click on “Compile Design” and let it run until all tasks have been completed successfully.

[Program the FPGA with the Switch-to-LED Design](#)

We are now ready to download the prime number detector design to the FPGA. Plug in the DE0-CV board to your computer using the USB cable provided in the kit. You will power the DE0-CV board through the USB cable. **You do not need to plug in the AC adapter that is provided in the Terasic box.** All of the FPGA designs in this manual are small enough that the power from the USB cable is sufficient. Turn on the DE0-CV board by pressing the large red power button. When the board comes on, it will run a program that is stored in its non-volatile memory that flashes the I/O in a variety of patterns. We will be overwriting this program with our own.

In the Quartus task pane, double click on “Program Device”. This will bring up the *Programmer* tool. In Quartus versions 17 and newer, the programmer tool will automatically go out and discover any devices on the board that the current design will fit into. When the programmer finds the correct Cyclone V device, you will see the results in [Figure 5.16](#). Notice that the file containing the information on how to configure the FPGA to implement our design is `output_files/top.sof`. The `top.sof` file is the end result of the Quartus synthesis and is what is to be downloaded to the FPGA.

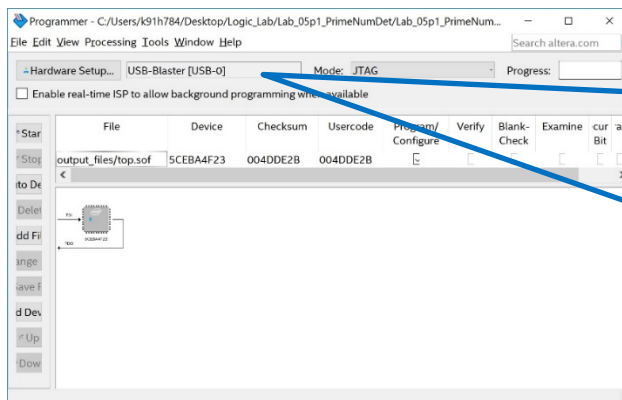


Figure 5.16
Quartus Programmer after Finding Device

If USB-Blaster isn't an option, you need to install the drivers manually so it will appear:

1. Plug in the USB cable between the computer and DE0-CV board, it will fail to load drivers.
2. Go to device manager in windows, go to USB controllers, right click on “Altera USB-Blaster” (it will have a yellow warning flag) – Update Software Driver
3. Choose Browse my computer
This is the key step: you browse to the following path:
`C:\intel\FPGA_lite\17.0\quartus\drivers\usb-blaster`
Then **STOP**, don't browse into the x32 or x64.
Next, install, etc.... follow prompts.

The next time you relaunch the programmer in Quartus, you should be able to select USB-Blaster in the dropdown box

If the programmer does not automatically find the device, you can press the “Auto Detect” button on the left of the programmer window. Once it finds the device, you need to assign the top.sof file. Highlight the device, right-click, and select “Add File”. Browse to the output_files folder and select top.sof.

When using the programmer for the first time, it may say “No Hardware” next to the hardware setup button. This means that the programmer is not using the correct drivers for the DE0-CV board. Click on the “Hardware Setup” button. For the “Currently Selected Hardware”, use the drop-down menus to select “USB-Blaster [USB-x]”. Click “Close”. Now the programmer will use the proper drivers to communicate with the DE0-CV board.

At this point we are ready to download the top.sof file to the FPGA in order to program it. Click on the “Start” button on the left side of the programmer window. The status of the programming will be displayed in the status bar in the upper-right corner of the window. When successful, it will say “100% (Successful)”.

[Test your Design to Drive the Switches to the LEDs of the DE0-CV](#)

Your design is now on the FPGA. You should be able to toggle the four slider switches on the DE0-CV and see the red LEDs turn on/off accordingly. If you are experiencing issues, you will need to go back and check each step in the Quartus design flow. The first place to start is in pin planner. Sometimes the pin locations will get dropped if you are still in edit mode when you close the pin planner window. Take a short video (<3 s) showing the proper operation of your switch-to-LED design. You should toggle through each of the four slider switches and verify that each red LED turns on. **This video satisfies the requirements for deliverable #1.**

[5.1.5.2 Implement a VHDL Design for a 4-Input Prime Number Detector](#)

[Connect the DE0-CV board to your Breadboard](#)

Now you are going to connect the DE0-CV board to your breadboard. Refer to figures [Figure 5.1](#) and [Figure 5.2](#) for the details of the connection. **Turn off** the DE0-CV board using the red power button. You will use 3x female-to-female jumper wires to make this connection. These jumper wires will plug onto three of the 0.1” pins on the GPIO_1 connector on the DE0-CV board. On the bread board side, you should use the 0.1” header pins to interface the female receptacle to your breadboard. One jumper wire will be used to connect the ground of the DE0-CV (pin 29 of GPIO_1) to the ground rails of your breadboard. A second jumper wire will be used to provide +3.4v (pin 30 of GPIO_1) to the power rails of your breadboard. A third jumper wire will connect the output of the detector *Prime* (pin 2 of GPIO_1) to the LED in your LED-driver circuit that is used to display the output of a logic circuit. You should also add a jumper wire on your breadboard to connect *Prime* to the input of your buzzer.

Once the connection is complete, turn on the DE0-CV board using the red power button. You should be able to toggle the slider switches on your breadboard and see the LEDs in your LED-driver circuit turn on/off. This verifies that your breadboard is receiving power and ground.

[Enter the VHDL Model for the Detector](#)

Now we are ready to enter the VHDL to implement the prime number detector. The VHDL for this detector will go in the architecture of your top.vhd after your signal assignment to display the DE0-CV slider switches on the red LEDs. You should implement your detector using either a conditional or selected signal assignment. These two approaches allow you to enter the desired functionality in an abstract, and easily interpreted form. Consider the following two code examples to help get you started.

Example of a *conditional signal assignment*:

```
Prime <= '0' when (SW = "0000") else
        '0' when (SW = "0001") else
        '1' when (SW = "0010") else -- the rest of the functionality goes below...
```

Example of a *selected signal assignment*:

```
with (SW) select
  Prime <= '0' when "0000",
          '0' when "0001",
          '1' when "0010", -- the rest of the functionality goes below...
```

Enter the VHDL for the detector in your `top.vhd` file in Quartus. You can choose whichever modeling method you'd like (i.e., conditional or selected). You'll need to complete the signal assignments given in the examples above for each input condition. Note that the above syntax uses *single apostrophes* and not the *back ticks*. If you try to copy/paste the above syntax directly into Quartus, it may paste incorrectly.

[Compile and Synthesize your Design](#)

Make sure to save your design and then double click on "Compile All" to perform a full synthesis of your design. Fix any errors you encounter and re-compile until all tasks are successful.

[Program the FPGA with your New Design](#)

After the synthesis is complete, download your design to the FPGA. When you launch the programmer tool (if it isn't already running), it will automatically point to the updated `top.sof` file to be downloaded. You will be able to simply click "Start" on the programmer to download your design.

[Test your Prime Number Detector Design](#)

Test your prime number detector by cycling through all possible 16 input codes on the DE0-CV slider switches and observing the *Prime* output on the breadboard LED and buzzer. Take a short video (<5 s) showing the proper operation of your prime number detector. You should show that the LED and buzzer on your breadboard assert for each prime number on the input. **This video satisfies the requirements for deliverable #2.**

5.1.5.3 [Save a Copy of your top.vhd for your Records](#)

You now want to locate for your records the `top.vhd` file for the third deliverable for this exercise. This file is located in your main working directory (`Desktop\Logic_Lab\Lab_05p1_PrimeNumDet\top.vhd`). Go into your working directory for this project and locate this file. **This file satisfies the requirements for deliverable #3.**

After you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 5.1 After completing this lab exercise, can you:

- Use the modern digital design flow to take a VHDL model for a combinational logic circuit and synthesize it for implementation on an FPGA?
- Interface an FPGA board to a breadboard to successfully transfer digital signals between the two systems.

Chapter 6: MSI Logic

Lab 6.1: 4-Input, 7-Segment Display Decoder (in VHDL)

6.1.1 Objective

The objective of this lab is to gain further experience with implementing combinational logic circuits using hardware description languages, implementing the systems on programmable logic devices, and interfacing two digital systems together. This lab will also demonstrate how to create a new Quartus project by copying a prior project so that design components can be reused.

6.1.2 Learning Outcomes

After completing this lab, you will be able to:

- Create a new Quartus project by copying your prime number detector project from lab 5.1 so that you can reuse components of its design.
- Create a 4-input, 7-segment display decoder (0_{16} to F_{16}) on an FPGA using VHDL.

6.1.3 Parts Needed

- Breadboard + wires.
- LED driver circuit from prior lab (8-position slider switch, 10 k Ω resistor network, 330 k Ω resistor network, 5x red LEDs, 1x 74HC04 inverter IC).
- Buzzer circuit from prior lab (magnetic buzzer, 2N3904 NPN transistor, 1N4002 diode, 10 k Ω axial resistor).
- 7-Segment display circuit from prior lab (7-segment display, 7x 150 Ω resistors).
- DE0-CV FPGA board.
- 10x female-to-female jumper wires.

6.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a 4-input, 7-segment display decoder + 4-input prime number detector implemented with VHDL on an FPGA (90% of exercise).
2. Provide your top.vhd design file (10% of exercise).

6.1.5 Lab Work & Demonstration

6.1.1.1 Implement a VHDL Design for a 7-Segment Decoder + 4-Input Prime Number Detector

You are going to design a 7-segment display decoder in VHDL. The decoder will take in the values from the 4x slider switches on the DE0-CV and output the necessary logic to display the characters on a display on your breadboard. The display will show characters 0_{16} to F_{16} corresponding to the 4-bit input. You will also reuse your prime number detector from lab 5.1 in order to assert the buzzer and an LED on your breadboard whenever the display is showing a prime number. Once again, the prime numbers between 0_{10} and 15_{10} are: 2, 3, 5, 7, 11 (b_{16}) and 13 (d_{16}). You will also show the values of the slider switches on the LEDs on the DE0-CV. [Figure 6.1](#) shows the block diagram for the 7-segment decoder system in this exercise.

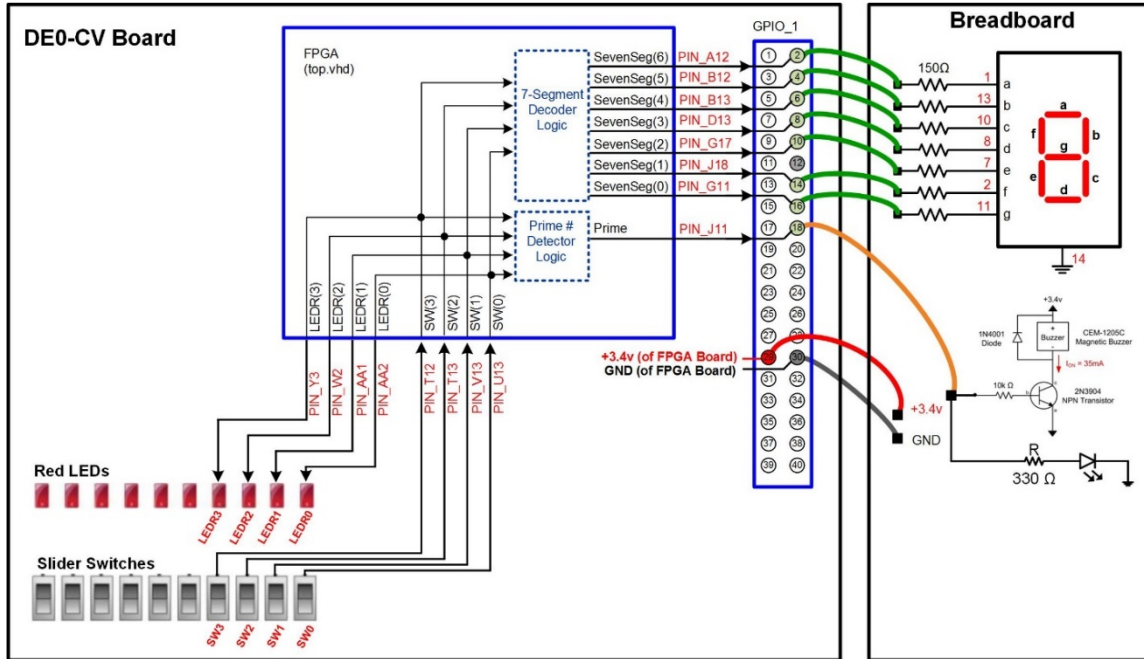


Figure 6.1
Block Diagram of the 7-Segment Decoder System

Figure 6.2 a picture the 7-segment decoder system.

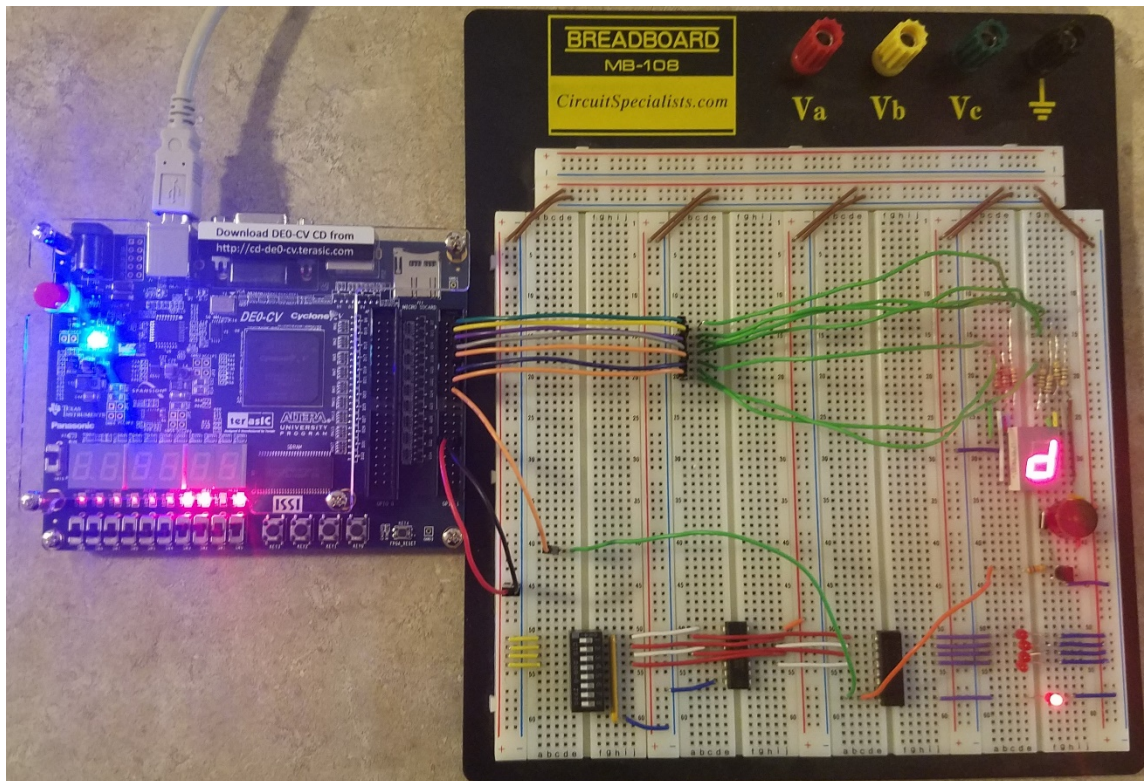


Figure 6.2
Picture of the 7-Segment Decoder System on the DE0-CV Board + Breadboard Interface

Determine the Logic Functionality for the Decoder

Before you can model the decoder in VHDL, you need to design the functionality you wish to model. In a prior lab, you created a 3-input 7-segment decoder using discrete parts by first completing a table for the logic of each of the 7 LEDs in the display. In this lab, you will need to redo the table to support all characters corresponding to a 4-bit input. The table in Figure 6.3 is provided to assist you. Complete this table to determine the logic for the 7 outputs that will drive the character display. The first row is completed to get you started.

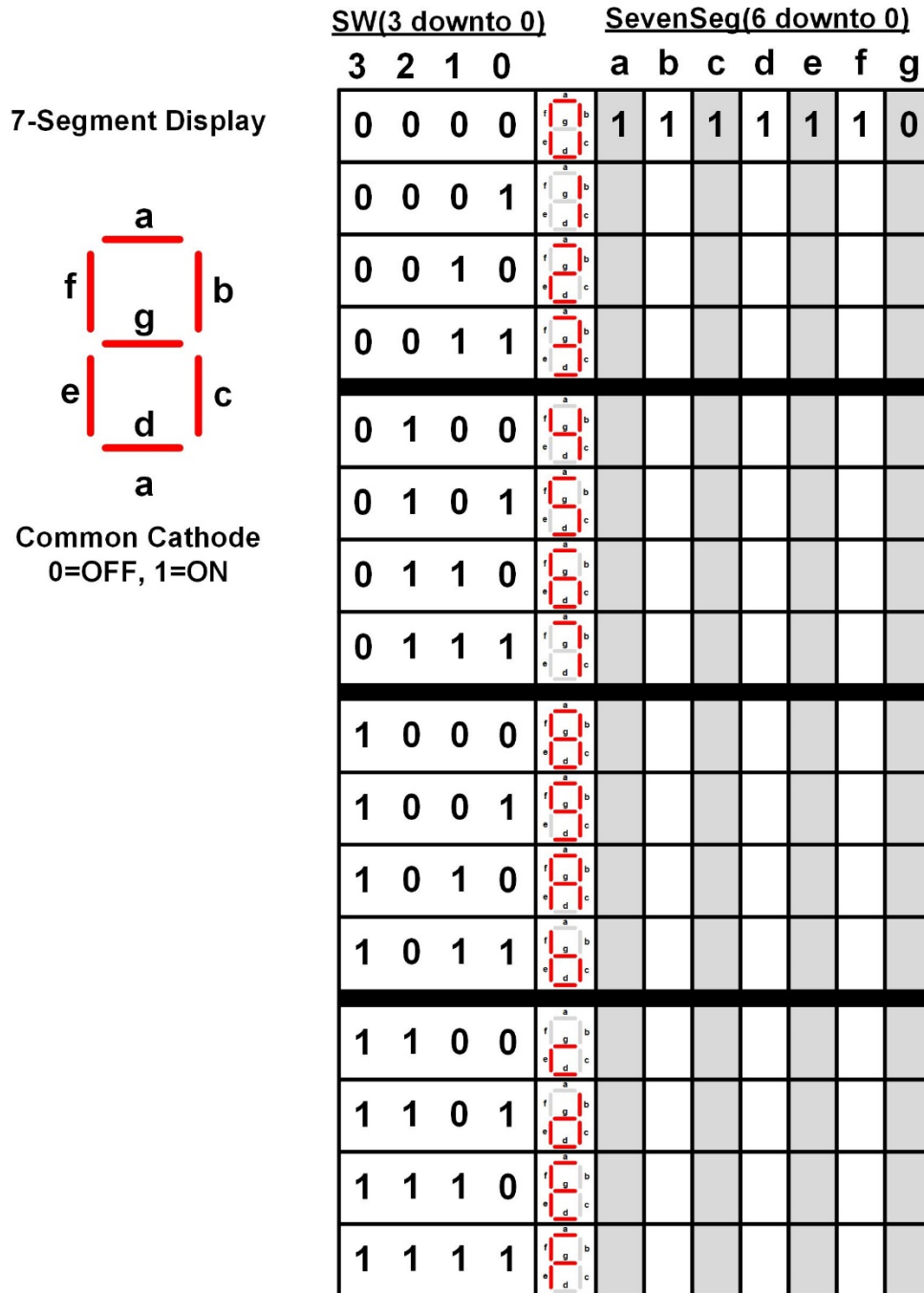


Figure 6.3
Truth Table for 4-input, 7-Segment Display Logic

Create a New Quartus Project by Copying your Prior Prime Number Project

In lab 5.1, we created a Quartus project for a 4-input prime number detector. We want to reuse many of the design aspects of this prior project including the FPGA selection, the logic to drive the slider switches to the LEDs, and the pin assignments for the slider switches and LEDs. Quartus provides the ability to create a new project by copying over a prior one.

Launch Quartus and then open your project for lab 5.1 using the pull-down menus: File → Open Project. You'll want to browse to the project file at: "Desktop\Logic_Lab\Lab_05p1_PrimeNumDet\Lab_05p1_PrimeNumDet.qpf" and select "Open".

Now you can copy the open project to a new project using the pull-down menus: Project – Copy Project. The *Copy Project* window allows you to give the new project a different name and create a folder for its location at the same time. Give the name of the project and folder "Lab_06p1_SevenSegDecoder". Also check the box to open the project. Your *Copy Project* settings window should look like [Figure 6.4](#).

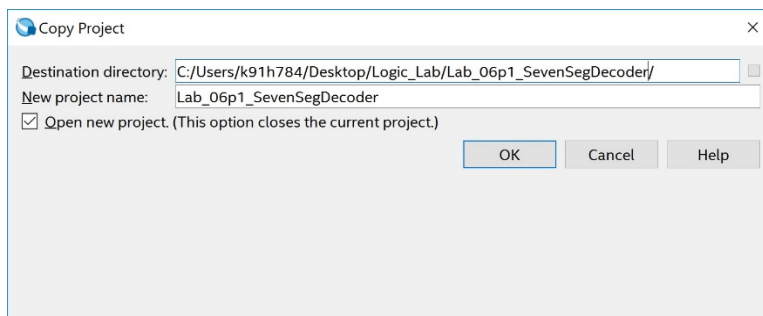


Figure 6.4
Quartus Copy Project Settings

Click "OK" to copy the project. Quartus will prompt you to verify that you want a new folder created for this project. Click "Yes".

Update the VHDL Entity for the 7-Segment Decoder

In the new project, double click on "top.vhd". You will see the VHDL from lab 5.1. The first thing we need to do is update the entity to reflect the new ports needed for the 7-segment decoder. We will be re-using the ports SW, LEDR, and Prime. The new ports for this design are:

- **SevenSeg(6 downto 0)** This is a 7-bit vector that will drive the individual LEDs on the 7-segment display. The MSB of this vector will correspond to the "a" LED while the LSB of this vector will correspond to the "g" LED.

Update the VHDL entity to reflect the new output port SevenSeg(6 downto 0). [Figure 6.5](#) shows how the entity should look.

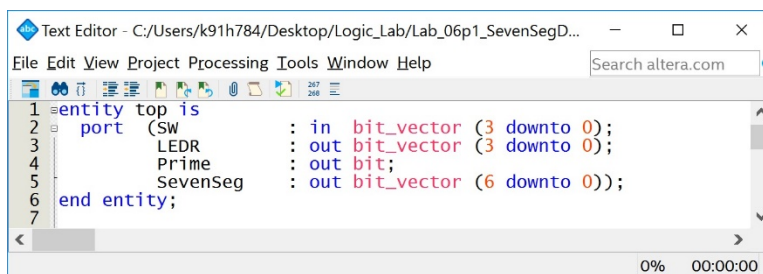


Figure 6.5
VHDL Entity for 7-Segment Decoder

[Enter the Functionality for the 7-Segment Decoder to the VHDL Architecture](#)

Now you will enter the functionality for the 7-segment decoder. Leave all of the VHDL functionality from lab 5.1 in your architecture. You can enter the functionality for the 7-segment decoder directly from the table in [Figure 6.3](#). You can use either a conditional or selected signal assignment to model the decoder. Consider the following two code examples to get you started.

Example of a *conditional signal assignment*:

```
SevenSeg <=  "1111110" when (SW = "0000") else
             "0110000" when (SW = "0001") else
             "1101101" when (SW = "0010") else -- remaining functionality goes below...
```

Example of a *selected signal assignment*:

```
with (SW) select
  SevenSeg <= "1111110" when "0000",
             "0110000" when "0001",
             "1101101" when "0001", -- remaining functionality goes below...
```

After completing your model for the 7-segment display, save your design and perform a "Compile All". Fix any syntax errors you may have and repeat compiling until all of the tasks complete 100%.

[Assign the Pins for the 7-Segment Decoder Ports](#)

Once your design successfully compiles, you can assign the pins for the SevenSeg port and update the assignment for Prime. Open the pin planner tool using the pull down menus: Assignments → Pin Planner. You should see existing assignments for the SW, LEDR, and Prime ports. You will leave the SW and LEDR assignments as is. You will need to update the pin assignment for the port Prime to reflect the pin connection shown in [Figure 6.1](#). The updated pin assignments should be made as follows:

- Prime PIN_J11
- SevenSeg[6] PIN_A12
- SevenSeg[5] PIN_B12
- SevenSeg[4] PIN_B13
- SevenSeg[3] PIN_D13
- SevenSeg[2] PIN_G17
- SevenSeg[1] PIN_J18
- SevenSeg[0] PIN_G11

Also change the "I/O Standard" settings for the output ports to "3.3-LVCMOS" and the "Current Strength" settings to "Maximum" for Prime and SevenSeg. When complete, your assignments should look like the settings in [Figure 6.6](#).

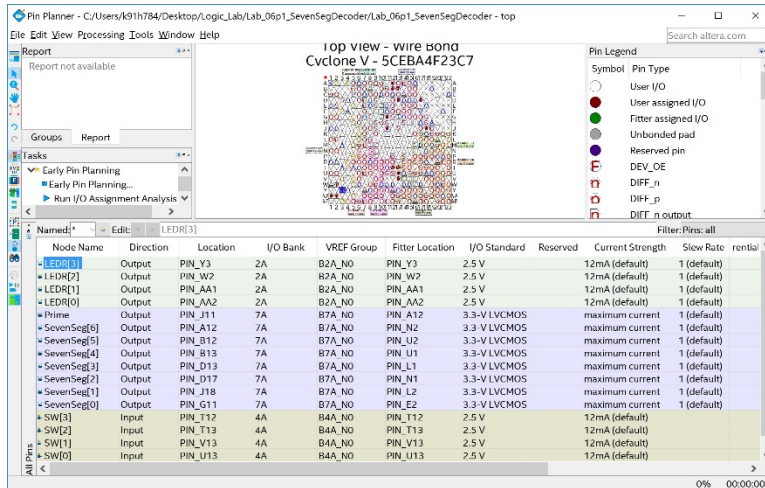


Figure 6.6
Pin Assignments for 7-Segment Decoder

Close the Pin Planner tool using the pull-down menus: File → Close. Recompile your design with the new pin assignments using the “Compile All” task.

[Connect the DE0-CV board to your Breadboard](#)

Now you are going to connect the DE0-CV board to your breadboard. Refer to figures [Figure 6.1](#) and [Figure 6.2](#) for the details of the connection. Make sure that the DE0-CV board is not plugged into your computer when doing the wiring. You will use 10x female-to-female jumper wires to make the connection for this exercise. These jumper wires will plug onto ten of the 0.1” pins on the GPIO_1 connector on the DE0-CV board. On the bread board side, you should use the 0.1” header pins to interface the female receptacle to your breadboard. One jumper wire will be used to connect the ground of the DE0-CV to the ground rails of your breadboard. A second jumper wire will be used to provide +3.4v to the power rails of your breadboard. Then seven additional jumper wires will connect the outputs of the 7-segment decoder logic to your character display on your breadboard. Finally, a tenth jumper wire will connect the *Prime* output to the LED in your LED-driver circuit. You should also add a jumper wire on your breadboard to connect *Prime* to the input of your buzzer.

Once the connection is complete, plug in the USB cable between the DE0-CV board and your computer. Turn on the DE0-CV board using the red power button. You should be able to toggle the slider switches on your breadboard and see the LEDs in your LED-driver circuit turn on/off. This verifies that your breadboard is receiving power and ground.

[Program the FPGA with your 7-Segment Decoder Design](#)

Now you are ready to download your design to the FPGA. When you launch the programmer tool (if it isn’t already running), it will automatically point to the updated top.sof file to be downloaded. You will be able to simply click “Start” on the programmer to download your design.

[Test your 7-Segment Decoder Design](#)

Test your 7-segment decoder by cycling through all possible 16 input codes on the DE0-CV slider switches and observing the outputs on the 7-segment display of your breadboard. For each input that is a prime number, your Prime output should assert the LED and buzzer on your breadboard. If your design is not working correctly, you will need to debug it. A few tips on debugging:

- If an LED segment in your display is not lighting up, ensure it is wired correctly. You can do this by taking the wire that is to be attached to the FPGA pin and instead plug it into the power rail of your breadboard. This will drive the LED directly and ensure that you have the resistor and character display pin correctly wired.

- If an LED segment still doesn't light up, check pin planner in Quartus.

Take a short video (<5 s) showing the proper operation of your 7-segment display decoder. You should show that the correct character is displayed for the corresponding input code on the slider switches. You should also show that the LED and buzzer on your breadboard assert for each prime number on the input. **This video satisfies the requirements for deliverable #1.**

6.1.1.2 Save a Copy of your *top.vhd* for your Records

Save a copy of your *top.vhd* file for the second deliverable of this exercise. Recall that this file is located in your main working directory. Go into your working directory for this project and locate this file. **This file satisfies the requirements for deliverable #2.**

When you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 6.1 After completing this lab exercise, can you:

- Take advantage of design re-use by creating a new project by copying a prior design in Quartus?
- Use concurrent signal assignment modeling techniques to create a 7-segment decoder design in VHDL and implement it using the modern digital design flow?

Chapter 7: Sequential Logic Design

Lab 7.1: 4-Bit Ripple Counter & Switch Debouncing

7.1.1 Objective

The objective of this lab is to introduce discrete sequential logic circuits. This lab will cover the design of a *ripple counter* using discrete D-flip-flops. This lab will then examine the behavior of mechanical switches and introduce debounce circuitry that helps provide clean edges from a switch by taking advantage of the storage capability of an S'R' latch.

7.1.2 Learning Outcomes

After completing this lab you should be able to:

- Design a ripple counter using discrete D-flip-flops.
- Explain why mechanical switches produce unclean edges.
- Explain how a NAND-debounce circuit works.
- Setup an oscilloscope for a single-shot measurement.
- Demonstrate the response of a mechanical switch before and after applying a NAND-debounce circuit.

7.1.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 2x 74HC74 dual, rising edge triggered D-flip-flop ICs.
- 4x, 150 Ω axial resistors.
- 4x, discrete red LEDs.
- Push button switch, SPDT.
- 1x 74HC00 2-input NAND gate IC.
- 2x 1k Ω axial resistors

7.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a 4-bit ripple counter implemented with discrete D-flip-flops. The counter will be clocked by the AWG and the outputs will be observed on LEDs. (40% of exercise).
2. Take a logic analyzer measurement of the 4-bit ripple counter (20% of exercise).
3. Demonstrate the *break-before-make* and *contact bounce* behavior found in mechanical switches. You will clock your counter using a simple push button switch and observe the unclean clock edges into your counter using a single-shot measurement on the oscilloscope. (20% of exercise).
4. Demonstrate a NAND-debounce circuit that produces clean logic transitions for the clock of your ripple counter. This will be implemented on your push button switch and observed using a single-shot measurement on the oscilloscope (20% of exercise).

7.1.5 Lab Work & Demonstration

7.1.5.1 Implement a 4-Bit Ripple Counter using D-Flip-Flops

A ripple counter is made from D-flip-flops that are connected in a *toggle flop* configuration (e.g., the Qn output is wired back to the D input). This configuration provides an output on the Q of the D-flip-flop that has a frequency exactly $\frac{1}{2}$ of the incoming clock. An inverted version of this divided down signal (e.g., Qn) can then be used to clock the next stage of the counter to provide a signal that is $\frac{1}{4}$ of the original clock. These divided down signals provide a simple

binary counter and can be scaled to whatever size is desired by adding more D-flip-flops. Figure 7.1 shows the logic diagram of a 4-bit ripple counter.

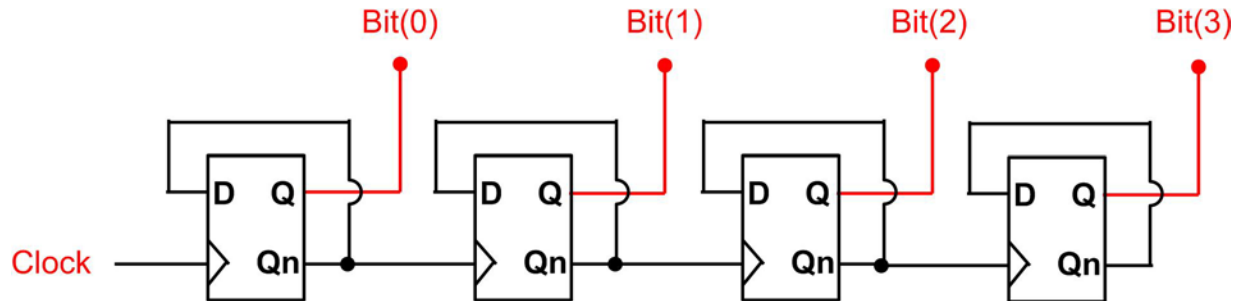


Figure 7.1
Logic Diagram of a 4-bit Ripple Counter Created with Discrete D-flip-flops

Breadboard the 4-Bit Ripple Counter

Breadboard the 4-bit ripple counter shown in Figure 7.1 using the 74HC74 discrete D-flip-flops provided in your lab kit. Note that each of these parts contains two separate D-flip-flops. Also note that each D-flip-flop has dedicated reset (CLRn) and preset (PREn) inputs. To enable the D-flip-flop, both of these pins should be pulled to a logic 1 (de-asserted) by wiring the pins to Vcc. Keep in mind that both D-flip-flops have the CLRn and PREn lines, so you will need FOUR wires for each 74HC74 part. Connect the 4-bit output of your counter to 4x resistor-LED circuits. You should use discrete 150 Ω axial resistors in series with discrete red LEDs with their cathodes connected to GND.

Connect the power and GND of the Analog Discovery to the power and ground rails of your breadboard. Also connect the AWG channel 1 (W1) of the Analog Discovery to the clock input of the ripple counter. Finally, connect logic channels 0, 1, 2, and 3 to the outputs of the ripple counter. Your connection should look like Figure 7.2.

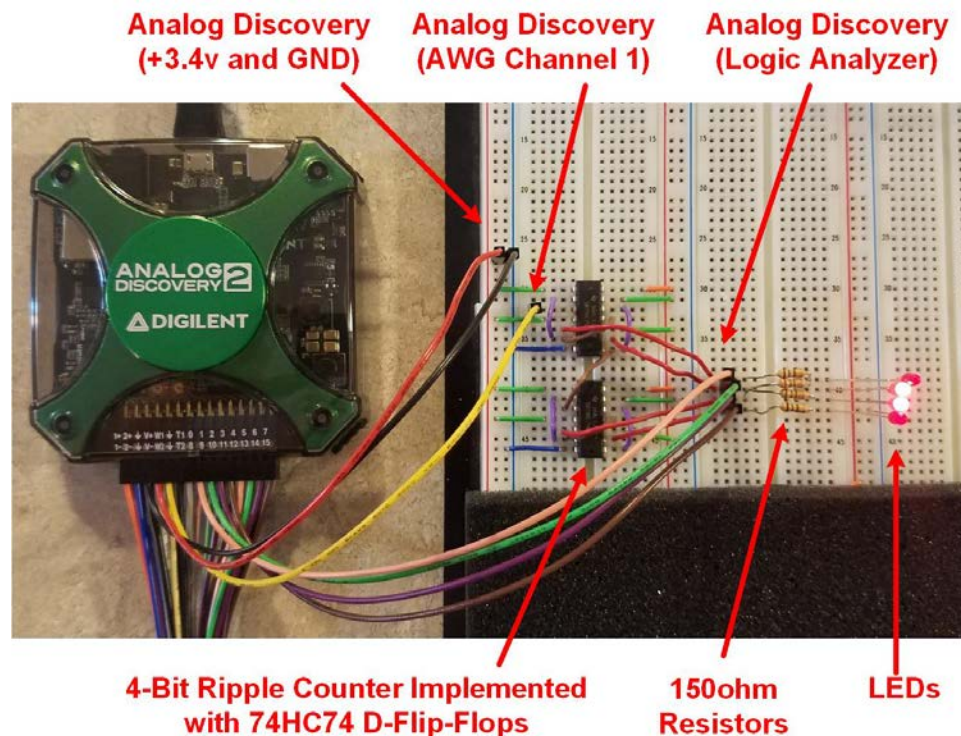


Figure 7.2
Breadboard Connections for the 4-Bit Ripple Counter

Configure the Analog Discovery

Launch Waveforms. Configure the power supply to output a +3.4v DC voltage for your breadboard. Run the power supply tool to power your breadboard.

Next, configure channel 1 of the AWG to output a square wave with the following settings:

- Type = Square
- Frequency = 4 Hz
- Amplitude = 1.7 V
- Offset = 1.7 V
- Symmetry = 50%
- Phase = 0°

Enable the AWG by pressing the “Run” button.

Test your Ripple Counter

At this point you should see a 4-bit binary counter on your LEDs. Take a short video (<5 s) showing the proper operation of your counter. **This video satisfies the requirements for deliverable #1.**

7.1.5.2 Take a Logic Analyzer Measurement of the 4-Bit Ripple Counter

Now we want to take a logic analyzer measurement of the 4-bit counter. First, set the frequency of the AWG to 1 kHz so that we can see more data on the screen of the measurement. In the logic analyzer tool, create a new bus called “Ripple_CNT” and add channels 0, 1, 2, and 3 to it. Run the logic analyzer by pressing the “Run” button. Set the position to 0 s and the timescale to 5 ms/div. Observe both the expanded view of the bits within the counter and the bus view. The bus view will display the decimal value of the counter by default. You should see the counter increment from 0 to 15 and then roll-over and start counting at 0 again. Your measurement should look like [Figure 7.3](#).

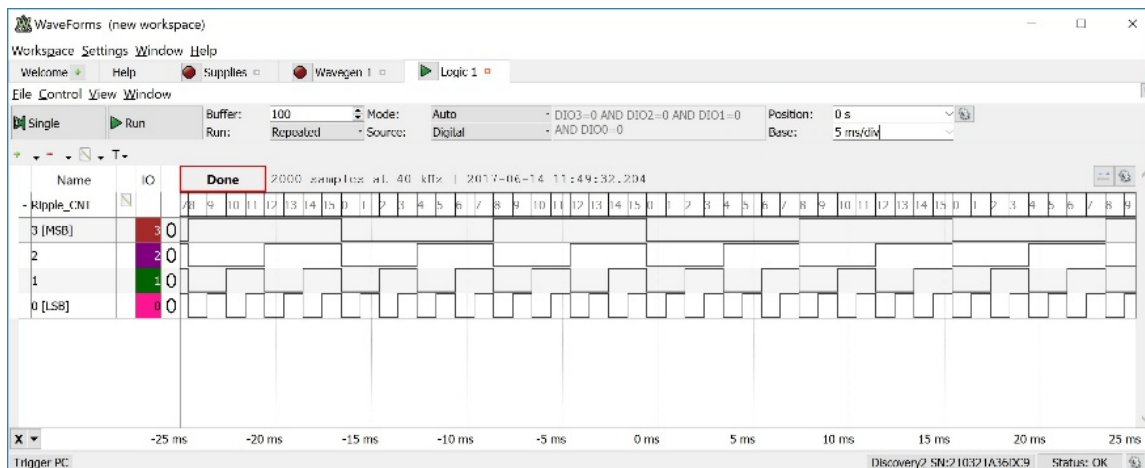


Figure 7.3
Logic Analyzer Measurement of 4-Bit Ripple Counter

Take a screenshot of the logic analyzer measurement of the ripple counter. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

7.1.5.3 Observing Issues with Mechanical Switches

One way to generate a clock edge for a sequential circuit is using a mechanical push button. [Figure 7.4](#) shows a circuit for a simple clock generator using a SPDT switch. Notice that when the button is *not* pushed, the clock output is a logic 0 because it is connected to GND through the switch. When the button is *is* pushed, the clock output is a logic 1 because it is connected to V_{CC} through the switch. In theory, this circuit should produce a rising edge of a clock each time the button is pressed; however, there are a variety of issues when using mechanical switches to produce logic

transitions. These include the *break-before-make* characteristic of a SPDT switch and *bounce* associated with the mechanical contact within the part. Refer to the section on switch bounce in Chapter 7 of the textbook describing these issues. In this part of the lab exercise, we will observe these issues with an oscilloscope.

A BAD CLOCK CIRCUIT

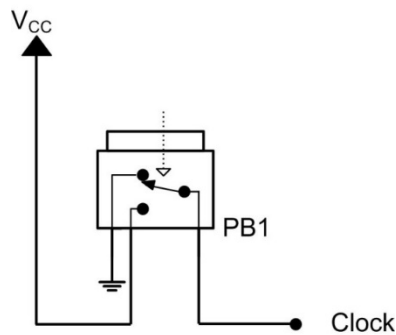


Figure 7.4
Generating a Clock Directly from a Mechanical Switch

Clock your Ripple Counter with the “Bad” Clock Generation Circuit

Breadboard the *bad* clock generation circuit in Figure 7.4 and drive the input clock of your 4-bit ripple counter with it. You will need to first disconnect the AWG. In waveforms, turn off the AWG and then disconnect it from your breadboard. Connect oscilloscope channel 1+ of the Analog Discovery to the clock signal being generated by your switch circuit. Connect the channel 1 reference (1-) to the ground rail of your breadboard. Your connection will look like Figure 7.5.

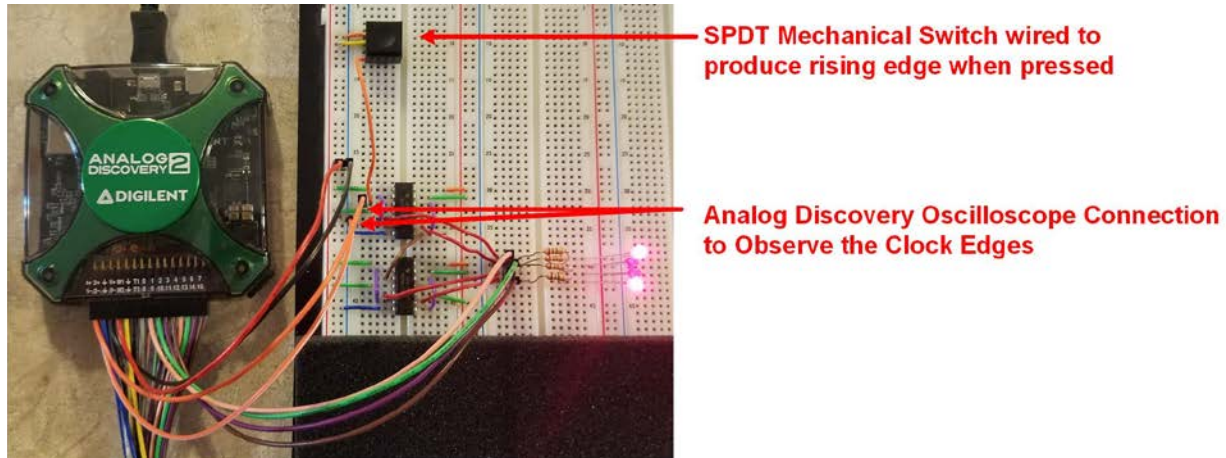


Figure 7.5
Breadboard Connections for the Mechanical Switch Clock Generator

You should now be able to press the switch and see your ripple counter change patterns. Notice that it will not always increment by 1 on every press. Instead, it may jump 2 or 3 counts per press. This is because the switch is not producing a clean clock edge. The break-before-make behavior and bouncing of the contact can cause multiple clock edges to be produced for each press.

Take a Single-Shot Oscilloscope Measurement of the Switch Clock Edge

Setup the oscilloscope to measure the clock edge waveform. To setup the oscilloscope for this measurement, first launch the *Scope* tool in waveforms. Configure the time settings to zoom in on the rising edge and position the measurement in the middle of the screen horizontally as follows:

- Position = 0 s
- Base = 0.5 us/div

Configure the Channel 1 settings to position the waveform in the middle of the screen vertically as follows:

- Offset = -1.5 V
- Range = 0.5 V/div

Now configure the trigger to take a *single shot measurement*. This type of measurement will only acquire data when it sees the trigger condition. Once the trigger condition is observed, the oscilloscope will fill the screen with data and then stop. This allows you to measure events that are not periodic or that occur infrequently. The trigger settings for the scope tool are along the top of the measurement screen. Configure these as follows:

- Mode = Normal
- Source = Channel 1
- Condition = Rising
- Level = 1.7 V

Now press the “Single” button. You will see the status of the oscilloscope is now *Armed* and waiting for the trigger. Once you press the switch on your breadboard, the oscilloscope will see the rising edge, trigger, fill the screen with data, stop, and display its status as *Done*. Press the switch and observe the waveform. You should repeat this measurement (i.e., press “Single”, then press the switch) until you get a rising edge that exhibits both the break-before-make behavior and switch bounce. An example measurement showing both behaviors is shown in [Figure 7.6](#).

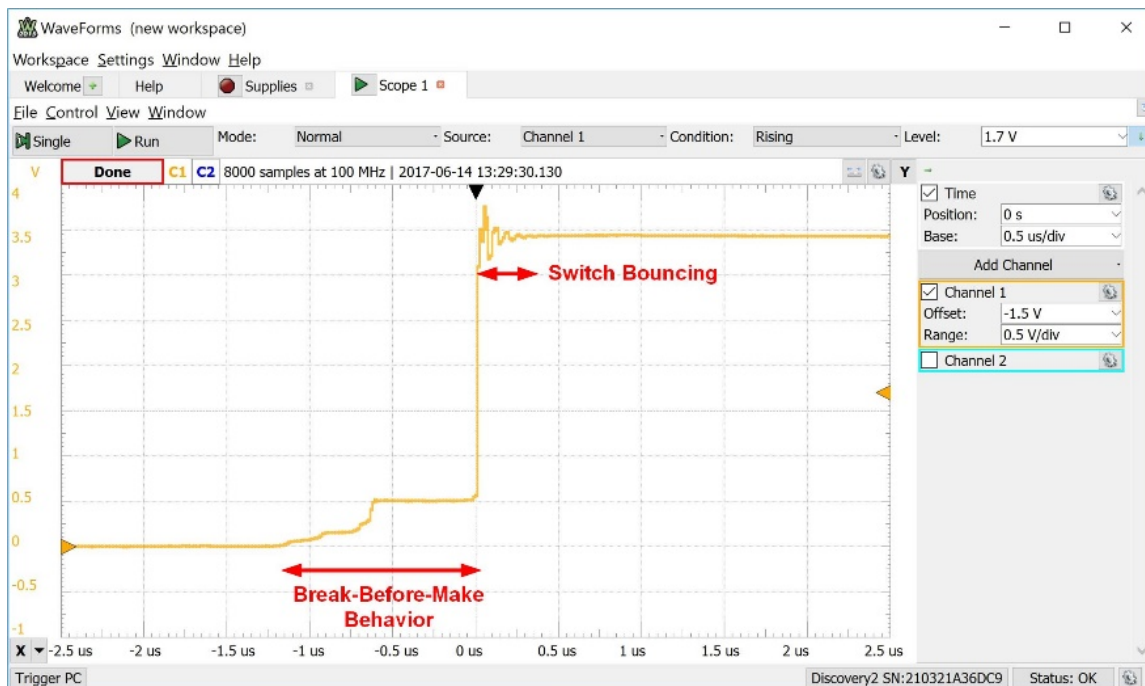


Figure 7.6
Oscilloscope Measurement of Switch showing Break-Before-Make and Bouncing Behaviors

Take a screenshot of the oscilloscope measurement of your switch output. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #3.**

7.1.5.4 Debouncing Mechanical Switches

[Figure 7.7](#) shows a *NAND-Debounce* circuit. This type of circuit eliminates the problems with SPDT mechanical switches observed in the prior measurement. This circuit by taking advantage of the storage capability of an S'R' latch. This circuit will hold the past value when the switch contact is in the *break* (or unconnected) zone. This handles the

initial issue observed in Figure 7.7 as the output will hold a zero when the switch contact enters the no-contact portion of the transition. This circuit also handles the final issue observed in Figure 7.7 because once the switch contacts V_{CC} , it will then bounce between V_{CC} and open. The NAND-debounce circuit will hold a one in this situation as the switch contact bounces between V_{CC} and no-contact.

A CLOCK CIRCUIT w/ NAND-DEBOUNCE

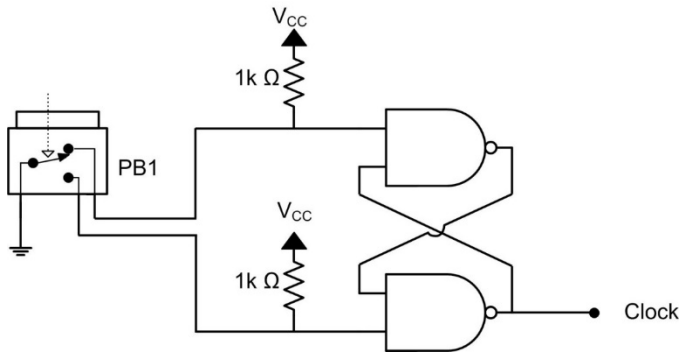


Figure 7.7
NAND-Debounce Circuit for SPDT Mechanical Switches

Breadboard the circuit in Figure 7.7 and use its output to drive your ripple counter. Repeat the single-shot oscilloscope measurement to observe the new clean clock edge. Your clock edge will now look similar to Figure 7.8.

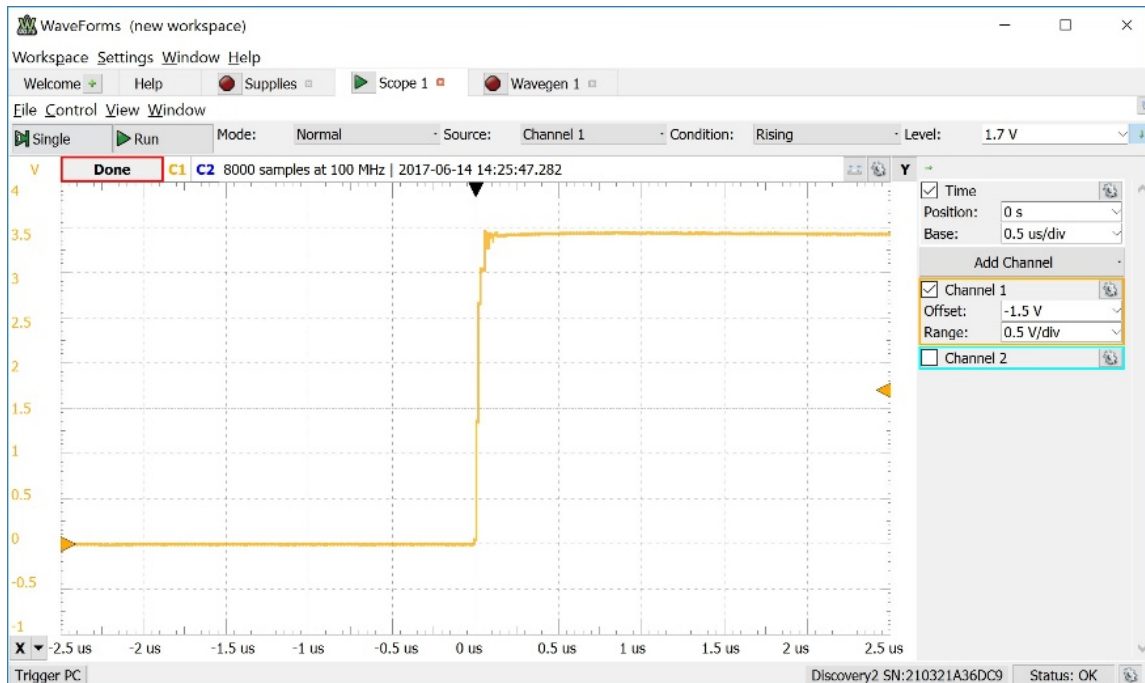


Figure 7.8
Oscilloscope Measurement of Debounced Switch Output

Take a screenshot of the oscilloscope measurement of your debounced switch output. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #4.**

CONCEPT CHECK

Lab 7.1 After completing this lab exercise, can you:

- Design a ripple counter using discrete D-flip-flops?
- Explain why mechanical switches produce unclean edges?
- Explain how a NAND-debounce circuit works?
- Setup an oscilloscope for a single-shot measurement?
- Demonstrate the response of a mechanical switch before and after applying a NAND-debounce circuit?

Lab 7.2: 3-Bit Binary Up/Down Counter

7.2.1 Objective

The objective of this lab is to gain experience implementing finite state machines using discrete parts. You will design and implement a 3-bit binary up/down counter.

7.2.2 Learning Outcomes

After completing this lab you should be able to:

- Design and implement a 3-bit binary up/down counter using discrete parts.
- Use a push button circuit to provide the reset to your counter.

7.2.3 Parts Needed

- Breadboard + wires.
- Analog Discovery 2.
- 2x 74HC74 dual, rising edge triggered D-flip-flop ICs.
- 3x, 150 Ω axial resistors.
- 3x, discrete red LEDs.
- Push button switch, SPDT.
- A variety of HC74 discrete logic ICs to implement combinational logic functions.

7.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Provide a final logic diagram for the 3-bit counter (10% of exercise).
2. Demonstrate a 3-bit binary up/down counter FSM with discrete components. The counter will be clocked by the AWG, the outputs will be observed on LEDs, and the reset will be provided by a SPDT switch. (90% of exercise). If you are unable to get a 3-bit counter working, you can implement a 2-bit binary up/down counter that is worth 60%.

7.2.5 Lab Work & Demonstration

7.2.5.1 Design the 3-Bit Binary Up/Down Counter

You are now going to design the finite state machine to implement the 3-bit binary up/down counter by hand. You should document each step in the process so that you can reference it later if you have issues with the implementation. Your counter will have an input called “Up” that will dictate the direction of the counter. When Up=1, the counter will increment on each rising edge of clock. When Up=0, the counter will decrement. You should call your output “Count” with the individual bits being called Count(2), Count(1), and Count(0).

Create the State Transition / Output Table

Since a counter traverses its states in a linear pattern, we typically skip the state diagram step and go straight to the state transition / output table. Create your table in the space below. For your counter, you will use state-encoded outputs. This means you will encode the states in binary and allow the states themselves to be the outputs of the counter. This will allow you to assign the state variables in the initial version of the table. Label your current state variables Q2_cur, Q1_cur, and Q0_cur. Label your next state variables Q2_nxt, Q1_nxt, and Q0_nxt. Remember that you will need to list your input Up in the table as it will dictate the next state for each row.

Derive the Logic Expression for Q2_{nxt}

Remember that each next state variable (Q2_{nxt} in this step) will depend on the current state variables (Q2_{cur}, Q1_{cur}, Q0_{cur}) and the input Up. This means you will use a 4-input K-map to derive this logic expression.

Derive the Logic Expression for Q1_{nxt}

Derive the Logic Expression for Q0_{next}

Derive the Logic Expressions for the Output signals Count

Since you will be using state encoded outputs, the logic expression for each output (Count(2), Count(1), and Count(0)) is simply setting it equal to their respective current state variable (Q2_{cur}, Q1_{cur}, Q0_{cur}). Despite this being a simple set of logic expressions, it is good design practice to document all logic in your system. Enter the logic expressions for the three output bits below.

Draw the Final Logic Diagram

Now draw the final logic diagram. You will need 3x D-flip-flops to hold the 3-bit state codes. You can simply list the input variable names for your next state logic circuits (instead of physically drawing in all of the connections). This will make the diagram more readable. Save an image of your logic diagram. If you drew your diagram manually, you can take a picture and as a JPG. If you created your logic diagram electronically, save it as a JPG. **This image satisfies the requirements for deliverable #1.**

7.2.5.2 Implement the 3-Bit Binary Up/Down Counter

Now implement the logic diagram you designed on your breadboard using discrete parts. Your implementation should follow these guidelines:

- Your breadboard will be provided power and ground from the Analog Discovery. Connect the Analog Discovery's power and GND to the power rails of your breadboard. When you are ready to test your design, you will need to configure the power supply in Waveforms to output +3.4 v and enable.
- Your counter will receive its clock from the AWG of the Analog Discovery. You should connect channel 1 (W1) of the Analog Discovery's AWG to the clock inputs of your counter. Remember that all three D-flip-flops need to receive the same clock. When you are ready to test your design, you will need to configure the AWG to output a 4 Hz square wave with an amplitude of +1.7 v and an offset of +1.7 v.
- Your "Up" input will come from one of the slider switches in your LED driver circuit. You can access the switch output by adding a wire to the node where one of the paths connects to the 10 k Ω pull-down resistor. The Up is used in your next state logic circuits.
- You will need to create a reset line for your D-flip-flops using your push button SPDT switch. You do not need to debounce the switch. You will simply have it drive a logic 1 to all resets in your counter (i.e., reset is deasserted) when the switch is *not* pressed. When the switch *is* pressed, it should drive a logic 0 to all resets in your counter (i.e., reset is asserted) to put the FSM into its reset state.
- You will drive the 3-bit output of your FSM (Count) to three discrete red LEDs. Each of the LEDs will be connected in series with a 150 Ω resistor. The cathode of the LED should be connected to ground.

Take a short video (<5 s) showing the proper operation of your counter. You should show that you can reset the FSM and also change the direction of the counter output. **This video satisfies the requirements for deliverable #2.**

CONCEPT CHECK

Lab 7.2 After completing this lab exercise, can you:

- Design and implement a 3-bit binary up/down counter using discrete parts?
- Use a push button circuit to provide the reset to your counter?

Lab 7.3: 4-Bit Binary Up/Down Counter Finite State Machine (in VHDL)

7.3.1 Objective

The objective of this lab is to gain experience designing finite state machines using VHDL. The state machine will implement a 4-bit binary, up/down counter using a combination of structural VHDL and concurrent signal assignments. The counter will be implemented on an FPGA where its output will be displayed on LEDs in addition to serving as inputs to the 7-segment decoder and prime number detector from prior labs.

7.3.2 Learning Outcomes

After completing this lab you should be able to:

- Design a finite state machine using a combination of structural VHDL for the state memory and concurrent signal assignments for the next state and output logic.
- Instantiate a pre-designed VHDL file into a Quartus project to be instantiated as a component (dfflipflop.vhd).

7.3.3 Parts Needed

- Breadboard + wires.
- DE0-CV FPGA board.
- Analog Discovery.
- 7-Segment display circuit from prior lab (7-segment display, 7x 150 Ω resistors).
- LED driver circuit from prior lab (8-position slider switch, 10 k Ω resistor network, 330 k Ω resistor network, 5x red LEDs, 1x 74HC04 inverter IC).
- Buzzer circuit from prior lab (magnetic buzzer, 2N3904 NPN transistor, 1N4002 diode, 10 k Ω axial resistor).
- 10x female-to-female jumper wires.

7.3.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a finite state machine implemented in VHDL to create a 4-bit, binary, up/down counter. The counter will be displayed on red LEDs of the DE0-CV board in addition to drive the inputs to a 7-segment display decoder + 4-input prime number detector (90% of exercise).
2. Provide your top.vhd design file (10% of exercise).

7.3.5 Lab Work & Demonstration

7.3.5.1 Implement the FSM for the 4-Bit Binary Up/Down Counter

You are going to design a Finite State Machine (FSM) that produces a 4-bit, binary up/down counter in VHDL. You will be provided a VHDL model for a D-Flip-Flop (dfflipflop.vhd) that you will instantiate in your design for the state memory of the state machine. You will create the next state and output logic using conditional signal assignments. The FSM will have an input called *Up* that will come from SW0 on the DE0-CV board. The clock for the FSM will come from the AWG of the Analog Discovery, which will be connected to a pin on the GPIO_1 connector. The reset for the FSM will come from the KEY_4 push button the DE0-CV board. The 4-bit counter output will be displayed on the 4x red LEDs on the DE0-CV board (LEDR3, LEDR2, LEDR1, and LEDR3). Internal to the FPGA, the 4-bit counter will drive the pre-designed 7-segment decoder logic and prime number detector from lab 6.1. As in lab 6.1, the outputs of the 7-segment decoder interfaced to your breadboard where they will drive the 7-segment character display and the prime number detector output will drive an LED, and the buzzer. [Figure 7.9](#) shows a block diagram of the FSM system.

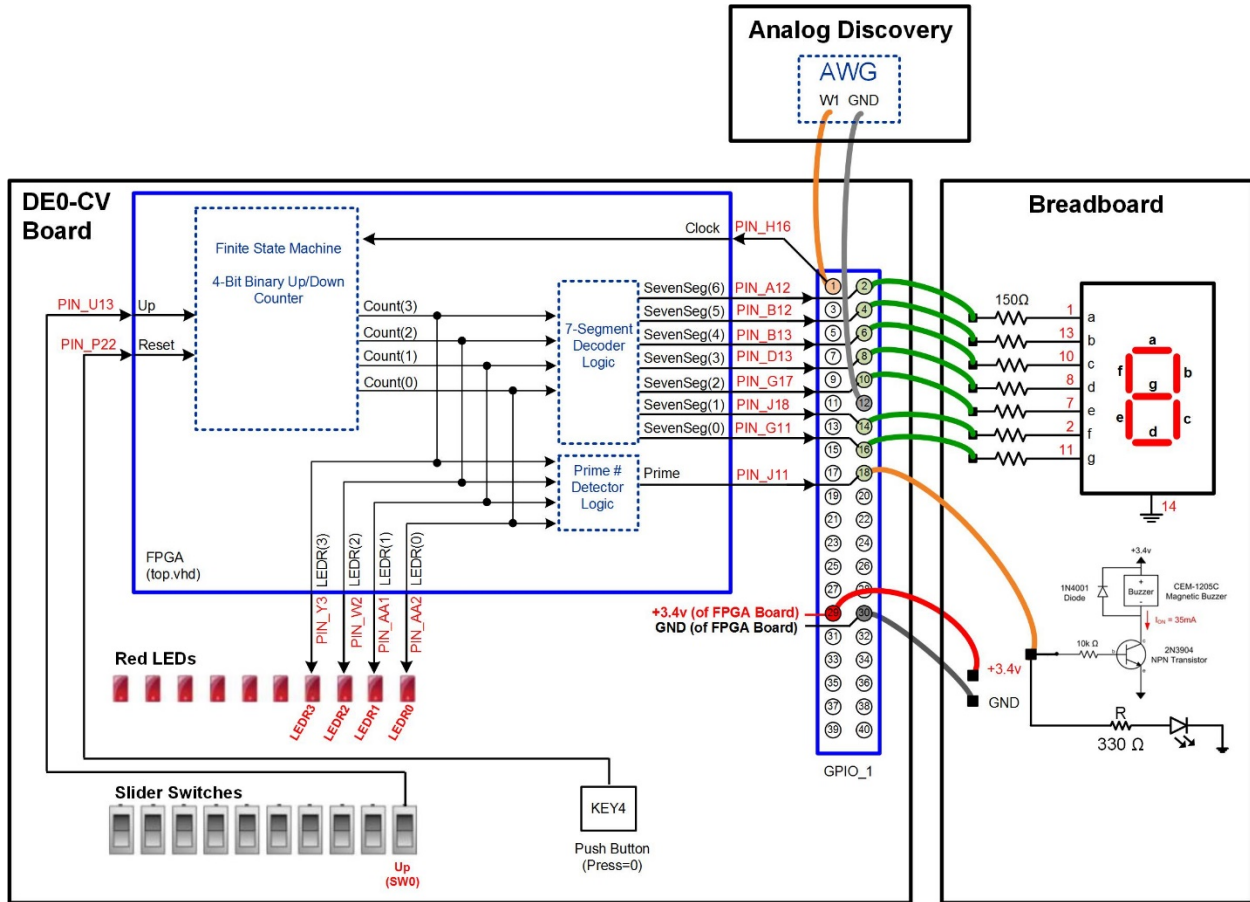


Figure 7.9
Block Diagram of the 4-Bit Binary, Up/Down Counter System

Figure 7.10 shows a picture of the FSM system.

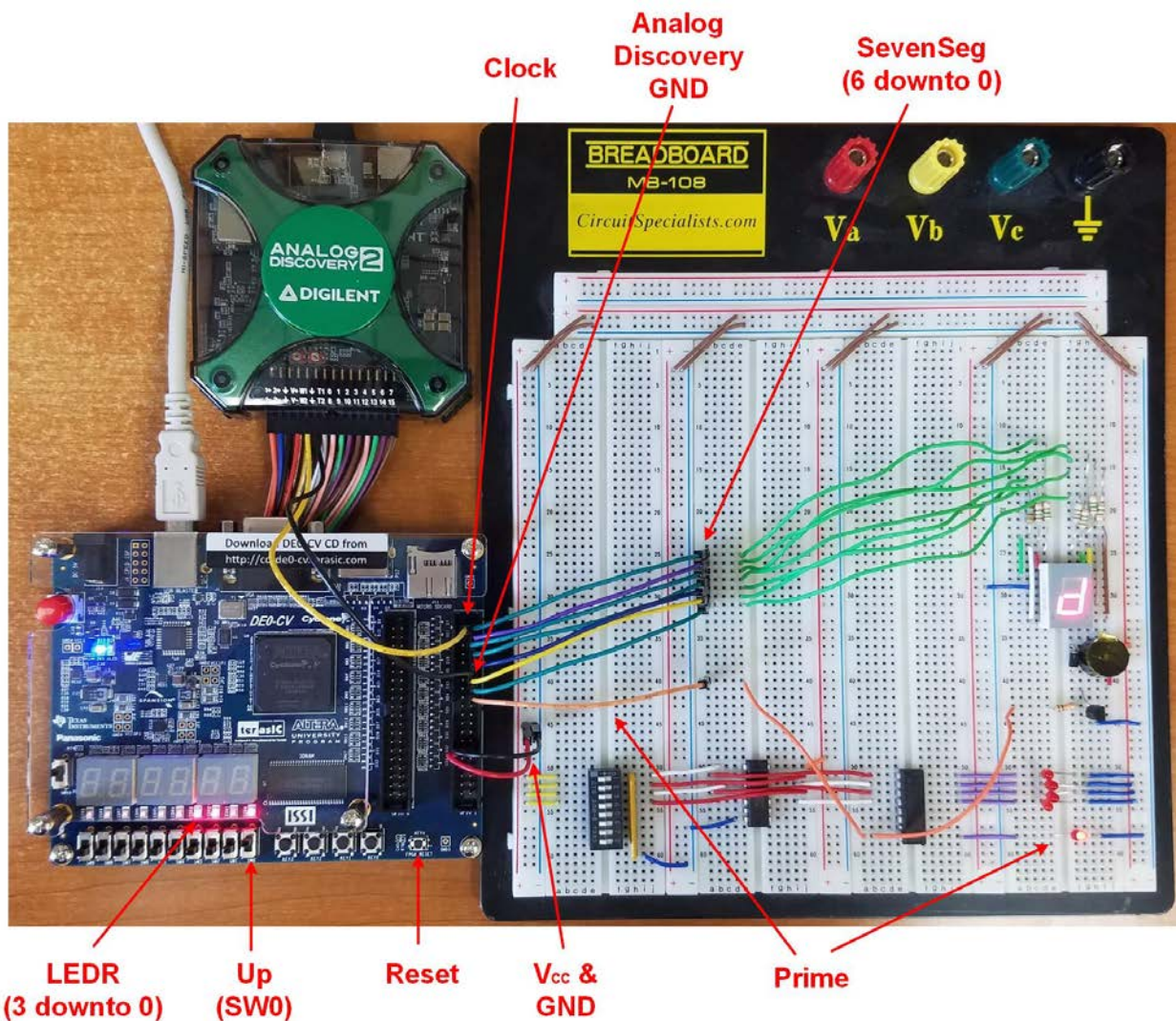


Figure 7.10
Picture of the 4-Bit Binary, Up/Down Counter System on the DE0-CV Board + Breadboard Interface

Breadboard the FSM System

Breadboard the system in [Figure 7.9](#) and [Figure 7.10](#). The power for the breadboard will come from the DE0-CV board; however, at this point don't turn on the DE-CV or enable any of the Analog Discovery tools.

Design the State Transition / Output Table for the Counter

The first step in this design is to document the desired behavior of the 4-bit counter. Since counters traverse their states linearly, we can skip the state diagram step of the design and move directly to the state transition table. The state transition table will be very important for this lab because you will be implementing the next state logic and output logic circuitry in VHDL using conditional signal assignments and the functionality can be typed directly in from the table. A template for the state transition table is provided in [Figure 7.11](#). The first two rows are filled in to help get you started. Fill in the remaining boxes for the counter.

	Current State Q_cur(3 downto 0)				In Up	Next State Q_nxt(3 downto 0)				Output Count(3 downto 0)				
	(3)	(2)	(1)	(0)		(3)	(2)	(1)	(0)	(3)	(2)	(1)	(0)	
S0	0	0	0	0	0	S15	1	1	1	1	0	0	0	0
	0	0	0	0	1	S1	0	0	0	1	0	0	0	0
S1						S0								
S2						S2								
						S1								
S3						S3								
						S2								
S4						S4								
						S3								
S5						S5								
						S4								
S6						S6								
						S5								
S7						S7								
						S6								
S8						S8								
						S7								
S9						S9								
						S8								
S10						S10								
						S9								
S11						S11								
						S10								
S12						S12								
						S11								
S13						S13								
						S12								
S14						S14								
						S13								
S15						S15								
						S14								
						S0								

Figure 7.11
State Transition / Output Table for the 4-Bit Counter FSM

[Create a New Quartus Project by Copying Lab 6.1](#)

We are now going to create a new Quartus project for this design. We want to re-use the 7-segment decoder and prime number detector logic from lab 6.1, so we will create this new project using the *Copy Project* feature in Quartus. Open Quartus. Next, open the lab 6.1 project. Next, copy the project to a new folder using the file pull-down menus:

Project – Copy Project. You should name the folder and project “Lab_07p3_4bit_Cnt_FSM”. After clicking “OK”, make sure to verify that you want to create the new folder for this project.

Add the dflipflop.vhd File to your Project

A VHDL model for a rising edge D-flip-flop has been provided for you. You will want to include this in your project and the instantiate it to implement the state memory of your FSM. Download the dflipflop.vhd file and place it in your project directory. It should be in the same directory as your top.vhd.

Use the pull-down menus to add this file to your project: Project → Add / Remove Files in Project. The *Settings* window shows in [Figure 7.12](#) will appear.

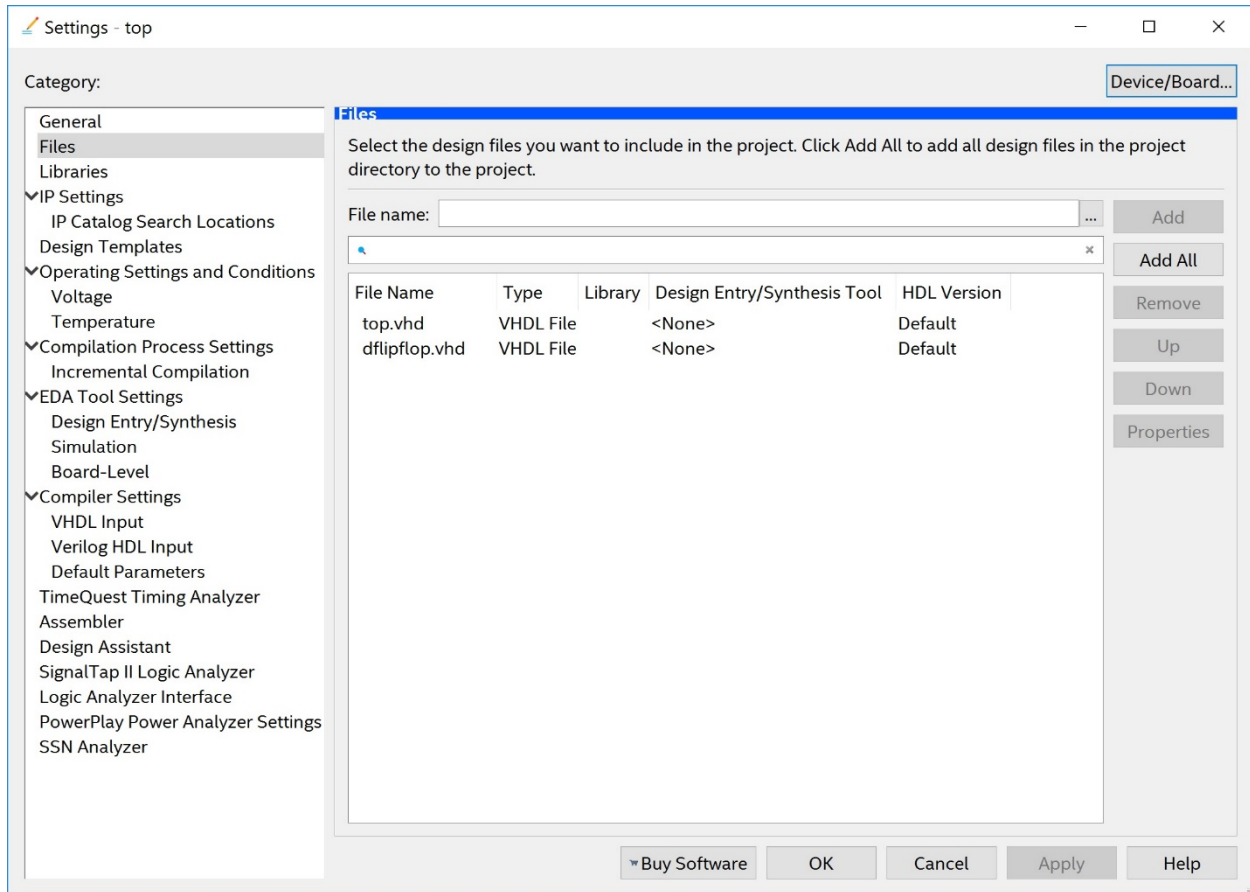


Figure 7.12
Adding Files to a Project in Quartus

Click on the browse button (the “...”) and browse to the dflipflop.vhd file in your project directory. Select the file and press “Open”. The dflipflop.vhd file will be added to your project. You need to move it down so that top.vhd is above it in the list, indicating that top is still the highest level in the hierarchy. Highlight the dflipflop.vhd file, and press the “Down” button the right. Once your settings are identical to [Figure 7.12](#), click “OK”.

Update the VHDL Entity for the 4-Bit Counter System

In Quartus, double click on “top” to open top.vhd in an editor. We need to update the entity to add and modify the ports from lab 6.1 to match the block diagram in [Figure 7.9](#). We need to add three signals to our entity (Clock, Reset, and Up) and remove the SW signal from lab 6.1 We will still use the ports LEDR, Prime, and SevenSeg. When complete, your entity should look like [Figure 7.13](#).

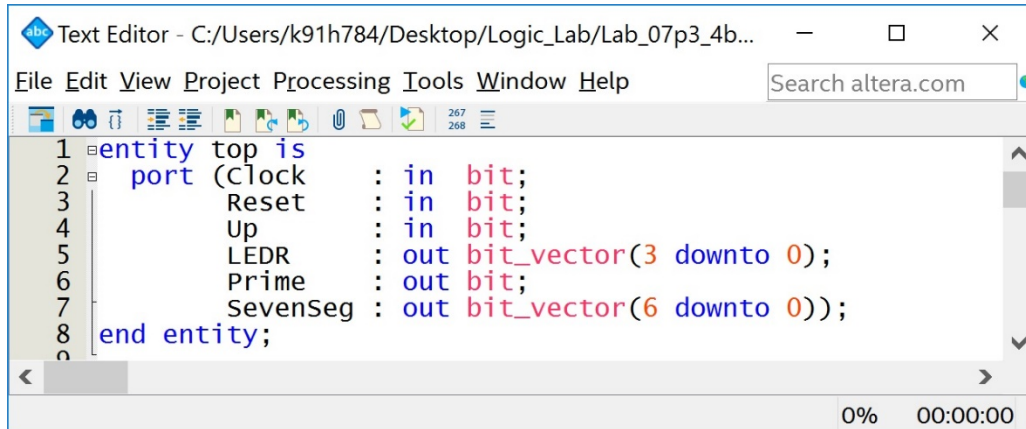


Figure 7.13
The VHDL Entity for the 4-Bit Counter FSM

[Declare the D-flip-flop as a Component in your Architecture](#)

In order to instantiate the D-flip-flop model you added, you need to first declare it in the architecture as a component. Recall that component declarations go before the *begin* statement in the architecture. Your VHDL component declaration should look like:

```

component dflipflop
  port (Clock      : in  bit;
        Reset      : in  bit;
        D          : in  bit;
        Q, Qn      : out bit);
end component;

```

[Declare the Next State and Output Logic Variables in your Architecture](#)

Next, you need to declare signals to hold the current state, next state, and the output variables. These will all be 4-bit vectors. We'll call the current state variable *Q_cur* and the next state variable *Q_nxt*. The individual bits of these vectors will be wired to the D-flip-flops components when instantiated. We should also create a vector for the Qn output of the D-flip-flop called *Qn_cur*. We will create a 4-bit vector for the FSM output called *Count*. Recall that signal declarations also go before the *begin* statement in the architecture. Your VHDL signal declaration should look like:

```

signal Q_nxt, Q_cur, Qn_cur : bit_vector (3 downto 0); -- FSM State Variables
signal Count                : bit_vector (3 downto 0); -- FSM Outputs

```

[Instantiate the D-flip-flops to Implement the State Memory of the FSM](#)

You are now ready to start implementing the behavior of your state machine. The first part of the state machine is the state memory. Recall that state memory is simply a set of D-flip-flops that hold the current state variables on their Q outputs. The D-flip-flops receive the next state codes on their D inputs. Since a FSM is synchronous, all D-flip-flops receive the same clock and reset. To implement a 4-bit counter, we will have 16 states and require 4x D-flip-flops. Each D-flip-flop will be port mapped to a corresponding bits of the current and next state vectors. You should instantiate four versions of the D-flip-flop as follows:

```

DFF0 : Dflipflop port map (Clock, Reset, Q_nxt(0), Q_cur(0), Qn_cur(0));
DFF1 : Dflipflop port map (Clock, Reset, Q_nxt(1), Q_cur(1), Qn_cur(1));
DFF2 : Dflipflop port map (Clock, Reset, Q_nxt(2), Q_cur(2), Qn_cur(2));
DFF3 : Dflipflop port map (Clock, Reset, Q_nxt(3), Q_cur(3), Qn_cur(3));

```

[Design the Next State Logic of the FSM](#)

Next, you need to implement the next state logic for the FSM. We will do this using a conditional signal assignment. One of the advantages of using vectors for *Q_nxt* is that assignments can be made to it 4-bits at a time. This greatly simplifies the amount of VHDL that needs to be written to create the 4x next state logic circuits. The next state logic

depends on both the current state (`Q_cur`) and the input (`Up`). A conditional signal assignment makes handling the input variables straightforward as a Boolean *and* condition can be used. Consider the following approach to modeling the next state logic for this FSM. Notice how both `Q_cur` and `Up` are considered in the assignment.

```
Q_nxt <= "0001" when (Q_cur="0000" and UP='1') else
         "0010" when (Q_cur="0001" and UP='1') else
         "0011" when (Q_cur="0010" and UP='1') else -- rest of the logic goes below
```

Since there are 5 bits of inputs in this statement (4x for `Q_cur` and 1x for `Up`), your full conditional signal assignment will have 32 separate conditions. These conditions can be entered directly from the table in [Figure 7.11](#). Complete the VHDL model for your next state logic.

[Design the Output Logic of the FSM](#)

In this FSM, we are using state-encoded outputs. This means we have encoded the states in a form that matches the counting pattern we desire (i.e., binary) and the code for each state will be the actual output. To model this behavior, we can create a single signal assignment as follows:

```
Count <= Q_cur;
```

At this point your FSM model is complete. You have created the three portions of the FSM using a combination of structural design (i.e., instantiating D-flip-flops for the state memory) and concurrent signal assignments (i.e., the assignments for the next state and output logic).

[Update the Prime Number Detector Model to use Count as its Input](#)

In the prime number detector model that was copied from lab 6.1, the signal assignments were based on the input `SW`. In this lab, we want the prime number detector to base its outputs on `Count`. Update the VHDL model to reflect this change.

[Update the 7-Segment Decoder Model to use Count as its Input](#)

In the 7-segment display decoder model that was copied from lab 6.1, the signal assignments were based on the input `SW`. In this lab, we want the 7-segment decoder to base its outputs on `Count`. Update the VHDL model to reflect this change.

[Compile the Design and Fix any Syntax Errors](#)

Compile and synthesize your design by double clicking on “Compile All”. Make sure to save your top.vhd file. Note that we have not assigned the pins for the new ports in this design. We will first do a compile and then the pin planner will reflect the new ports.

[Assign the Pins for this Design](#)

Launch the pin planner tool using the drop down menus: Assignments → Pin Planner. The first thing to do is to delete the rows for `SW`. Highlight the rows for `SW` and press “Delete” on your keyboard. You will be asked to confirm the operation for each row.

Next, you need to assign the locations for all of the new ports in this design. There are three new ports that need pin assignments. Make the assignments for the location and associated I/O standard as follows.

- Up PIN_U13 (2.5 V)
- Reset PIN_P22 (2.5 V)
- Clock PIN_H16 (3.3-V LVCMOS)

When complete, your assignments should look like [Figure 7.14](#).

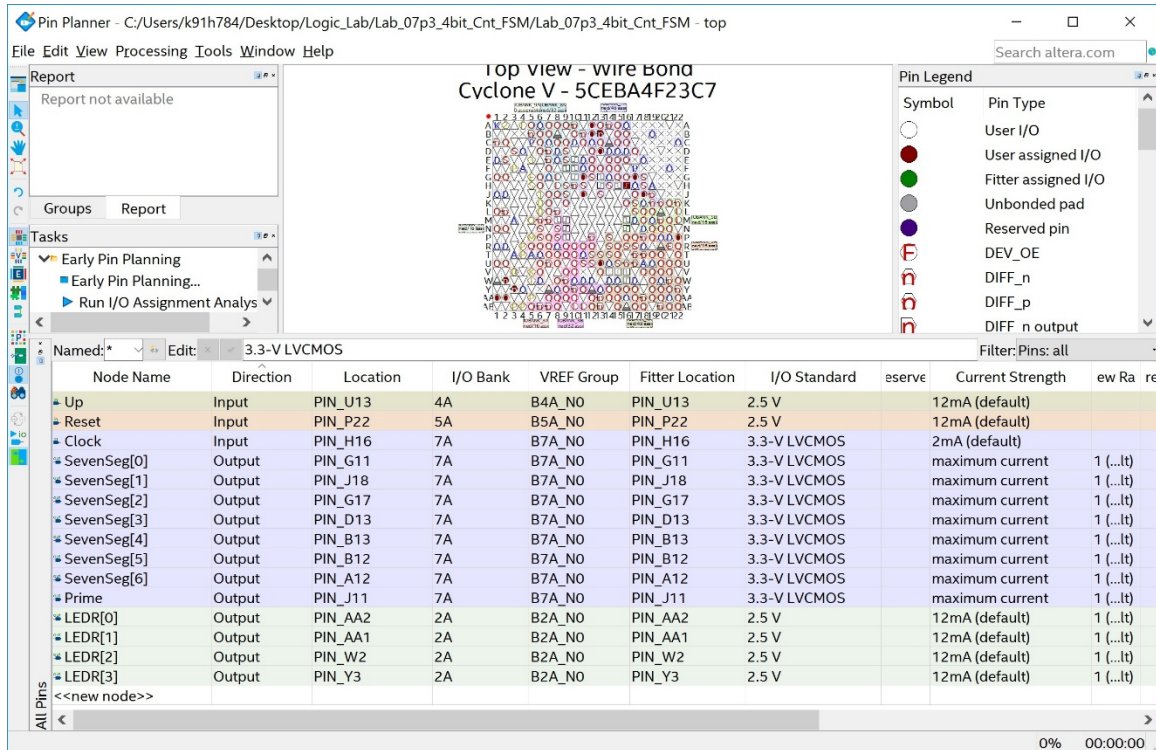


Figure 7.14
Pin Assignments for the 4-Bit Counter FSM

Close the pin planner tool using the pull down menus: File → Close

[Re-Compile the Design and Fix any Syntax Errors](#)

Re-compile and synthesize your design with the new pin assignments by double clicking on “Compile All”. Fix any errors you encounter.

[Program the FPGA with your 4-Bit Counter Design](#)

Now you are ready to download your design to the FPGA. Plug in the USB cable for the DE0-CV board and turn it on using the red power button. Verify that the DE0-CV board is providing power to your breadboard by toggling your slider switches in your LED driver circuit. You should see the corresponding LEDs turn on in your driver circuit.

In Quartus, double click on the “Program Device” task. Click “Start” on the programmer to download your design.

[Turn on the AWG](#)

Your design is ready to accept a clock from the AWG. Launch Waveforms and configure channel 1 of the AWG to output a square wave with the following settings:

- Type = Square
- Frequency = 4 Hz
- Amplitude = 1.7 V
- Offset = 1.7 V
- Symmetry = 50%
- Phase = 0°

Enable the AWG by pressing the “Run” button. Note that we are **not** providing power to the breadboard using the Analog Discovery so no other tools are needed other than the AWG.

Test your 4-Bit Counter Design

At this point you should see your system counting. The binary count will be displayed on the LEDRs of the DE0-CV board and will be updated at a rate of 4 Hz. The corresponding hex symbol will be shown on the 7-segment display. Whenever the code is a prime number, the LED and buzzer on the breadboard should assert. You should be able to change the direction of the counting pattern by toggling SW0 on the DE0-CV. You should also be able to reset the counter to 0 by pressing KEY_4 on the DE0-CV. Take a short video (<5 s) showing the proper operation of your FSM system. **This video satisfies the requirements for deliverable #1.**

7.3.5.2 Save a Copy of your top.vhd for your Records

Save a copy of your top.vhd file for the second deliverable of this exercise. Recall that this file is located in your main working directory. Go into your working directory for this project and locate this file. **This file satisfies the requirements for deliverable #2.**

When you are done, close your Quartus project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit. Exit Waveforms. Turn off your DE0-CV board.

CONCEPT CHECK

Lab 7.3 After completing this lab exercise, can you:

- Add a pre-existing VHDL file to a Quartus project to be used as a component?
- Implement the state memory of a FSM structurally by instantiating individual D-flip-flops?
- Implement the next state and output logic of a FSM using continuous signal assignments?

Chapter 8: VHDL (part 2)

Lab 8.1: 7-Segment Display Decoder using a Process

8.1.1 Objective

The objective of this lab is to begin modeling combinational logic using a VHDL process and conditional programming constructs. You will create the decoder logic in VHDL to drive one of the 7-segment displays on the DE0-CV board (HEX0).

Note: As this lab exercise is commonly used as the starting point for the 2nd course in the logic design sequence, the steps to create a Quartus project from scratch will be repeated. These are the same steps that were covered in lab 5.1. Also of note is that all lab exercises from this point on will only use the DE0-CV board and the Analog Discovery.

8.1.2 Learning Outcomes

After completing this lab you should be able to:

- Use the Quartus toolchain to synthesize, technology map, place/route and implement a VHDL model on an FPGA.
- Create a 4-input, 7-segment display decoder (0_{16} to F_{16}) on an FPGA using a VHDL process and conditional programming constructs.
- Use the *std_logic_vector* data type from the STD_LOGIC_1164 library.

8.1.3 Parts Needed

- DE0-CV FPGA board.

8.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a VHDL design on an FPGA that drives the HEX0 7-segment display on the DE0-CV board using the slider switches (SW3:SW0) as inputs (90% of exercise).
2. Provide your top.vhd design file (10% of exercise).

8.1.5 Lab Work & Demonstration

8.1.5.1 Implement the 7-Segment Decoder in VHDL on the DE0-CV Board

You are going to design a 7-segment display decoder in VHDL using a process and a conditional programming construct (*if/then* or *case*). The decoder will take in the values from the slider switches on the DE0-CV board (SW3, SW2, SW1, and SW0) and display the corresponding HEX characters (0_{16} to F_{16}) on one of the 7-segment displays (HEX0). You will also drive the values of the slider switches on the red LEDs on the DE0-CV (LEDR). [Figure 8.1](#) shows the block diagram for the system you will design. The pin numbers for each of the I/O used in this lab are shown in red. Note that you will be creating your VHDL ports as vectors in order to simplify your model.

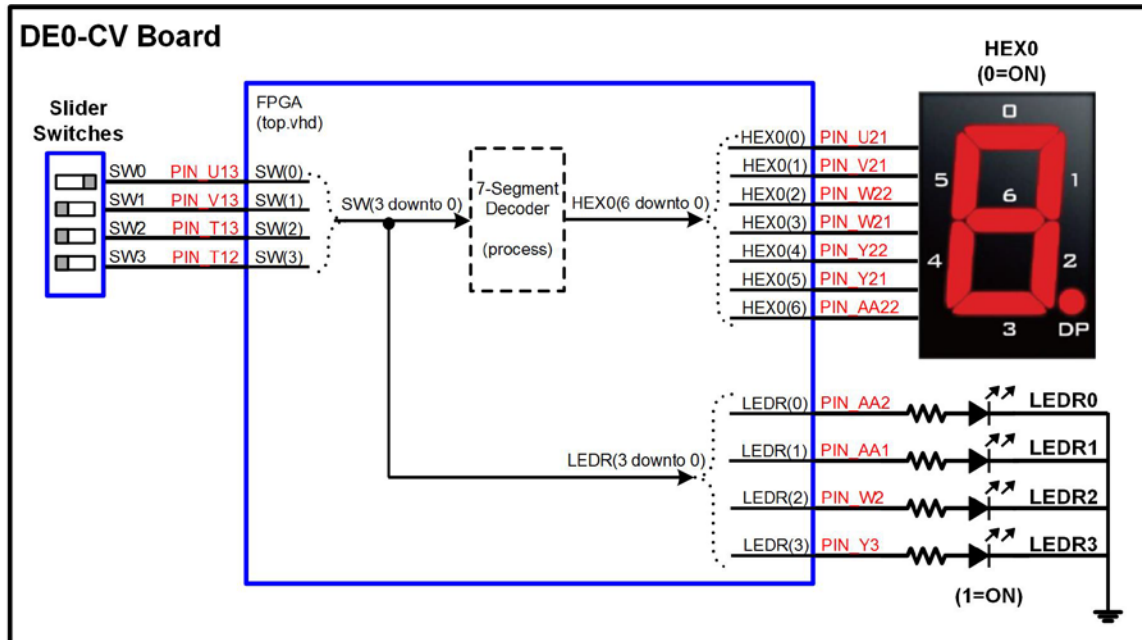


Figure 8.1
Block Diagram of the 7-Segment Decoder System

Figure 8.2 shows the I/O on the DE0-CV that you will be using.

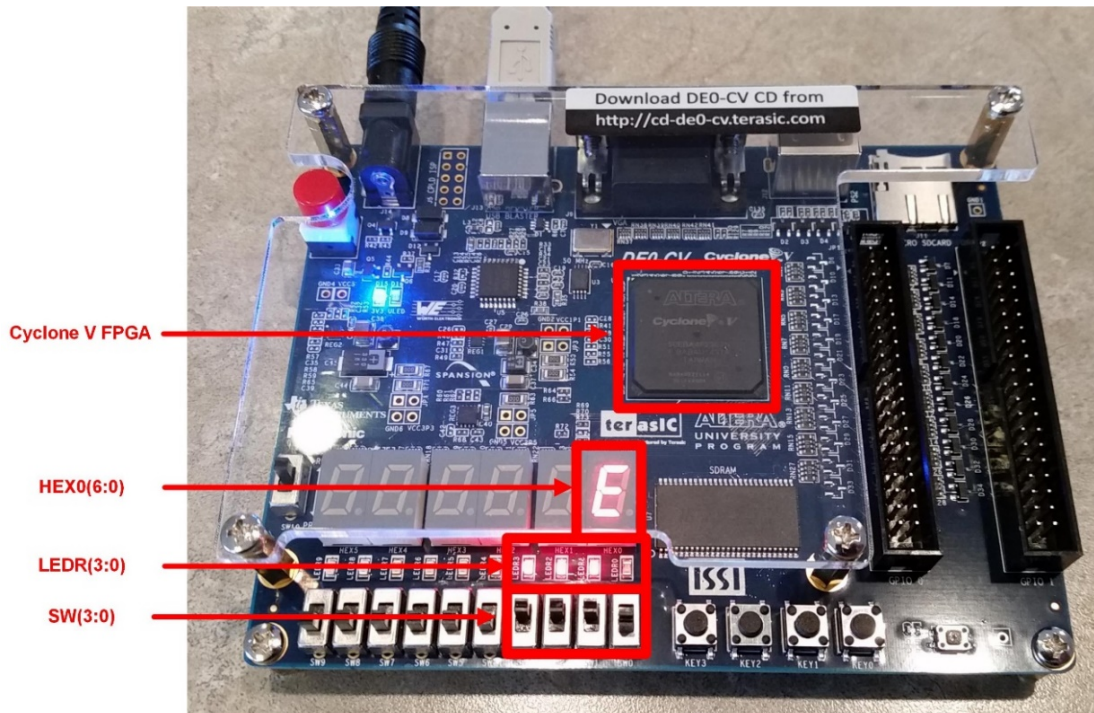


Figure 8.2
Picture of the 7-Segment Decoder System on the DE0-CV Board

Determine the Logic Functionality for the HEX0 Decoder

Before you can model the decoder in VHDL, you need to determine the functionality you wish to model. You will need to come up with the values to drive to the character display in order to generate the appropriate symbols for each input code. Tabulate your logic using Figure 8.3. The character displays on the DE0-CV board use a *common anode* configuration. This means to turn on an LED in the display, you need to drive it with a logic 0. Also note that we will be using the naming conventions found in the DE0-CV user's manual. In the manual the individual LEDs of the HEX displays are labeled 0, 1, 2, 3, 4, 5, and 6. We will be defining a 6-bit VHDL port called *HEX0* that will drive these individual bits. To make the signal conventions consistent, we will map bit 0 of the HEX0 vector to LED 0 on the display and bit 6 of the vector to LED 6.

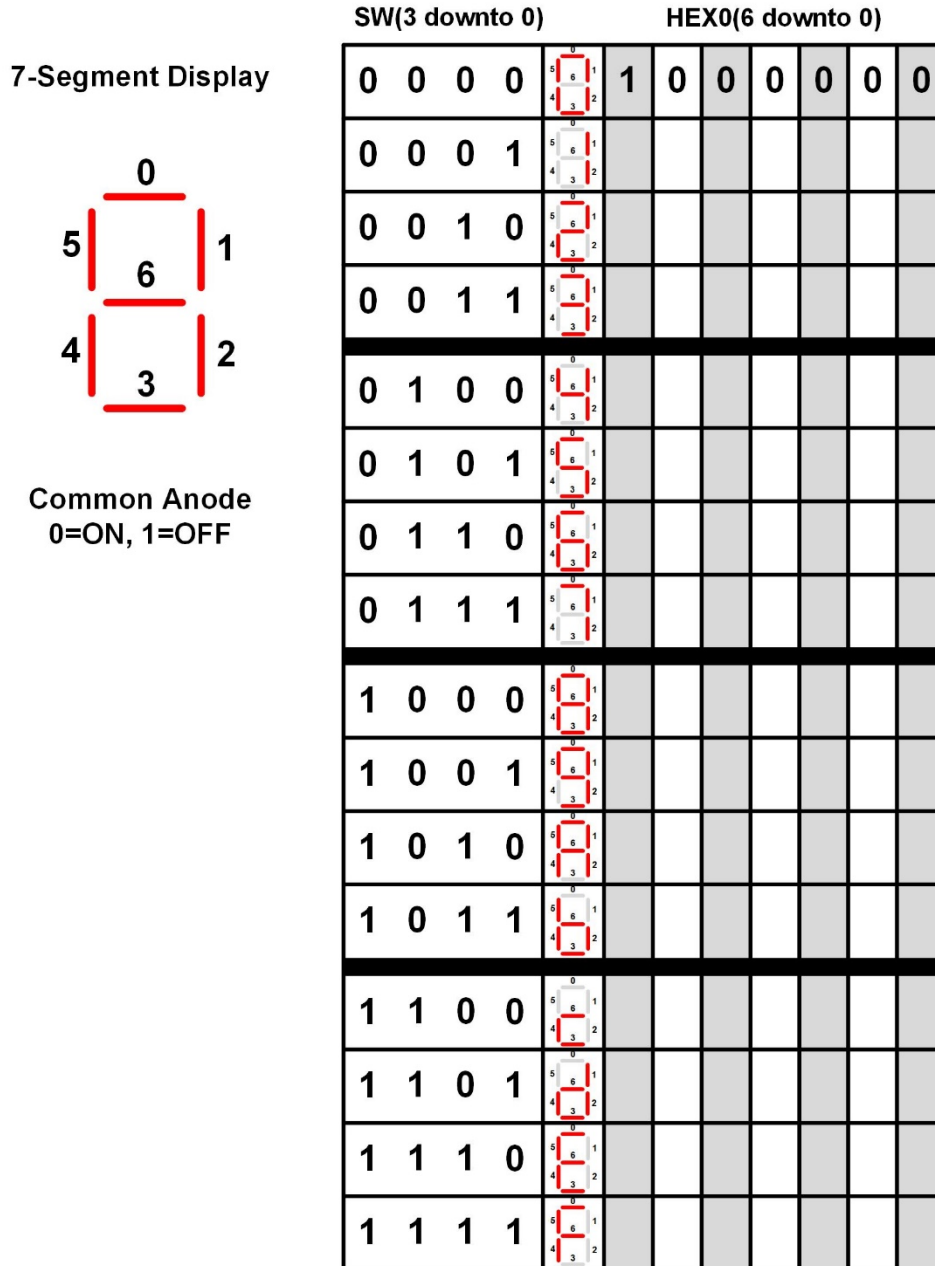


Figure 8.3
Table for HEX0 Display Logic

[Install the Quartus Lite Software \(if not already installed on your computer\)](#)

The Quartus Lite software can be downloaded for free from www.altera.com. This will take you to an *intel* website, which purchased Altera Inc. in 2016. Once on this page, click on “Support” and then “Downloads”. You will need to create a free account. There you’ll be given the option of downloading various versions of Quartus. You want to download the “Lite” version. Click on the “Download” icon next to the Lite version. In the next screen you’ll be given options on what all should be downloaded. You want to select three items:

- Select edition: Lite
- Select release: “latest release, i.e., the highest number”
- Operating System: windows (assuming you are on windows)
Download Method: Direct download

Then click on the “Individual Files” tab. Select the following files to download:

- Quartus Prime (includes Nios II EDS)
- ModelSim-Intel FPGA Edition (includes Starter Edition)
- Cyclone V device support

Then click “Download Selected Files”. It will ask you where to download the files. Choose your Desktop and select “OK”. It will then proceed to download three files. The files are large so it will take a while. Once they are downloaded to your desktop, double click on the Quartus install file (named something similar to “QuartusLiteSetup-17.0.0.595-windows.exe”). This will automatically install the ModelSim software and the Cyclone V drivers that you downloaded as long as they are in the same location (i.e., the Desktop). Accept the defaults for all options in the install.

When complete, it will ask if you want to install desktop icons, launch Quartus, and/or launch the driver install tool. Select the option to launch the driver install tool. Follow the instructions to install the drivers for the DE0-CV board. When you plug in the DE0-CV board in the next few steps, the driver installation will complete.

[Create a Folder for to Hold All of your VHDL Projects \(if not already created\)](#)

For each Quartus project, you will manually create a folder and then direct Quartus to put all design files into that directory. We want to first create a main folder that will hold all of the project folders that will be created throughout this manual to help us stay organized. We will create a folder on the Desktop called “Logic_Lab”. Right click on the Desktop of your computer and select “New → Folder”. Give it the name “Logic_Lab”.

[Create a Folder for this Exercise](#)

Now we need to create the folder that will contain all of the Quartus files for this project. We want to name this project something descriptive. Browse to your Logic_Lab folder that you created in the prior section and manually create a folder called “Lab_08p1_7segment_decoder_using_process”.

[Launch Quartus](#)

On a windows 10 or equivalent machine, the Quartus application can be launched at: Start → Intel FPGA *version* Lite Edition → Quartus (Quartus Prime *version*). [Figure 8.4](#) shows the Quartus startup window that will appear.

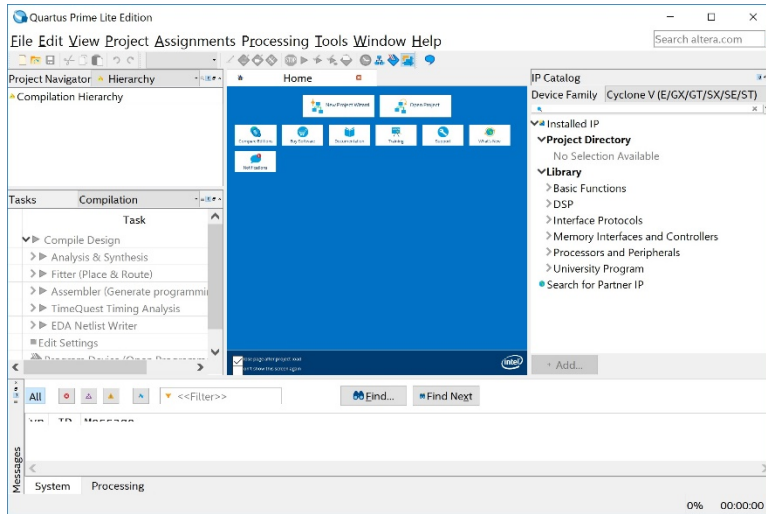


Figure 8.4
Quartus Startup Window

[Create a New Project using the “New Project Wizard”](#)

Click on the “New Project Wizard” button in the *Home* pane of the Quartus window. The *Introduction* window shown in [Figure 8.5](#) will appear.

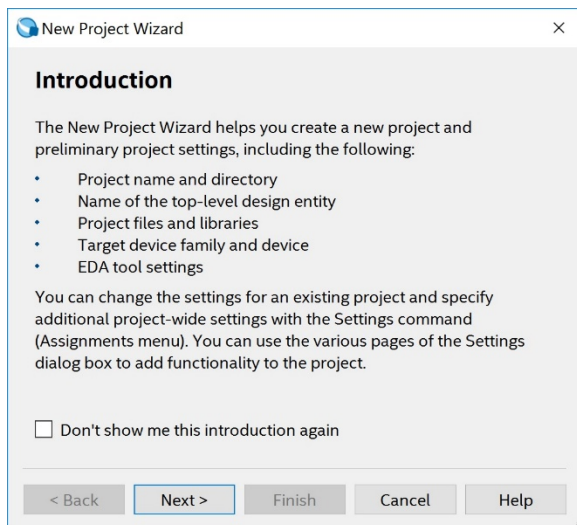


Figure 8.5
Quartus New Project Wizard (Introduction)

Click on “Next” to go the next window. If you don’t want the introduction window to show up next time you run New Project Window, you can check the box. The *Directory, Name, Top-Level Entity* window shown in [Figure 8.6](#) will appear.

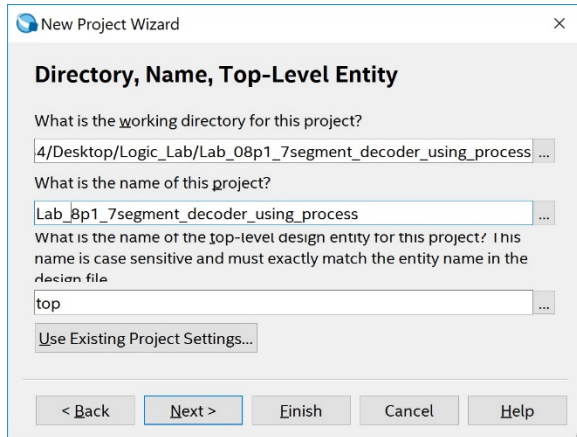


Figure 8.6
Quartus New Project Wizard (Directory, Name, Top-Level Entity)

For the working directory, click on the “...” button and browse to the “Logic_Lab/Lab_08p1_7segment_decoder_using_process” folder you created on your desktop. This project folder will contain all of the Quartus design files.

For the name of the project, enter “Lab_08p1_7segment_decoder_using_process”. Note that as you type this name, Quartus automatically enters the same name for the top-level entity. **You do not want to use this name as your top-level entity.**

For the top-level design entity, enter “top”. The term *top* is the standard naming convention for the VHDL file that is at the highest level of hierarchy in the system. At the top level, the ports of the entity will be the physical pins of the device. For this exercise, we will only have one top-level file. We will create this file (*top.vhd*) in a later step.

Click on the “Next” button. The *Project Type* window shown in [Figure 8.7](#) will appear.

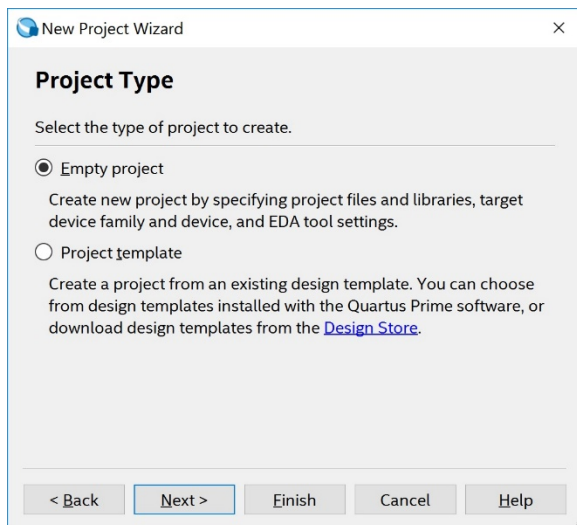


Figure 8.7
Quartus New Project Wizard (Project Type)

Leave the project type as “Empty” and click “Next”. The *Add Files* window shown in [Figure 8.8](#) will appear.

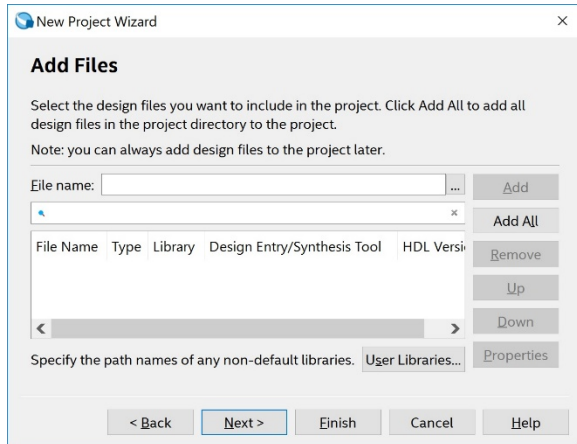


Figure 8.8
Quartus New Project Wizard (Add Files)

This window is where we can add existing design files. For this exercise we do not have any existing design files. Instead, we will be creating a new file called `top.vhd`. We will do this later outside of the New Project Wizard. Do not do anything in this window except click “Next”. The *Family, Device & Board Settings* window shown in Figure 8.9 will appear.

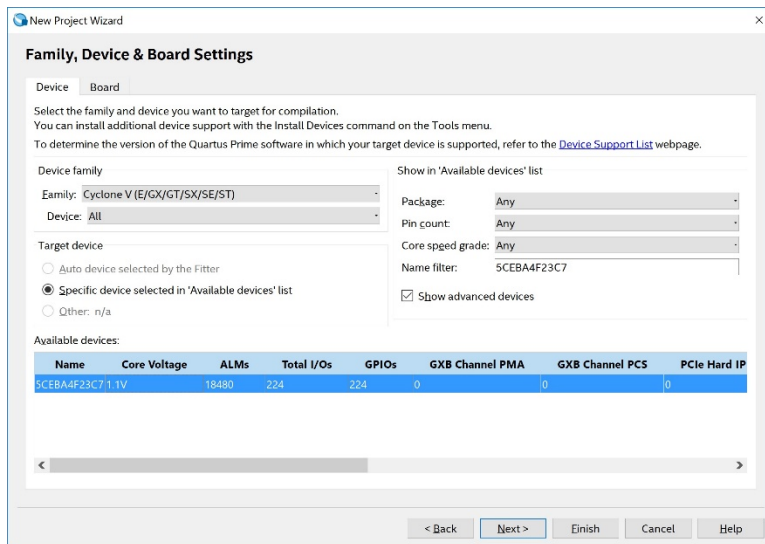


Figure 8.9
Quartus New Project Wizard (Family, Device & Board Settings)

In this window we tell the Quartus synthesizer which device we will be targeting. Since we only installed the Cyclone V FPGA family, only those devices will be shown. Notice that there are many different Cyclone V devices that can be selected. We want to select the FPGA device that is on our DE0-CV board. We can begin filtering down the selection by typing in the “Name filter” field on the right side of the pane. Begin typing `5CEBA4F23C7`. After each character is typed, it will reduce the number of devices that are available. Once you type the last character, there will only be one device available. Highlight this device and click “Next”. The *EDA Tool Settings* window shown in Figure 8.10 will appear.

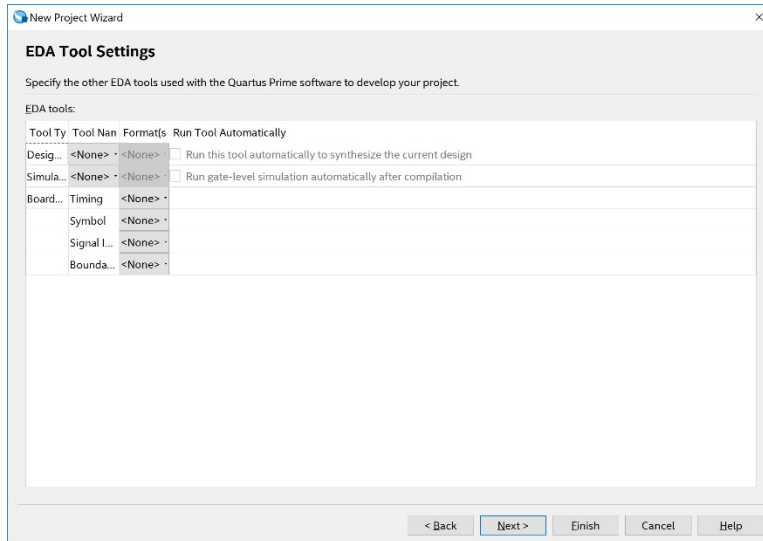


Figure 8.10
Quartus New Project Wizard (EDA Tool Settings)

In this window you can direct Quartus to read files from various CAD tools from other vendors. For this exercise, we will be using the Quartus tool by itself. Leave all the settings at *<None>* and click “Next”. The *Summary* window shown in [Figure 8.11](#) will appear.

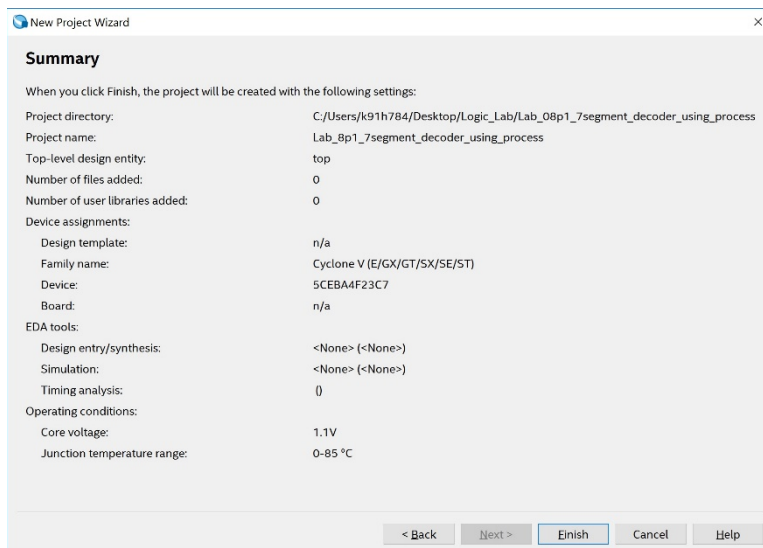


Figure 8.11
Quartus New Project Wizard (Summary)

Review the settings in this window and if correct, click “Finish”. If any of the settings are incorrect, you can click the “Back” button to go back and change them. You will now see a new blank project window with all of your settings as shown in [Figure 8.12](#).

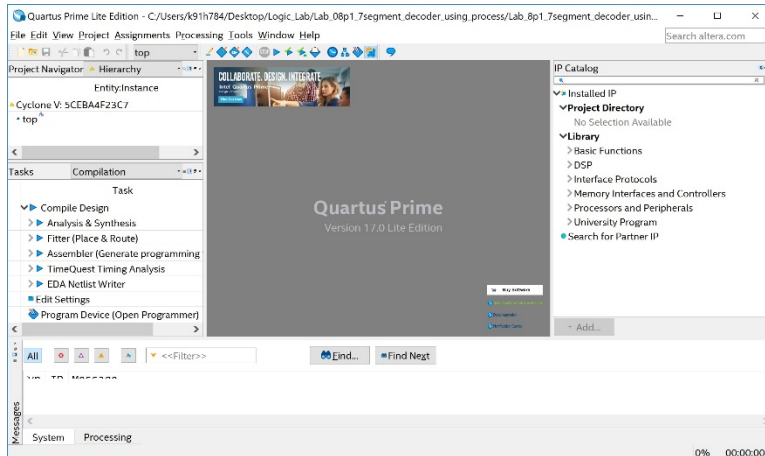


Figure 8.12
New Blank Project for the Cyclone V on the DE0-CV FPGA Board

[Create the top.vhd File for this Project](#)

Now we are ready to create the top.vhd file. In the Quartus window, use the pull-down menus to select: File → New. In the *New* window that appears, select “VHDL File” under the *Design Files* group. Click “OK”.

Now we need to save the file and name it accordingly. In the Quartus window, use the pull-down menus to select: File → Save As. By default, the name of the file should be called top.vhd and be located in your project directory. Verify that these settings are correct (if not, fix them) and click “Save”. The top.vhd is now open for editing. If you close this file it can be reopened by double clicking on it in the *Project Navigator* pane of the Quartus window.

[Add the STD_LOGIC_1164 Library](#)

The first step in the design is to add any **VHDL packages** we will need. From this point forward, we will be using the std_logic and std_logic_vector data types (instead of bit and bit_vector). These data types are found in the std_logic_1164 library. You will always include the syntax to include this library at the top of all of your VHDL files going forward. Add the following VHDL statements to the beginning of your top.vhd file.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

[Enter the VHDL Entity](#)

We are now ready to enter the **VHDL entity** for the design. The entity contains all of the ports for the system. Based on the block diagram provided in [Figure 8.1](#), the ports are:

- **SW (3 downto 0)** This is the 4-bit input vector for the 4x slider switches on the DE0-CV board.
- **LEDR (3 downto 0)** This is the 4-bit output vector for the 4x red LEDs on the DE0-CV board.
- **HEX0 (6 downto 0)** This is the 7-bit output vector that will drive the 7x LEDs within the HEX0 display on the DE0-CV board.

Enter the following VHDL entity into your top.vhd. Notice that the entity name matches the file name in addition to the name of the top-level of the design in Quartus. This tells Quartus that any ports that are declared will be connected to pins on the FPGA board.

```
entity top is
  port (SW : in std_logic_vector (3 downto 0);
        LEDR : out std_logic_vector (3 downto 0);
        HEX0 : out std_logic_vector (6 downto 0));
end entity;
```

At this point your project should look like [Figure 8.13](#). Notice that Quartus recognizes the VHDL syntax and will color code based on the construct type.

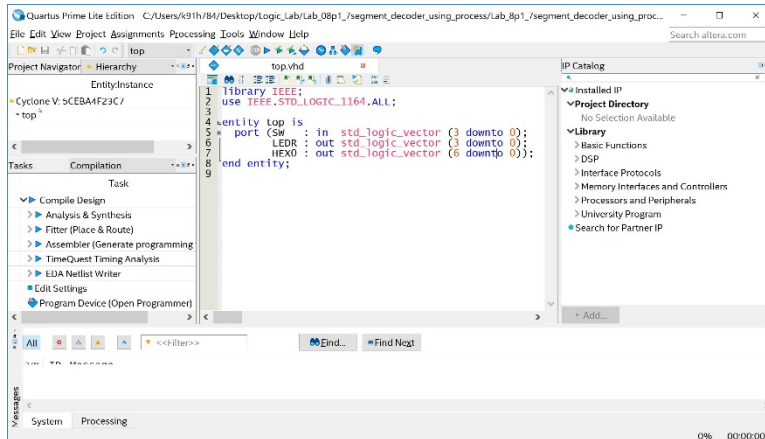


Figure 8.13
VHDL Entity for 7-Segment Display Decoder using a Process

Enter the VHDL Architecture

Now we are ready to enter the **VHDL architecture**. The first part of your architecture should drive the SW inputs to the LEDR outputs. This can be done using a simple signal assignment. This does NOT need a process.

The second part of the architecture is the decoder logic. You should create a process to perform the decoding. The process is implementing combinational logic so the sensitivity list should contain all inputs (i.e., SW). You can use either if/then or case statements within the process to implement the decoder. You will need to include a final assignment in your conditional statement that handles the assignment for ;all of the other possible input codes available in the std_logic data type that were not explicitly listed. Implement the logic from your table in [Figure 8.3](#). Remember that the process is driving the output vector HEX0 depending on the inputs SW.

```
architecture top_arch of top is
begin
    LEDR <= SW;

    -- Decoder process goes here...

end architecture;
```

Save your file using the pull-down menus File → Save.

Compile your Design

Now we are going to compile the top.vhd file and fix any errors. To launch a task in Quartus, you double click on the desired task name within the *Task* pane on the left hand side of the window. Notice that underneath the task “Compile Design” there are a handful of other tasks (i.e., Analysis & Synthesis, Fitter, etc.). When you click on “Compile Design”, Quartus will run all of the lower-level tasks. If it encounters an error, it will stop at the task where the error was found. If the task completes successfully, it will turn green. We typically allow Quartus to run as many tasks as it can when we compile to attempt full synthesis of the design. To launch the compiler and synthesizer, double click on “Compile Design”. This will take longer than a simple syntax check as Quartus is performing synthesis. The status of the task will appear in the *Messages* window at the bottom of the pane. You will also see a status bar for each of the tasks being performed. Once you have fixed any errors that have occurred and successfully completed the Compile Design tasks, you will see the status in [Figure 8.14](#).

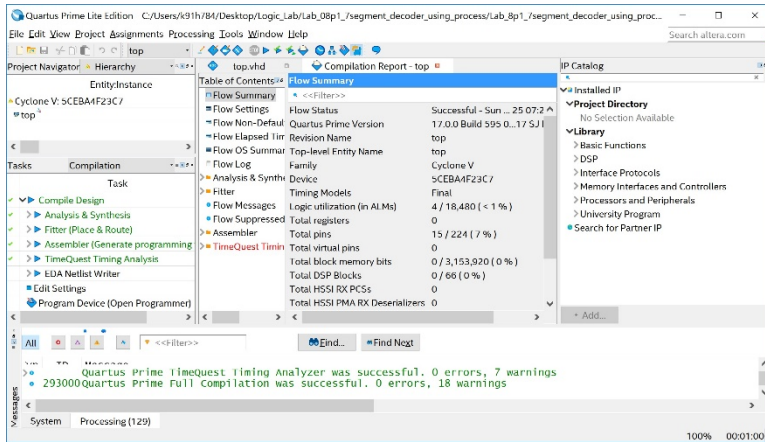


Figure 8.14
Quartus Window After Successful Compile and Synthesis

Assign the Ports to the Pins of the FPGA

We will now use a tool called *Pin Planner* to assign the ports of our top-level entity to the pins of the FPGA. Launch Pin Planner using the pull-down menus: Assignments → Pin Planner. You will see a graphical depiction of the FPGA with fields at the bottom to assign locations to any port that was in the entity during the most recent compile. At the bottom of Pin Planner, enter the pin locations for the 15x I/O of the decoder from Figure 8.1. The pin numbers will go in the “Location” column. Notice that there is a column called “Fitter Location” that has values for pin locations. When pins aren’t assigned manually during the first compile, Quartus will make automatic pin assignments for you. We **do not** want to use these locations. We want locations that correspond to the switches, LEDs, and HEX0 pins we are using in this exercise. To enter a value you can either click in the location field and type in the exact pin name or you can double click in the field to access the drop-down menu of available pins. Once you are complete, you should see the results in Figure 8.15.

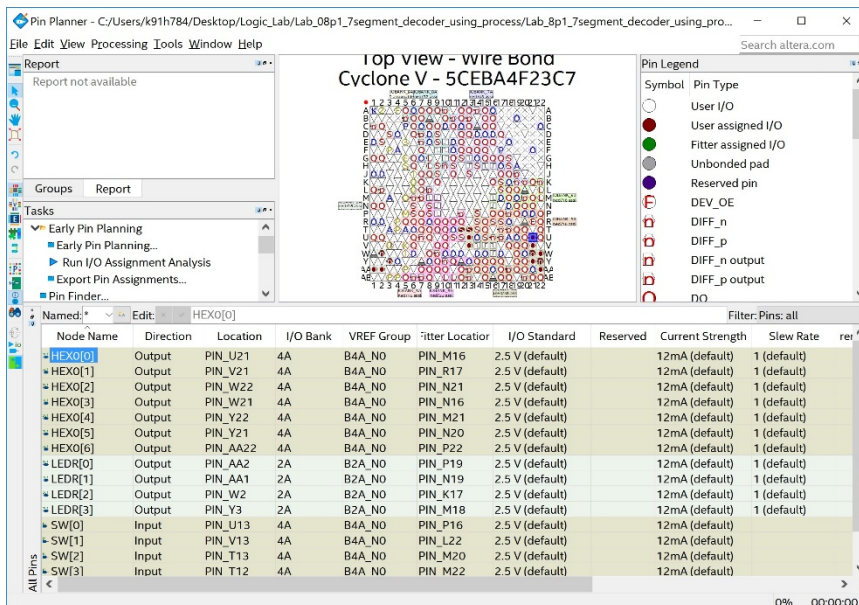


Figure 8.15
Quartus Pin Planner for the HEX0 7-Segment Decoder Project

Pin Planner does not have a save option. Instead, you simply close it using the drop-down menus: File – Close. Back in Quartus you'll notice that the tasks are no longer green. This indicates that the design needs to be compiled and synthesized again to take the new pin assignments into account. Double click on “Compile Design” and let it run until all tasks have been completed successfully.

Program the FPGA

We are now ready to download the prime number detector design to the FPGA. Plug in the DE0-CV board to your computer using the USB cable provided in the kit. You will power the DE0-CV board through the USB cable. **You do not need to plug in the AC adapter that is provided in the Terasic box.** All of the FPGA designs in this manual are small enough that the power from the USB cable is sufficient. Turn on the DE0-CV board by pressing the large red power button. When the board comes on, it will run a program that is stored in its non-volatile memory that flashes the I/O in a variety of patterns. We will be overwriting this program with our own.

In the Quartus task pane, double click on “Program Device”. This will bring up the *Programmer* tool. In Quartus versions 17 and newer, the programmer tool will automatically discover any devices on the board that the current design will fit into. When the programmer finds the correct Cyclone V device, you will see the results in [Figure 8.16](#). Notice that the file containing the information on how to configure the FPGA to implement our design is output_files/top.sof. The top.sof file is the end result of the Quartus synthesis and is what is to be downloaded to the FPGA.

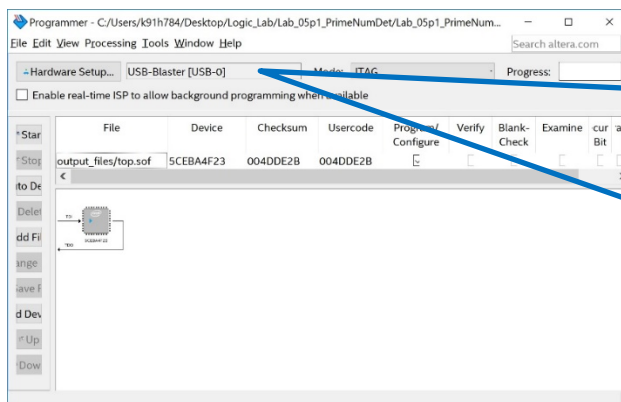


Figure 8.16
Quartus Programmer after Finding Device

If USB-Blaster isn't an option, you need to install the drivers manually so it will appear:

6. Plug in the USB cable between the computer and DE0-CV board, it will fail to load drivers.
7. Go to device manager in windows, go to USB controllers, right click on “Altera USB-Blaster” (it will have a yellow warning flag) – Update Software Driver
8. Choose Browse my computer

This the key step: you browse to the following path:

C:\intel\FPGA_lite\17.0\quartus\drivers\usb-blaster

Then **STOP**, don't browse into the x32 or x64.

Next, install, etc.... follow prompts.

The next time you relaunch the programmer in Quartus, you should be able to select USB-Blaster in the dropdown box

If the programmer does not automatically find the device, you can press the “Auto Detect” button on the left of the programmer window. Once it finds the device, you need to assign the top.sof file. Highlight the device, right-click, and select “Add File”. Browse to the output_files folder and select top.sof. When using the programmer for the first time, it may say “No Hardware” next to the hardware setup button. This means that the programmer is not using the correct drivers for the DE0-CV board. Click on the “Hardware Setup” button. For the “Currently Selected Hardware”, use the drop-down menus to select “USB-Blaster [USB-x]”. Click “Close”. Now the programmer will use the proper drivers to communicate with the DE0-CV board.

At this point we are ready to download the top.sof file to the FPGA in order to program it. Click on the “Start” button on the left side of the programmer window. The status of the programming will be displayed in the status bar in the upper-right corner of the window. When successful, it will say “100% (Successful)”.

Test your Design

Your design is now on the FPGA. You should be able to toggle the four slider switches on the DE0-CV and see the red LEDs turn on/off accordingly and the proper symbol displayed on the HEX0. If you are experiencing issues, you will need to debug your VHDL and/or go back and check each step in the Quartus design flow. If your VHDL seems correct, a good place to check for errors is in pin planner. Sometimes the pin locations will get dropped if you are still in edit mode when you close the pin planner window. Take a short video (<5 s) showing the proper operation of your design. **This video satisfies the requirements for deliverable #1.**

8.1.5.2 Save a Copy of your top.vhd for your Records

You now want to locate for your records the top.vhd file for the third deliverable for this exercise. This file is located in your main working directory (Desktop\Logic_Lab\ Lab_08p1_7segment_decoder_using_process\top.vhd). Go into your working directory for this project and locate this file. **This file satisfies the requirements for deliverable #2.**

After you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 8.1 After completing this lab exercise, can you:

- Use the modern digital design flow to take a VHDL model and synthesize it for implementation on an FPGA?
- Use a VHDL process and a conditional programming construct to model a combinational logic circuit?
- Use the *std_logic_vector* data type in a VHDL model?

Lab 8.2: Design Re-Use and Binary Characters on the 7-Segment Displays

8.2.1 Objective

The objective of this lab is to gain experience creating and using lower-level subsystems. This will be accomplished by creating a 7-segment decoder component and instantiating it numerous times within the top level entity. This lab will also give experience using signal concatenation within a port map, creating a new Quartus project from a prior project, and importing signal assignments from an external file.

8.2.2 Learning Outcomes

After completing this lab you should be able to:

- Create a new Quartus project by copying an existing project.
- Create a 7-segment decoder component and instantiate it multiple times in a higher-level system.
- Use the DE0-CV User's Guide to find pin assignments for the Cyclone V FPGA.
- Import signal assignments into Quartus from an external CSV file.
- Use signal concatenations within the port map of a lower level component.

8.2.3 Parts Needed

- DE0-CV FPGA board.

8.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a design that uses multiple instantiations of a 7-segment decoder component to drive all the HEX character displays on the DE0-CV FPGA board (50% of exercise).
2. Demonstrate a design that displays the binary values of the four slider switches on the character displays (40% of exercise)
3. Provide your top.vhd design file (10% of exercise).

8.2.5 Lab Work & Demonstration

8.2.5.1 *Creating a 7-Segment Decoder Subsystem to Drive all the Character Displays*

In the lab 8.1 you created a 7-segment decoder using a process. In this lab, you are going to put your decoder logic into its own subsystem so that it can be instantiated numerous time by the top level design. You will call your subsystem "char_decoder.vhd". You are going to drive all six of the character displays on the DE0-CV board (HEX5, HEX4, HEX3, HEX2, HEX1, and HEX0) with their own decoder. The input to all decoders will be a 4-bit code coming from the slider switches (SW3, SW2, SW1, and SW0). You will also drive the LEDs on the DE0-CV board (LEDR3, LEDR2, LEDR1, and LEDR0) with the values coming from the slider switches. As you change the binary values of the slider switches, the corresponding HEX character will show on all six character displays. [Figure 8.17](#) shows the block diagram for this design.

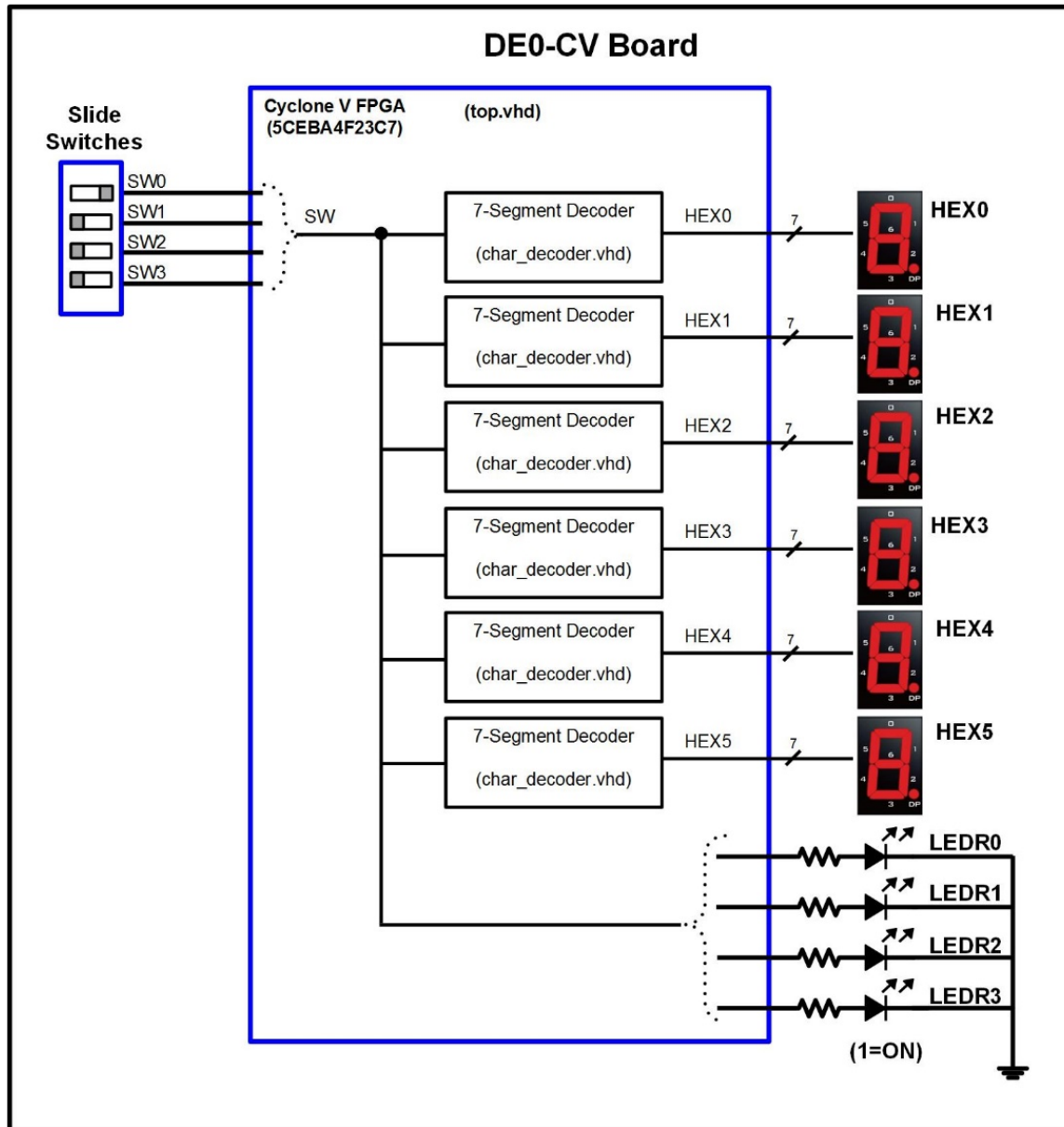


Figure 8.17
Block Diagram of the System to Drive Each Character Display with its own Decoder Component

Figure 8.18 shows a picture of the I/O on the DE0-CV board that will be used in this part of the exercise.

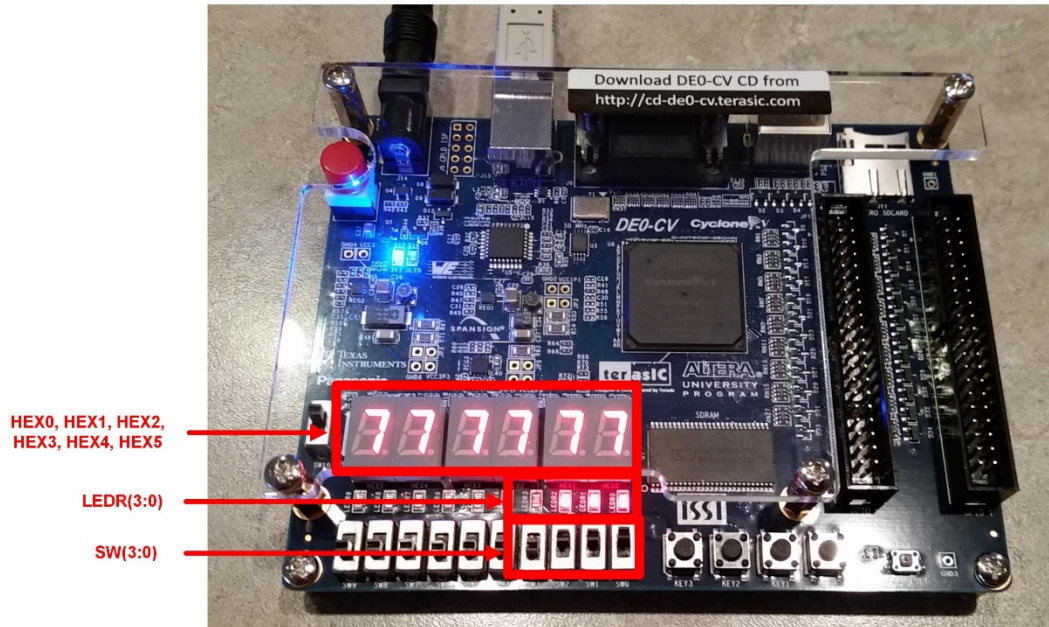


Figure 8.18
Picture of the System to Drive Each Character Display with its own Decoder Component on the DE0-CV Board

Create a Quartus project by Copying a Prior Lab (8.1)

In lab 8.1 you created a project in which you spent a significant amount of time entering project settings, creating the top.vhd file, and assigning pins. Quartus allows you to create a new project by copying a prior project so that you don't lose past effort. Launch Quartus and then open lab 8.1 using the pull down menus: File → Open Project. Browse to your Logic_Lab folder and select "Lab_08p1_7segment_decoder_using_process.qpf".

Once the project is open, copy it to a new project using the pull down menus: Project → Copy Project. In the dialog that comes up, you can specify the location and name of the new project. Note that Quartus will make the new project folder so you don't have to do this manually. Browse to the folder where you are working on these labs (i.e., Logic_Lab) and append the name of the new folder to its path: give the name: "Lab_8p2_design_reuse_and_binary_chars_part1". Give the name of the project the same name. Your settings should look like . Make sure the "Open new project" box is checked. Once your settings look like [Figure 8.19](#), click "OK".

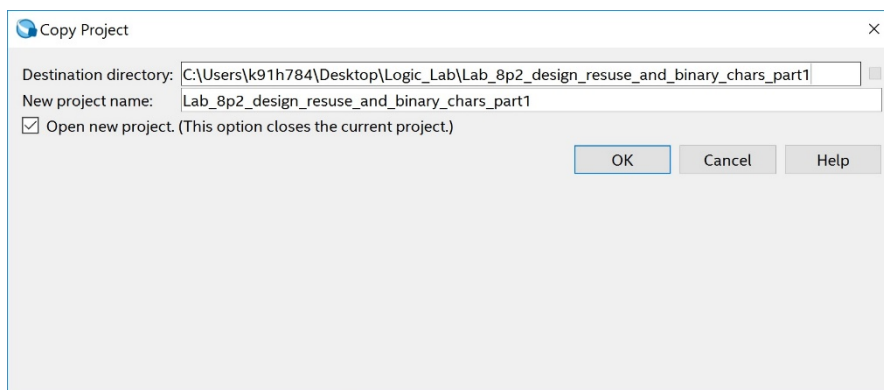


Figure 8.19
Copy Project Settings in Quartus

Create the 7-segment decoder subsystem (char_decoder.vhd).

You need to add a new file to your project called “char_decoder.vhd” that will contain the logic to drive the HEX character displays. This logic was created in last week’s lab so you’ll just need to create a new file, copy in the functionality, rename signals, redeclare the ports. Note that when you copied over the project from 8.1, this functionality resides within the existing top.vhd file. We will take it out of top.vhd and put it into a file called char_decoder.vhd.

In Quartus, create a new VHDL file using the pull down menus: File → New. Select “VHDL File” in the dialog that appears. A blank file will come up in Quartus. Save the file with the desired name using the pull down menus: File → Save As and name it “char_decoder.vhd”. Note that since the top.vhd already exists, the new char_decoder.vhd file will be inserted below top in the hierarchy, which is what we want.

The first step in designing the subsystem is including the necessary libraries. We will be using the std_logic and std_logic_vector data types from the std_logic_1164 library. To include this library, add the following VHDL statements to the beginning of your char_decoder.vhd file.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

We are now ready to enter the VHDL entity for the design. The decoder subsystem will have the following ports:

- **BIN_IN (3 downto 0)** This is the 4-bit binary input vector to the decoder.
- **HEX_OUT (6 downto 0)** This is the 7-bit output vector to drive a character display.

Enter the following VHDL entity into your char_decoder.vhd.

```
entity char_decoder is
  port (BIN_IN   : in  std_logic_vector (3 downto 0);
        HEX_OUT  : out std_logic_vector (6 downto 0));
end entity;
```

Now create the architecture for the decoder. You will copy your process from the current top.vhd file. You’ll need to rename the architecture and all signals to reflect the new entity declaration (i.e., SW will be replaced with BIN_IN, LEDR will be replaced with HEX_OUT). Save your char_decoder.vhd file.

Modify your top.vhd File to Drive all six HEX displays

You are now ready to modify your top.vhd to reflect the design for this exercise. You will still use the SW input to your system. You will still drive the SW inputs to the lower 4-bits of the red LEDs on the DE0-CV board (LEDR). However, you will now be driving all six of the character displays on the DE0-CV board. These should be called HEX0, HEX1, HEX2, HEX3, HEX4, and HEX5 and all have a data type of std_logic_vector(6 downto 0). Your entity should look like:

```
entity top is
  port (SW      : in  std_logic_vector(3 downto 0);
        LEDR   : out std_logic_vector (3 downto 0);
        HEX0   : out std_logic_vector (6 downto 0);
        HEX1   : out std_logic_vector (6 downto 0);
        HEX2   : out std_logic_vector (6 downto 0);
        HEX3   : out std_logic_vector (6 downto 0);
        HEX4   : out std_logic_vector (6 downto 0);
        HEX5   : out std_logic_vector (6 downto 0));
end entity;
```

You are now ready to alter the architecture. In the existing architecture that was copied over, you can leave the signal assignment of LEDR <= SW; but you should delete the decoder process. You will implement six decoders by instantiating six versions of the char_decoder.vhd component. To use the decoder it must first be declared **before** the begin statement. The syntax for the component declaration will look like:

```
component char_decoder
  port (BIN_IN   : in  std_logic_vector (3 downto 0);
        HEX_OUT  : out std_logic_vector (6 downto 0));
end component;
```

Once declared, you can instantiate the subsystem **after** the *begin* statement. You will need to instantiate it six times, once for each of the displays on the DE0-CV board. Remember that each of the subsystems will have an input of *SW*. The syntax for instantiating the subsystem is:

```
C0 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX0);
C1 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX1);
C2 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX2);
C3 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX3);
C4 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX4);
C5 : char_decoder port map (BIN_IN => SW, HEX_OUT => HEX5);
```

Save your top.vhd file.

[Compile your Design](#)

Compile your project by double clicking on “Compile Design” in the task pane. Fix any syntax errors you have and re-compile until successful.

[Import the Pin Assignments Using an External file](#)

After a successful compile, Quartus will read in the new ports being used in the design. You will need to assign the pins for the five new HEX displays (35 new signals!). As digital systems get larger and larger, it becomes too time consuming to use the graphical Pin Planner tool within Quartus. Instead, designers typically use an external CSV file to import the assignments into Quartus. You are going to create one of these files manually and then import it into Quartus.

A CSV file is a text file with comma delimited data (CSV = comma separated values). For this lab, it is easiest to use a simple text editor to create the file. MS Excel is often used to create CSV files; however, sometimes the CSV file created in Excel contains special characters that prevent it from being imported successfully. As a result, it is suggested that you use a simple text editor. Start a text editor (e.g., notepad, notepad++,...). You are going to type in the signal name, port direction, and pin assignment comma delimited. Start by typing the following into your text file:

```
# This is the pinout assignments for the DE0-CV Board (# means comment)
To,Direction,Location
HEX0[6],Output,PIN_AA22
HEX0[5],Output,PIN_Y21
HEX0[4],Output,PIN_Y22
HEX0[3],Output,PIN_W21
HEX0[2],Output,PIN_W22
HEX0[1],Output,PIN_V21
HEX0[0],Output,PIN_U21
```

Notice that the above text defines the pin assignments for the HEX0 port, which you have already manually entered in the pin planner tool. When this file is imported, it will overwrite the values in pin planner. Save the CSV file in your project directory and name it “pin_assignments.csv”. Use the DE0-CV user’s manual to look up the pin numbers for all of the ports used in this design. You will need to add the assignments for the existing I/O SW and LEDR in addition to the new I/O HEX1, HEX2, HEX3, HEX4, and HEX5.

Once the CSV file is created, it can be imported into Quartus. In Quartus, use the pull down menus to import the pin assignments using: Assignments → Import Assignments. Browse to your CSV file and click “OK”. This should have imported in all of the assignments for your design. To verify that everything was imported correctly, run the Pin Planner tool and scroll through the assignments. If you made any syntax errors in your CSV file, the assignments in Pin Planner will be blank. If you entered the wrong pin, you won’t be able to find the mistake until you download your design. Re-Compile your design.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see the values of the slider switches being displayed on the red LEDs and all of the HEX displays showing the same corresponding character. Verify that your design works for each of the 16 possible input codes. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design. **This video satisfies the requirements for deliverable #1.**

8.2.5.2 Display Binary Characters on the 7-Segment Displays

Now you are going to design a system that will display the characters on the HEX displays corresponding to the binary values on the slider switches. You will only use four of the character displays for this part (HEX3, HEX2, HEX1, and HEX0). The character displays will only show the symbol 0 or 1 based on the value on the corresponding slider switch. For example, if you set the slider switches to “1010”, the displays should show the symbols 1 0 1 0. [Figure 8.20](#) shows how the system will work.

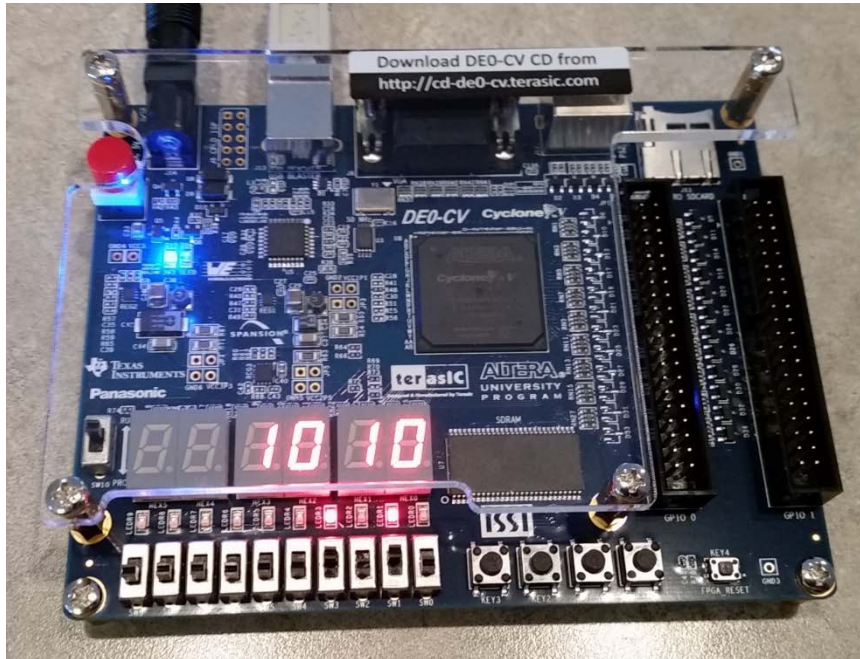


Figure 8.20
Picture of System to Drive Binary Characters to the 7-Segment Displays

[Create a New Quartus Project by Copying Part 1](#)

Create a new project for this design by copying the project from part 1. Use the Quartus “Copy Project” operation and name the new folder and project “Lab_8p2_design_reuse_and_binary_chars_part2”.

NOTE: The Quartus *copy project* command will only copy project files. The “pin_assignments.csv” file you put in the project folder for part 1 will not be automatically copied over. It is a good practice to manually copy this into your new project folder so that you have the most up to date assignment file.

[Design the Binary Character Display Logic](#)

You are now ready to implement the functionality for part 2 of this exercise. In the top.vhd, delete the HEX5 and HEX4 ports from the top entity. Next, delete the component instantiations that that drove the HEX5 and HEX4 ports in the architecture.

At this point you need to figure out a way to drive only 0 and 1 characters to the displays based on the corresponding slider switch. Consider the case of SW0 driving HEX0. Notice that the input into your char_decoder.vhd is a 4-bit vector. What if you could drive in 0’s for bits (3 downto 1) of this 4-bit vector and then connect SW0 to bit 0? This would have the effect that when SW0 was a 0, the 4-bit input vector to the char_decoder would be “0000” and HEX0 would display the symbol 0. If SW0 was a 1, the 4-bit input vector to the char_decoder would be “0001” and HEX0 would display the symbol 1. All that you need to do to accomplish this is to concatenate three leading zeros with the switch input.

There are multiple ways to concatenate signals in VHDL. You *could* create a new 4-bit signal and then use a signal assignment and the concatenation operator (&). A faster way to accomplish this when driving component inputs

is to put the concatenation directly in the port mapping. VHDL supports this type of concatenation using the following syntax:

```
C0 : char_decoder port map (BIN_IN => "000" & SW(0), HEX_OUT => HEX0);
```

Modify your architecture to implement this behavior for each of the four char_decoder.vhd subsystems. Save your design.

[Compile your Design](#)

Compile your project by double clicking on “Compile Design” in the task pane. Fix any syntax errors you have and re-compile until successful.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see the binary value of SW (3 downto 0) being displayed on HEX3, HEX2, HEX1, and HEX0. Verify that your design works for each of the 16 possible input codes. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design. **This video satisfies the requirements for deliverable #2.**

[8.2.5.3 Save a Copy of your top.vhd for your Records](#)

Locate the top.vhd file for part 2 of this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 8.2 After completing this lab exercise, can you:

- Create a new Quartus project by copying an existing project?
- Create a 7-segment decoder component and instantiate it multiple times in a higher-level system?
- Use the DE0-CV User's Guide to find pin assignments for the Cyclone V FPGA?
- Import signal assignments into Quartus from an external CSV file?
- Use signal concatenations within the port map of a lower level component?

Chapter 9: Behavioral Modeling of Sequential Logic

Lab 9.1: Ripple Counter and the Character Displays

9.1.1 Objective

The objective of this lab is to gain experience designing and using sequential storage devices in VHDL. You will create a D-flip-flop model using a process and then instantiate it multiple times to build a 38-bit ripple counter. The most significant bits of the counter (i.e., the slowest toggling bits) will be used to drive the LEDs and HEX character displays on the DE0-CV FPGA board. This lab will also give experience using a logic analyzer to measure the frequency of a counter.

9.1.2 Learning Outcomes

After completing this lab you should be able to:

- Create a model of a D-flip-flop storage device using a process in VHDL.
- Create a 38-bit ripple counter out of the D-flip-flop subsystem.
- Use portions of the counter to drive I/O on the DE0-CV board (LEDR, HEX displays, and GPIO_1).
- Measure the frequency of the counter using a logic analyzer.

9.1.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

9.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a 38-bit ripple counter made of D-flip-flops where the most significant bits drive the HEX displays and LEDs on the DE0-CV FPGA board (70% of exercise).
2. Measure the frequency of the counter using a logic analyzer (20% of exercise).
3. Provide your top.vhd design file (10% of exercise).

9.1.5 Lab Work & Demonstration

You are going to build a 38-bit ripple counter out of D-flip-flops. You will first create a model of a rising edge triggered D-flip-flop in VHDL (dfflipflop.vhd) and then use it in your top.vhd to create the counter. The counter will be clocked from the 50 MHz oscillator on the DE0-CV board. The reset for the counter will come from the active LOW Reset push button on the DE0-CV board. The counter needs to be 38-bits wide in order to provide signals that toggle slow enough to be seen with the human eye when observed on the DE0-CV LEDs. The upper 24-bits of the counter will be used to drive the 6x HEX character displays on the DE0-CV board through your char_decoder.vhd components. The upper 10-bits of the counter will be used to drive the 10x red LEDs on the DE0-CV board. The lower 8-bits of the counter will drive the GPIO_1 header on the DE0-CV board to enable a logic analyzer measurement of the counter. Note that you will need to add the GPIO_1 port to your entity and find the pin assignments for this I/O from the DE0-CV User's Guide. [Figure 9.1](#) shows a block diagram of the ripple counter system.

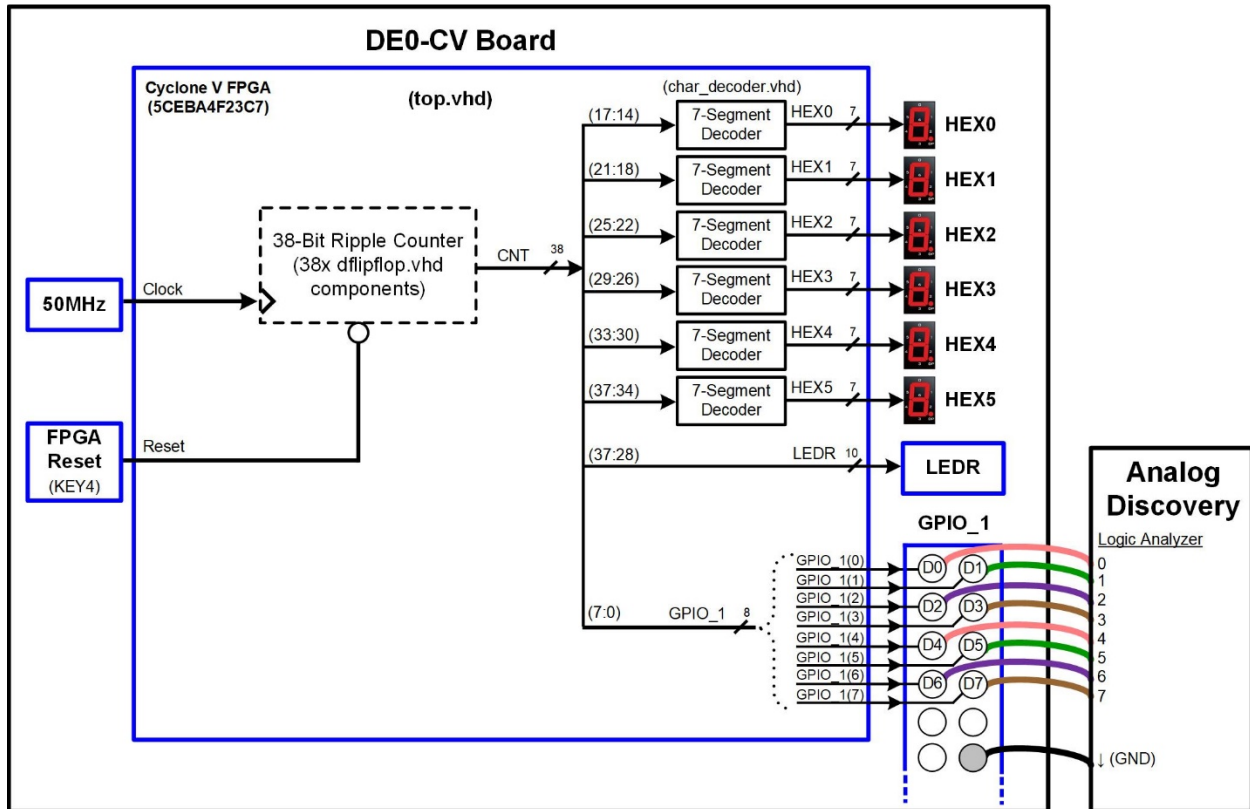


Figure 9.1
Block Diagram of the Ripple Counter System

Figure 9.2 shows the I/O that will be used in this exercise.

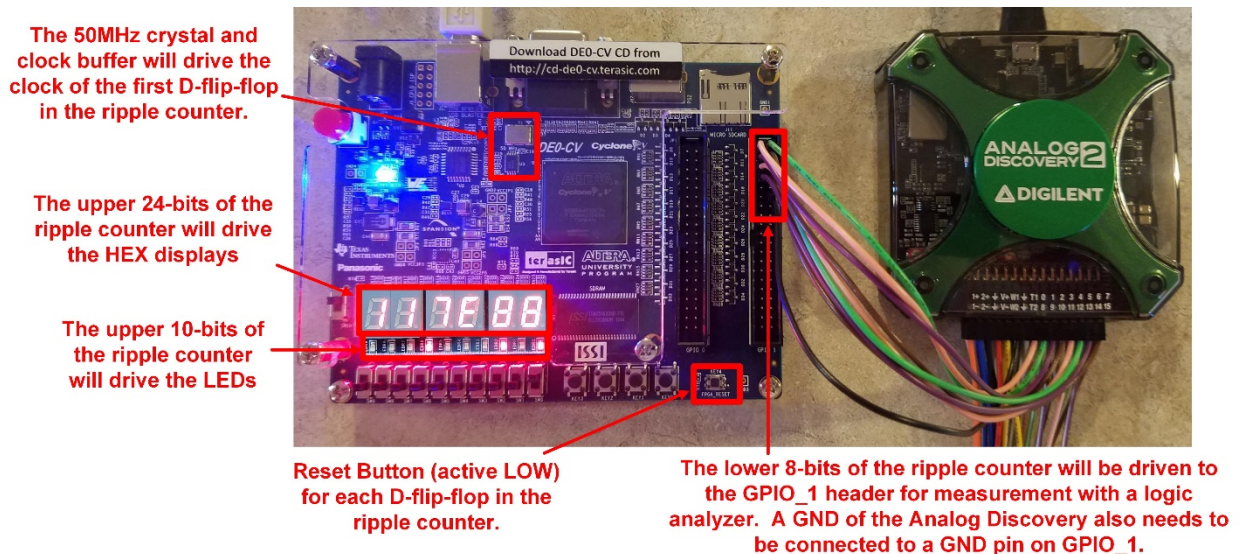


Figure 9.2
Picture of the Ripple Counter System on the DE0-CV Board

9.1.5.1 Implement the Ripple Counter System in VHDL on the DE0-CV Board

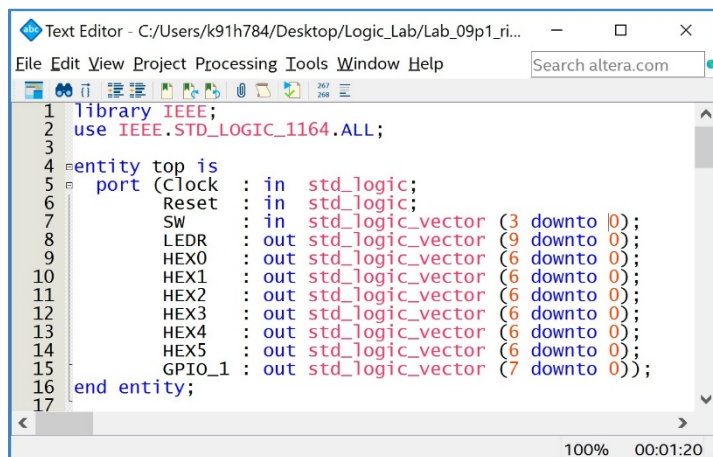
Create a New Quartus Project by Copying Lab 8.2

We want to re-use many of the I/O that were used in lab 8.2 so we'll create a new Quartus project for this exercise by copying lab 8.2. Open lab 8.2 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project "Lab_09p1_ripple_counter".

Note that when you copy the project, the pin_assignments.csv file will not automatically copy over. You should manually copy this file into your new project directory.

Modify your top.vhd file Entity to Support the Additional I/O that will be Used in this Exercise

In this lab you will be using additional ports beyond the entity definition from lab 8.2. These signals include *Clock* and *Reset* to drive the synchronous circuitry in the design in addition to widening the LEDR vector to 10-bits to drive all of the red LEDs on the DE0-CV board. You will also be adding the lower 8-bits of the GPIO_1 header. All ports should be declared as types `std_logic` or `std_logic_vector`. Make the entity modifications to your top.vhd file. Once complete, your entity definition should look like [Figure 9.3](#).



```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity top is
5     port (Clock      : in  std_logic;
6           Reset      : in  std_logic;
7           SW         : in  std_logic_vector (3 downto 0);
8           LEDR       : out std_logic_vector (9 downto 0);
9           HEX0       : out std_logic_vector (6 downto 0);
10          HEX1       : out std_logic_vector (6 downto 0);
11          HEX2       : out std_logic_vector (6 downto 0);
12          HEX3       : out std_logic_vector (6 downto 0);
13          HEX4       : out std_logic_vector (6 downto 0);
14          HEX5       : out std_logic_vector (6 downto 0);
15          GPIO_1     : out std_logic_vector (7 downto 0));
16 end entity;
17

```

Figure 9.3
Entity Definition for the 38-Bit Ripple Counter System

Create a Model for a Rising Edge Triggered D-flip-flop

Now you are going to create a model for a D-flip-flop. This model will be contained in a file named "dfflipflop.vhd". The model should be rising edge sensitive and have an active low reset. The data input will be called *D* and the outputs will be called *Q* and *Qn*. In Quartus, create a new VHDL file using the pull down menus: File → New. Select "VHDL File" in the dialog that appears. A blank file will come up in Quartus. Save the file with the desired name using the pull down menus: File → Save As and name it "dfflipflop.vhd". Note that since the top.vhd already exists, the new dfflipflop.vhd file will be inserted below top in the hierarchy, which is what we want.

Enter the VHDL for a rising edge triggered D-flip-flop with active LOW reset. Your model should use the `std_logic` data types from the `std_logic_1164` library. You should use a single process to implement your D-flip-flop. Once complete, save your file.

Create the 38-bit Ripple Counter by Instantiating D-Flip-Flop Components

Recall that a ripple counter is created by wiring D-flip-flops in a toggle-flop configuration and then using the Qn output to drive the next stage of the counter. Figure 9.4 shows the architecture of a ripple counter for reference.

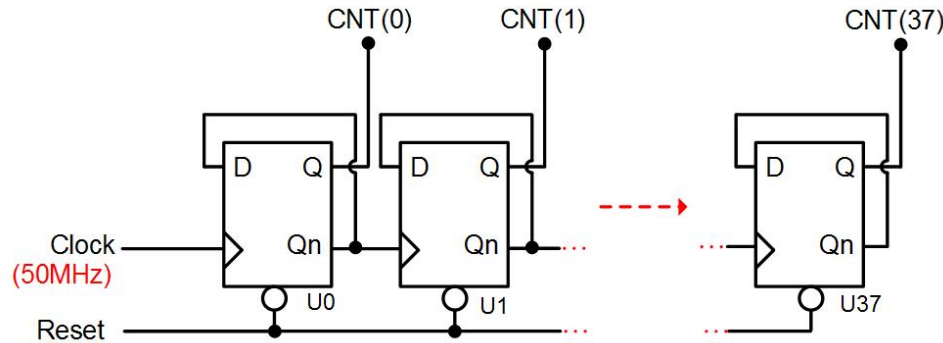


Figure 9.4
Ripple Counter Architecture

The clock for the ripple counter will come from the 50 MHz oscillator on the DE0-CV board. This will result in the LSB of the counter running at 25 MHz. This is much too fast to observe on an LED. In order to be able to observe some of the bits of the counter on LEDs, we need to observe bits that are toggling in the Hz range. Since each subsequent bit in the counter will run at $\frac{1}{2}$ the frequency of the preceding bit, we can continue to add bits to the counter until the most significant bits are slow enough to be observed. This will require 38 bits in the counter. To create a 38-bit counter, you will need to instantiate your D-flip-flop 38 times and wire them in the ripple counter configuration shown in Figure 9.4. You will need to create internal signals to connect to the D, Q, and Qn ports of the D-flip-flops. Create two, 38-bit signal vectors called **CNT** and **CNTn**.

Connect the Ripple Counter Outputs to LEDR, HEX, and GPIO_1 Outputs

Connect the most significant 24-bits of the counter to your 6x instantiations of your char_decoder.vhd component to drive the 6x HEX displays. You'll connect the bits in groups of four. For example, CNT(37 downto 34) will drive HEX5, CNT(33 downto 30) will drive HEX4, etc. Connect the most significant 10-bits of your counter directly to the 10x red LEDs on the DE0-CV board. Connect the least significant 8-bits of your counter to the GPIO_1 port.

Compile your Design

Compile your design and fix any errors that you encounter.

Assign the Pins for the Additional I/O used in this Exercise

Locate the pin assignments for the Clock, Reset, additional LEDR I/O, and GPIO_1 pins used in this lab using the DE0-CV User's Guide. Add these locations to your pin_assignments.csv document and then import into Quartus. Verify that the pin assignments are correct by launching the Pin Planner tool. Once they are verified, close the Pin Planner tool and recompile.

Download and Test Your Design

Open the programmer tool and download your design to the FPGA. You should now see a counter pattern on the LEDRs and HEX displays. Verify that your design works as expected including the reset functionality. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design on the LEDs and HEX displays.

This video satisfies the requirements for deliverable #1.

9.1.5.2 Take a Logic Analyzer Measurement of your Counter

Now you are going to take a logic analyzer measurement of the lower 8-bits of your counter, which are being driven to the pins on the GPIO_1 header. A logic analyzer is an instrument that measures the digital values on a set of signals (aka, a “bus”). Since only 1’s and 0’s are stored, the circuitry to implement a logic analyzer is much simpler than an oscilloscope. This allows the logic analyzer to have more channels. Logic analyzers can have hundreds of channels, which is useful for debugging digital systems because many vectors can be observed in real-time. The busses can be displayed in different radices and in both waveform and listing format. The Analog Discovery has 16 logic analyzer channels. We will use the logic analyzer to measure the lower 8-bits of the ripple counter and determine its frequency.

[Connect the Analog Discovery’s Logic Channels to the GPIO_1 port on the DE0-CV](#)

The logic analyzer channels on the Analog Discovery are labeled as numbers (i.e., 0, 1, 2...). You should connect the lower 8 channels of the logic analyzer to the GPIO_1 header pins labeled D0 → D7 in the DE0-CV user’s manual. The block diagram in [Figure 9.1](#) shows how the channels between the GPIO_1 header are mapped to the logic analyzer. Also connect one of the grounds of the Analog Discovery to a ground pin on the GPIO_1 header. The grounds of the Analog Discovery are labelled with the ↓ symbol. After the connection to the DE0-CV board is complete, plug in the Analog Discovery to your computer using the USB cable. Your connection should will look like [Figure 9.2](#).

[Launch Waveforms](#)

The Analog Discovery is controlled using an application called *Waveforms*. If *Waveforms* isn’t installed on the computer you are using, you can download it for free from Digilent.com (<http://store.digilentinc.com/>). Once on this website, on the left select “Scopes, Instruments, & Circuits”, and then on the “Waveforms 2015 (Download Only)” product. On the next screen, select “Download Here”. On the next screen you will find “Latest Downloads” where you can choose “Windows”. The 65MB download will then commence. Once downloaded, run the *.exe file and the software will be installed.

Launch *Waveforms* (Start –Digilent – Waveforms). The software will automatically recognize the Analog Discovery and connect. The *Waveforms* startup window shown in [Figure 9.5](#) will appear. From this window, you can launch all of the tools associated with the Analog Discovery. Each tool brings up a new tab within the workspace window. Multiple tools can be launched and ran at the same time.

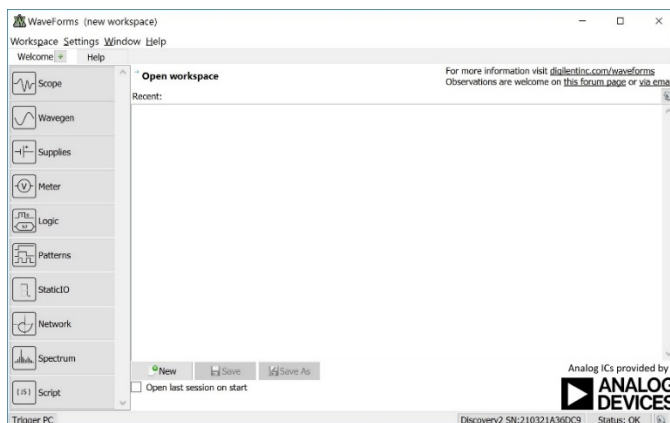


Figure 9.5
Waveforms 2015 Startup Window

[Configure the Logic Analyzer Tool](#)

Start the logic analyzer by clicking on the “Logic” button on the left side of the welcome screen. The logic analyzer tool will appear. The first thing to do is define the signals that we are measuring. On the left of the screen there is a section that shows all of the signals in the measurement. Select the “Click to Add channels” button and then choose “Bus”. A *bus* is the term used to describe a group of signals. In this lab exercise, we will be measuring an 8-bit bus on the GPIO_1 header that is driven by the lower 8-bits of CNT on the FPGA.

In the options window that appears, name the bus *CNT*. On the left side of the window there is a list of available logic analyzer channels to include in the bus. Highlight the eight channels “DIO 7” through “DIO 0” by selecting them while holding down the Shift key. Once selected, click on the + sign to add them to the box on the right. Your settings should look like [Figure 9.6](#). Once the bus looks correct, click the “Add” button.

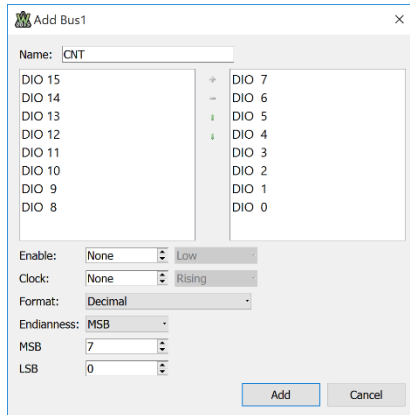


Figure 9.6
Logic Analyzer Bus Setup

Run the Logic Analyzer Tool

While there are a variety of settings that will need to be configured, go ahead and click on the “Single” button to take a measurement. After the measurement, you will see a decimal value for CNT and also some edges on the signals (7:0). You may need to adjust the zoom to see the signals correctly.

Adjust the zoom by selecting different values in the *Base* setting drop-down in the upper right corner of the screen. Each time you zoom in or out, you’ll need to click the “Single” button to re-run the analyzer and fill the screen. At 50 MHz, a setting of 0.05 us/div is a good zoom factor. Your measurement should look like [Figure 9.7](#).

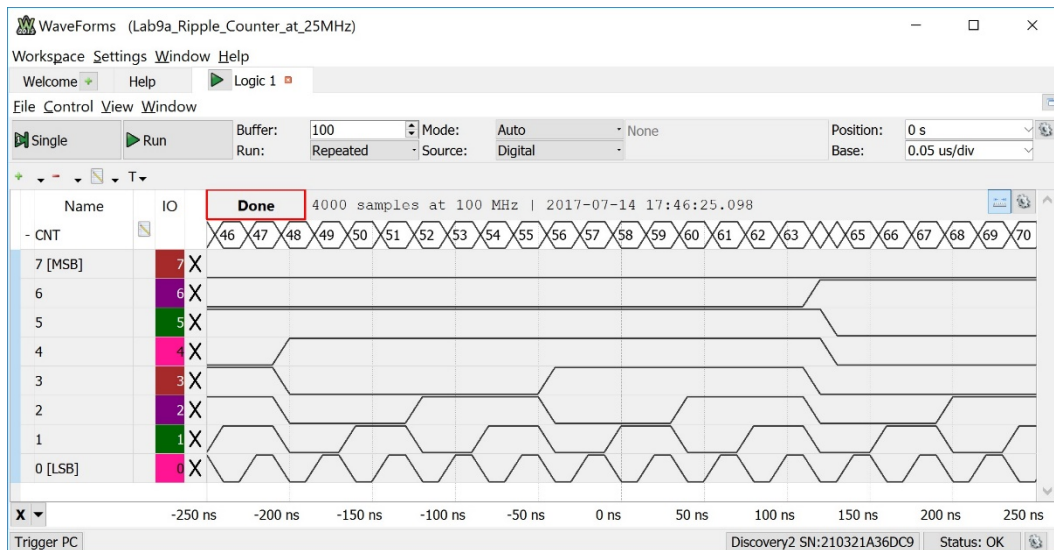


Figure 9.7
Logic Analyzer of Ripple Counter with No Trigger

Setup the Trigger

A “trigger” is a pattern that the logic analyzer will search for and then place in the center of the measurement. We want to setup a trigger to look for CNT=x”00”. Highlight all of the bits within CNT. Click the trigger button (it is the “T” right below the run button). Select “0 LOW”. Now when you click the “Single” button, you’ll notice that the center of the waveform screen is the transition between 255 to 0. Your measurement will now look like [Figure 9.8](#).

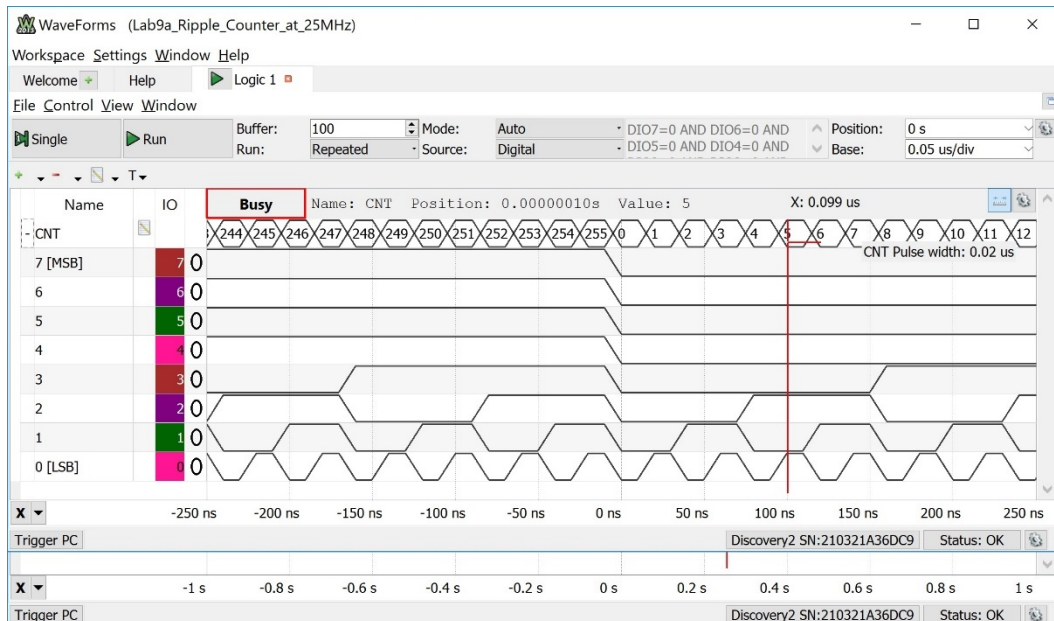


Figure 9.8
Logic Analyzer Measurement of CNT with Trigger set to x”00”

Click “Run” and allow the analyzer to *free run*. When using the “Run” button (instead of the “Single” button), the analyzer will continually take measurements and replace the data on the screen. Since we have the trigger set to CNT=x”00”, each measurement will mostly look the same and the screen should appear somewhat static. To prove to yourself that the data is actually live, unplug the D0 channel from the DE0-CV board and observe that the signal goes flat in the analyzer.

Measure the Frequency of the Counter

You can take a variety of measurements using the “HotTrack” feature within the waveform window. This button is the *ruler* icon in the upper right corner of the waveform pane. Click on the “HotTrack” button to turn it on. Now as you move your cursor around on the waveform, you’ll see measurements pop up. Place your cursor over the bus and measure the pulse width. You should see a measurement that gives the pulse width of the counter. You can take the inverse of this period to find the rate that your counter is incrementing. Since the incoming clock is 50 MHz, CNT(0) will have a frequency of 25 MHz. However, since both the HIGH and LOW values of CNT(0) are considered when interpreting the bus as a number, the counter is actually producing numbers at a rate of 50 M *bits per second (bps)*. This corresponds to a pulse width of 0.02 us.

Take a screenshot of the logic analyzer measurement displaying the HotTrack pulse width value. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

Save your Waveforms Workspace

You can save your Analog Discovery measurement setup. This is useful for larger logic analyzer measurements in which there are numerous steps required to setup the busses and triggers. Click “Workspace – Save”. Give a descriptive name and save within your Quartus project directory. Close Waveforms.

9.1.5.3 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 9.1 After completing this lab exercise, can you:

- Create a model of a D-flip-flop storage device using a process in VHDL?
- Create a 38-bit ripple counter out of the D-flip-flop subsystem?
- Use portions of the counter to drive I/O on the DE0-CV board (LEDR, HEX displays, and GPIO_1)?
- Measure the frequency of the counter using a logic analyzer?

Lab 9.2: A “Walking 1” Finite State Machine

9.2.1 Objective

The objective of this lab is to gain experience designing finite state machines (FSMs) in VHDL using a behavioral modeling approach. You are going to design a FSM that will produce a walking 1 pattern on the red LEDs of the DE0-CV board. You will also drive the walking 1 pattern to the pins of the GPIO_1 connector for observation with a logic analyzer measurement. You will also gain experience using a divided down clock to trigger your FSM.

9.2.2 Learning Outcomes

After completing this lab you should be able to:

- Create a FSM using a three process, behavioral modeling approach.
- Use a divided down clock in order to drive the FSM at a slower rate.
- Take logic analyzer measurement of a walking 1 pattern.

9.2.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

9.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a walking 1 pattern on the red LEDs of the DE0-CV board (70% of exercise).
2. A logic analyzer measurement of the walking 1 pattern (20% of exercise).
3. Provide your top.vhd design file (10% of exercise).

9.2.5 Lab Work & Demonstration

A *walking 1* pattern is one that asserts one, and only one, signal at any given time within a group of signals. When this is displayed on LEDs, it appears that the asserted LED *walks* across the display. This type of pattern can be created using a finite state machine in which each state represents one of the output patterns. The Moore type outputs simply assert the desired signal when in each corresponding state of the machine. An input can be used to set the direction of the walking 1 pattern.

In this lab exercise you are going to create a FSM that will produce a walking 1 pattern within a 10-bit vector. The 10-bit output vector will drive the 10x red LEDs and lower 10x pins on the GPIO_1 header. This FSM will require 10x states and use Moore-type outputs. The direction of the walking 1 pattern will be dictated by SW(0). When SW(0)=1, the pattern will walk from LEDR(0) to LEDR(9) and repeat. When SW(0)=0, the pattern will walk from LEDR(9) to LEDR(0) and repeat. [Figure 9.5](#) shows a graphical depiction of the walking 1 patterns as seen on the red LEDs.

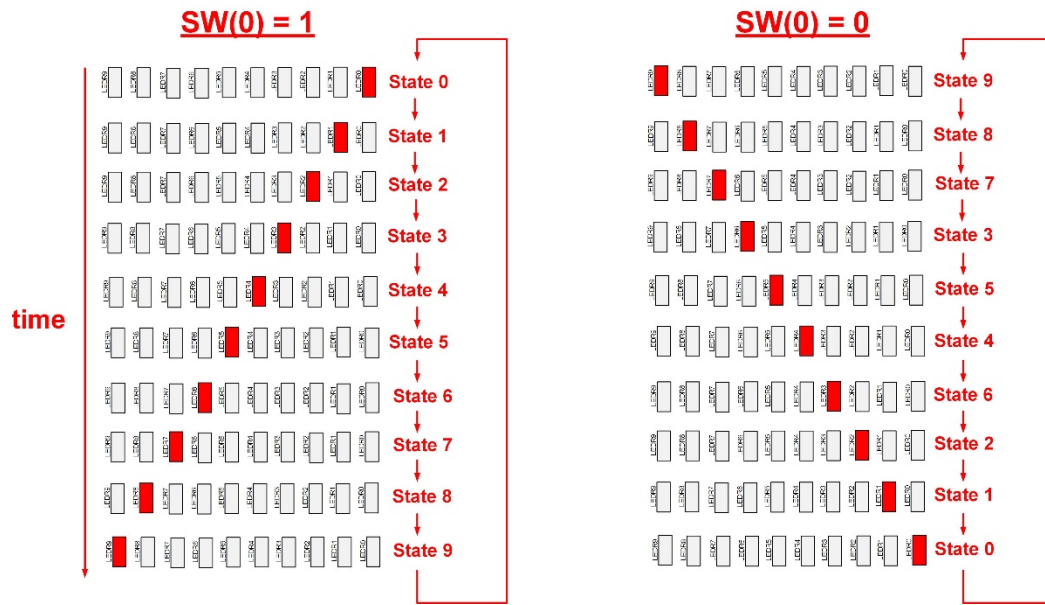


Figure 9.9
Walking 1 Patterns on the Red LEDs

The FSM will need to be clocked using a frequency that is slow enough so that the walking 1 pattern can be seen with the human eye on the red LEDs. One of the ways to create a slower clock signal is to continually divide the incoming 50 MHz clock by two using a series of toggle flops. You have already implemented this type of clock divider in lab 9.1 in the form of a 38-bit ripple counter. In this lab you will use **bit 21** of your 38-bit ripple counter to clock your FSM. This bit has a frequency of 11.9 Hz.

After observing the walking 1 pattern on the red LEDs, you will take a logic analyzer measurement on the 10-bit vector on the GPIO_1 header. **Figure 9.10** shows the block diagram for the walking 1 FSM system.

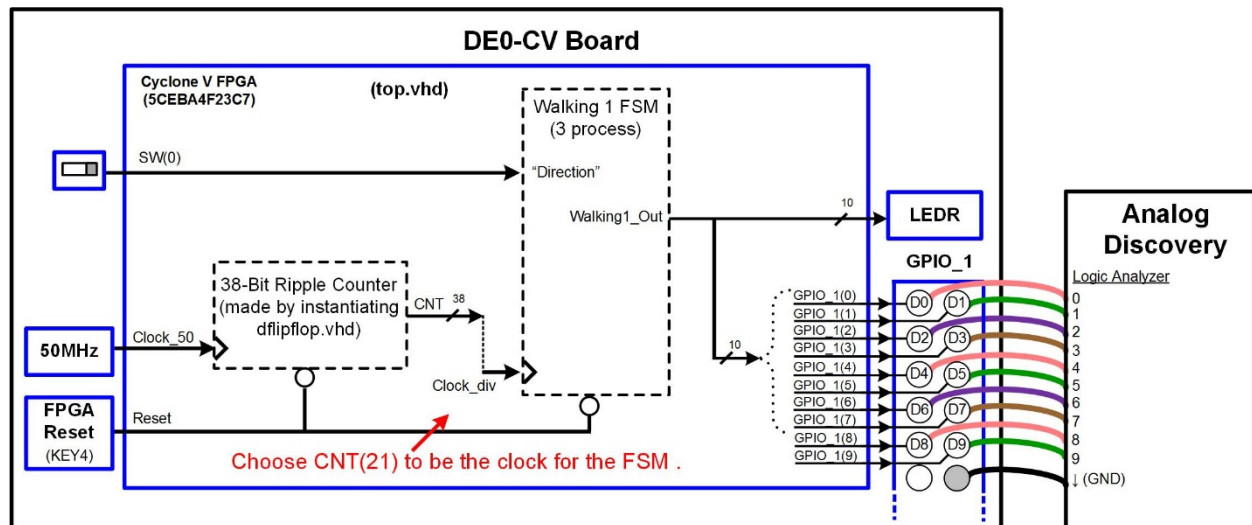


Figure 9.10
Block Diagram of the Walking 1 FSM System

Figure 9.11 shows the I/O that will be used in this lab.

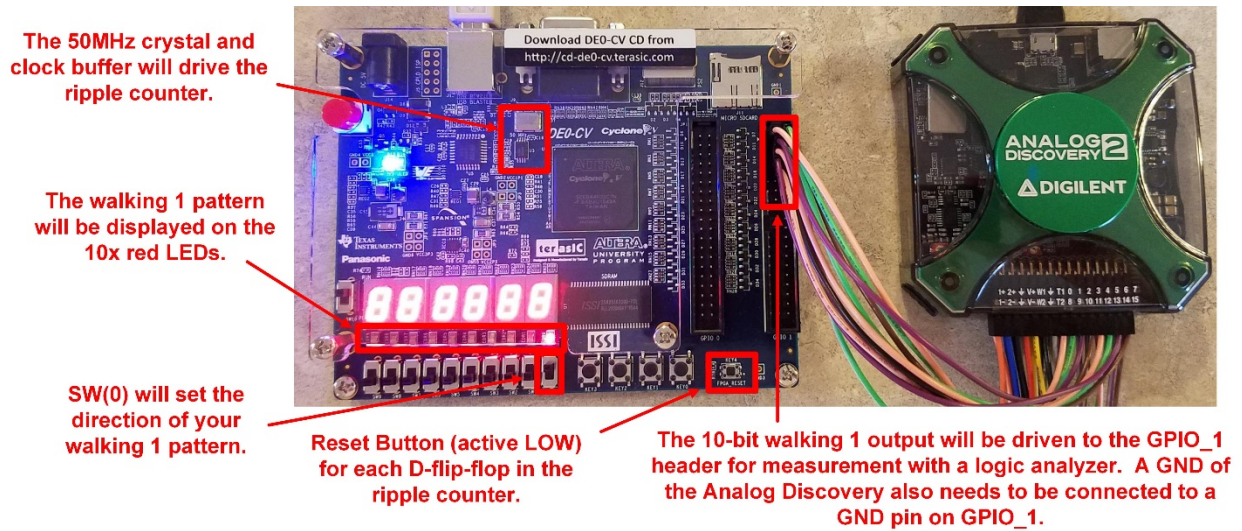


Figure 9.11
Picture of the Walking 1 System on the DE0-CV Board

9.2.5.1 Implement the Walking 1 FSM

Create a New Quartus Project by Copying Lab 9.1

We are going to use many of the same I/O that were used in lab 9.1 so we'll create a new Quartus project for this exercise by copying lab 9.1. Open lab 9.1 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_09p2_walking1_fsm”. Once again, keep in mind that when you copy the project, the pin_assignments.csv file will not automatically copy over. Manually copy this file into your new project directory. **Don't delete your ripple counter VHDL from lab 9.1.** You will be using it in this lab to produce the divided down clock.

Modify your top.vhd file Entity to Support the Additional I/O that will be Used in this Exercise

In this lab you will have a slightly different entity definition from lab 9.1. First, we want to change the name of the incoming 50 MHz clock signal to differentiate it from the internal divided down clock we will use to clock our FSM. Call the incoming clock port *Clock_50*. You also need to expand the GPIO_1 output port to 10-bits so that the entire walking 1 pattern can be measured by the logic analyzer. Once complete, your entity definition should look like Figure 9.12.

```

Text Editor - C:/Users/k91h784/Desktop/Logic_Lab/Lab_09p2_walki...
File Edit View Project Processing Tools Window Help
Search altera.com
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity top is
5     port (Clock_50 : in  std_logic;
6           Reset    : in  std_logic;
7           SW       : in  std_logic_vector (3 downto 0);
8           LEDR     : out std_logic_vector (9 downto 0);
9           HEX0     : out std_logic_vector (6 downto 0);
10          HEX1     : out std_logic_vector (6 downto 0);
11          HEX2     : out std_logic_vector (6 downto 0);
12          HEX3     : out std_logic_vector (6 downto 0);
13          HEX4     : out std_logic_vector (6 downto 0);
14          HEX5     : out std_logic_vector (6 downto 0);
15          GPIO_1   : out std_logic_vector (9 downto 0));
16 end entity;
17
100% 00:01:04

```

Figure 9.12
Entity Definition for the Ripple Counter System

[Assign the Pins for the Additional I/O used in this Exercise](#)

You will now need to update your `pin_assignments.csv` file to reflect the changes and additions for this lab. Open your `pin_assignment.csv` file. First, you will need to change the name of the incoming clock from “Clock” to “Clock_50”. Next, you will need to add the pin assignments for the additional pins that you are using on the GPIO_1 header. The pin locations can be found in the DE0-CV User’s Guide. Import the assignments into Quartus and verify they are correct by looking in the Pin Planner tool.

Note that when you change the name of a signal, it will leave the old assignment in Pin Planner. You should manually delete the signal `Clock` from the pin assignment table. Otherwise there will be a conflict with both `Clock` and `Clock_50` being assigned to `PIN_M9`. In Pin Planner, right-click on the `Clock` signal, select Edit, and then Delete. Once the assignments are verified, close the Pin Planner tool.

[Create the Internal Divided Down Clock](#)

You are going to clock your FSM at a rate that is slow enough that the walking 1 pattern can be observed on the red LEDs on the DE0-CV board. You will select one of the output bits of your ripple counter to be the internal clock. First, declare a new signal called `Clock_div` of type `std_logic`. Then assign bit 21 of your ripple counter to it. This bit will give a clock frequency of ~11.9 Hz, which is slow enough to make the walking 1 pattern visible to the human eye, but fast enough that a logic analyzer measurement won’t need to wait for too long to fill up its measurement buffer.

[Design the FSM using a Three Process Behavioral Modeling Approach](#)

Design your walking 1 FSM using a three process, behavioral modeling approach. Clock your FSM the internal `Clock_div` signal. Use the Reset input port as the active LOW reset for your FSM. Use the `SW(0)` input as the “direction” input for your FSM. When `SW(0)` is a logic 1, your pattern should move from `LEDR(0)` to `LEDR(9)` and move in the opposite direction when `SW(0)` is a logic 0. Note that the entity you copied from lab 9.1 brings in `SW` as a 4-bit vector. It is OK to leave this as is even though you are not using `SW(3)` down to 1). The synthesizer will automatically remove the unused pins from the design. Your FSM will have 10x states and traverse through them in a linear pattern with the direction dictated by `SW(0)`. Create a new 10-bit signal called `Walking1_Out` to hold the output of your FSM. Design your output logic to be of Moore-type (i.e., only depending on the current state). In each state, assert one and only bit of the 10-bit vector `Walking1_Out`. Design the outputs so that your `Walking1_Out` vector can be directly assigned to `LEDR` and produce the desired pattern on the LEDs.

[Assign the Output of your FSM to the LEDR and GPIO_1 Ports](#)

After your FSM is designed, you should make signal assignments from the output `Walking1_Out` to both `LEDR` and `GPIO_1`. Since all of these vectors are 10-bits, the assignments can be made directly.

[Compile your Design](#)

Compile your design and fix any errors that you encounter.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see a walking 1 pattern on the red LEDs. Verify that your design works as expected including the direction and reset functionality. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design on the red LEDs. **This video satisfies the requirements for deliverable #1.**

[9.2.5.2 Take a Logic Analyzer Measurement of your Walking 1 Pattern](#)

Now we want to take a logic analyzer measurement of the 10-bit walking 1 pattern on the `GPIO_1` pins. Connect the logic channels 0 → 9 of the Analog Discovery to the `GPIO_1` header as shown in [Figure 9.10](#) and [Figure 9.11](#). Make sure to connect the ground of the Analog Discovery to the `GPIO_1` header. Launch Waveforms and click on the Logic tool. In the logic analyzer tool, create a new bus called “Walking1_Out” and add channels `DIO0` → `DIO9` to the bus. Click “Add”. You are measuring a relatively slow signal so you should set the position to 0 s and the timescale to 0.2 s/div. Click on the “Single” button to run the measurement. Your measurement should look like [Figure 9.13](#). The measurement will take a few seconds to display because it is sampling at a very slow rate at this zoom setting. Take

measurements of the walking 1 pattern running in both directions to verify its functionality. Use the HotTracks tool to measure the pulse width of one of the walking 1s. It should have a pulse width of 0.084 s, which corresponds to a FSM clock rate of 11.9 Hz.

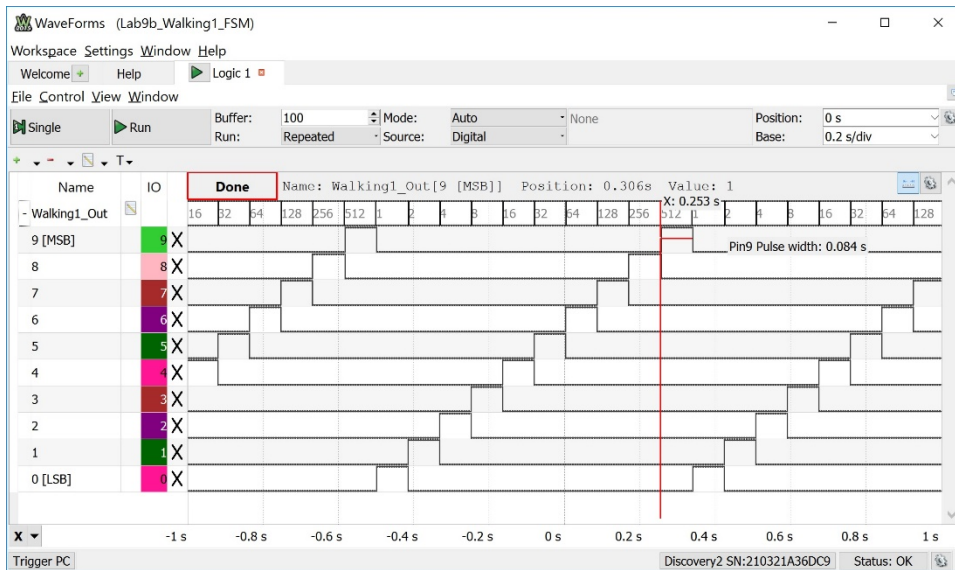


Figure 9.13
Logic Analyzer Measurement of Walking 1 Pattern

Take a screenshot of the logic analyzer measurement displaying the HotTrack pulse width value. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

Save your Analog Discovery workspace in your Quartus project directory so that you can recreate this measurement in the future if needed. Close Waveforms.

9.2.5.3 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your project using the pull-down menus: File → Close Project. Exit Quartus using the pull-down menus: File → Exit.

CONCEPT CHECK

Lab 9.2 After completing this lab exercise, can you:

- Create a FSM using the three process, behavioral modeling approach?
- Use a divided down clock in order to drive the FSM at a slower rate?
- Take logic analyzer measurement of a walking 1 pattern?

Lab 9.3: Counters using a Single Process and a 2ⁿ Clock Divider

9.3.1 Objective

The objective of this lab is to gain experience designing counters with a single process in VHDL. This lab will also give experience building a selectable, 2ⁿ clock divider based on a ripple counter. The clock divider will allow the counter to be run at a speed that can be observed on the LEDs of the DE0-CV board. This lab will also give experience using a logic analyzer to measure the frequency of a counter.

9.3.2 Learning Outcomes

After completing this lab you should be able to:

- Implement a counter using a single process in VHDL.
- Implement a selectable, 2ⁿ clock divider subsystem.
- Measure the frequency of the counter using a logic analyzer.

9.3.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

9.3.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstrate a counter implemented with a single process and displayed on the HEX displays and red LEDs of the DE0-CV board (70% of exercise).
2. A logic analyzer measurement of the frequency of the counter (20% of exercise).
3. Provide your top.vhd design file (10% of exercise).

9.3.5 Lab Work & Demonstration

You are going to design a 24-bit counter using a single process in VHDL. The 24-bits will be used to drive the six HEX character displays on the DE0-CV board through your `char_decoder.vhd` subsystem. The lower 10-bits of the counter will drive the 10x red LEDs on the DE0-CV board. The lower 8-bits of the counter will drive the GPIO_1 header on the DE0-CV board to enable a logic analyzer measurement of the counter.

You are also going to create a new subsystem that will selectively divide the frequency of the incoming 50 MHz clock so that the rate of your counter can be changed in real-time. This new system will be called “`clock_div_2ton.vhd`” and will create the internal clock (`Clock_div`) that will be used by your 24-bit counter process. [Figure 9.14](#) shows the block diagram for this exercise.

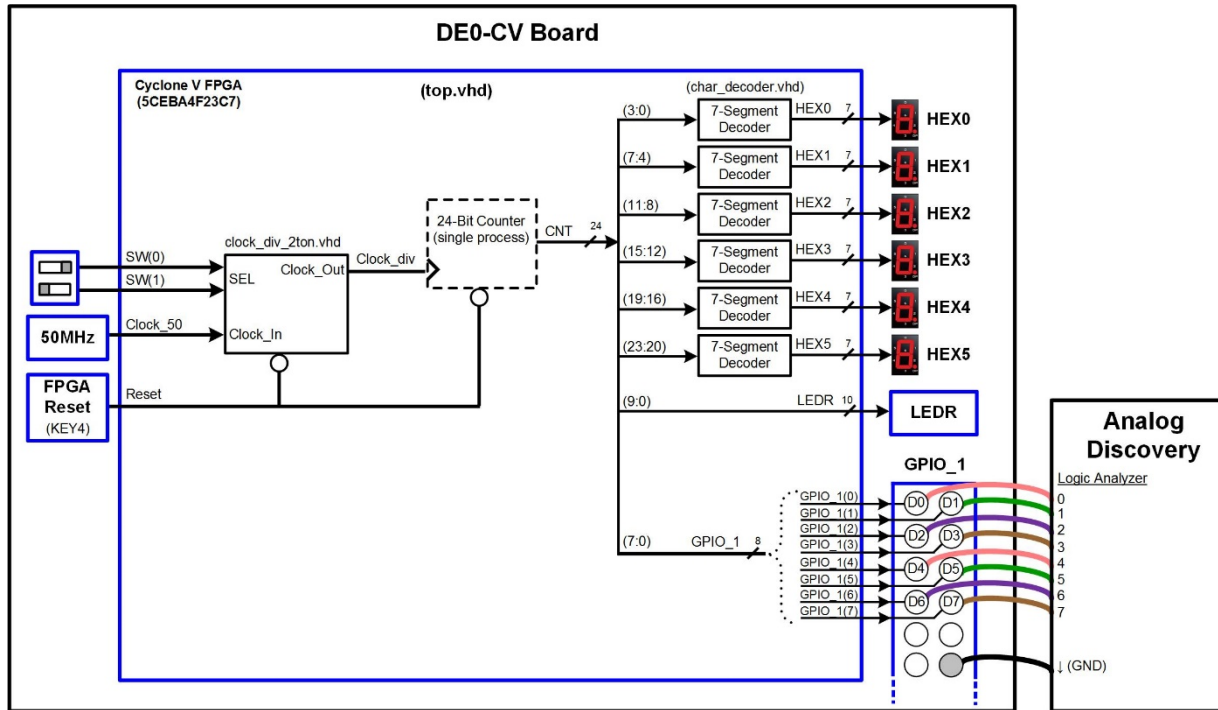


Figure 9.14
Block Diagram of the Counter System with 2^n Clock Divider

Figure 9.9 shows the I/O that will be used in this exercise.

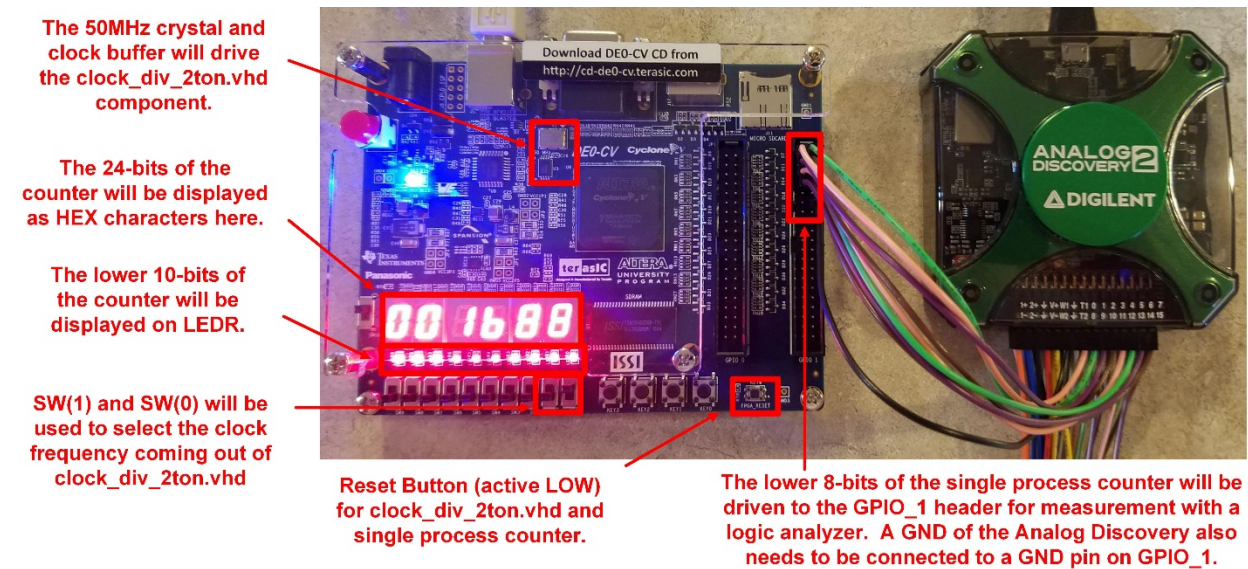


Figure 9.15
Picture of the Counter System with 2^n Clock Divider on the DE0-CV Board

9.3.5.1 Design the 24-Bit Counter and Selectable 2ⁿ Clock Divider

Create a New Quartus Project by Copying Lab 9.2

Open lab 9.2 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_09p3_counter_n_clockdiv_2ton”. Manually copy the pin_assignments.csv file into your new project directory. **Don’t delete your ripple counter VHDL from lab 9.2.** You will be moving the VHDL for your ripple counter into your clock_div_2ton.vhd later in this lab.

Add the numeric_std Library

In this exercise you will be creating a counter by taking advantage of the “+” operator. This operator is not supported in the VHDL *standard* library for the type *std_logic_vector*. It is provided in the *numeric_std* library for type *unsigned*, which can be type casted back to *std_logic_vector*. To gain access to this operator, add the *numeric_std* package to your top.vhd file.

Modify the Entity for this Exercise

In this exercise, you will be using almost the same I/O from lab 9.2 with the exception that the GPIO_1 port will be reduced to 8 bits. Modify the size of the GPIO_1 port. When done, the package and entity portion of your design should look like Figure 9.16.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5
6 entity top is
7   port (Clock_50 : in  std_logic;
8         Reset    : in  std_logic;
9         SW       : in  std_logic_vector (3 downto 0);
10        LEDR     : out std_logic_vector (9 downto 0);
11        HEX0     : out std_logic_vector (6 downto 0);
12        HEX1     : out std_logic_vector (6 downto 0);
13        HEX2     : out std_logic_vector (6 downto 0);
14        HEX3     : out std_logic_vector (6 downto 0);
15        HEX4     : out std_logic_vector (6 downto 0);
16        HEX5     : out std_logic_vector (6 downto 0);
17        GPIO_1   : out std_logic_vector (7 downto 0));
18 end entity;
19

```

Figure 9.16
Package and Entity Definition for the One-Process Counter System

Create a Selectable, 2ⁿ Clock Divider

In order to view the counter on the LEDs and character displays on the DE0-CV board with the human eye, you will need to slow down the clock frequency. The clock divider you will design in this lab will be based on your 38-bit ripple counter from prior labs. Your divider will choose one of the bits of the ripple counter to serve as the divided down clock for your overall system. Since each bit of the ripple counter produces a frequency that is ½ of the prior bit, the clock frequencies available are multiples of 2ⁿ. As such, we typically call this a “2ⁿ Clock Divider”. You are going to move your ripple counter into its own VHDL file called clock_div_2ton.vhd. The clock divider will take in the 50 MHz clock from the DE0-CV oscillator on an input port and produce one of four selectable clock frequencies (25 MHz, 191 Hz, 6 Hz, and 1.5Hz). The output clock will be selected by a multiplexer process within clock_div_2ton.vhd whose select lines come from the slider switches on the FPGA board. Figure 9.17 shows the architecture for the selectable, 2ⁿ clock divider.

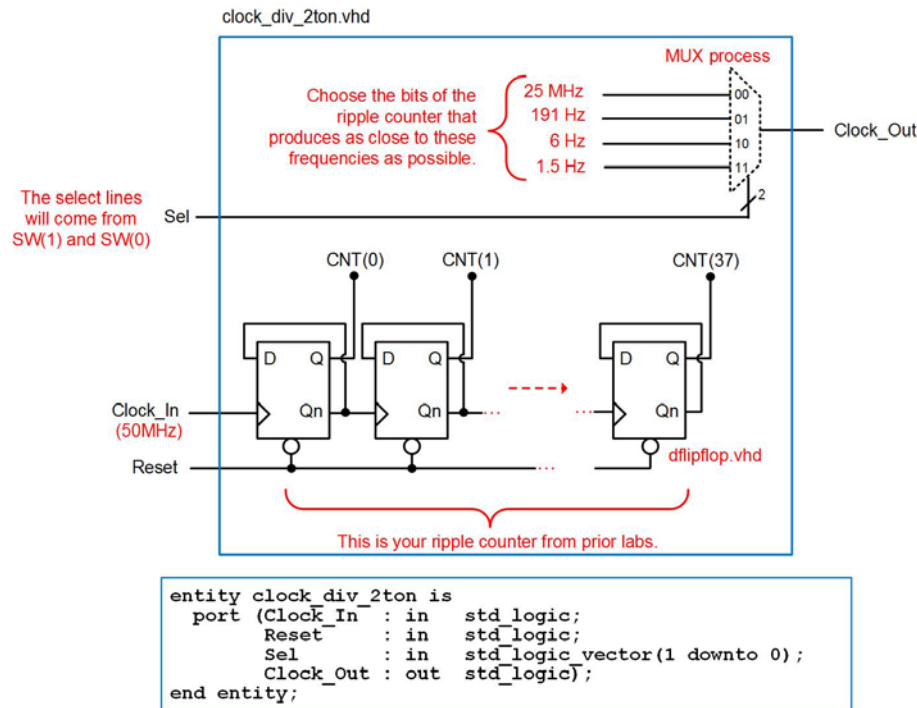


Figure 9.17
Selectable, 2^n Clock Divider Architecture

Designing the 2^n clock divider subsystem will require creating a new VHDL file in Quartus called `clock_div_2ton.vhd`. Within this new file, use the entity definition provided in [Figure 9.17](#). You will insert your 38-bit ripple counter from prior labs into this file and clock it using the input port “Clock_In”. You will then create a 4-to-1 multiplexer process that will choose one of the output bits of the ripple counter to drive the output port “Clock_Out”. The selection will be made based on the input “Sel”. Since Sel is a 2-bit vector, it can select one of four bits of the counter. Choose the counter bits for the multiplexer so that they give the clock frequencies 25 MHz, 191 Hz, 6 Hz, or 1.5 Hz based on an incoming clock of 50 MHz. Note that your clock divider requires the `dflipflop.vhd` subsystem. If this system isn’t in your project directory, add it. Save your design.

Back in your `top.vhd` file, declare your `clock_div_2ton.vhd` design as a component and instantiate it. Drive the divider component with the input port `Clock_50`. The output of the divider will be an internal signal called “Clock_div”. The signal `Clock_div` will be the signal that drives the main 24-bit counter process that you will design next. Use the input port “Reset” to drive the Reset input of your clock divider. Finally, use the `SW(1)` and `SW(0)` input ports to drive the “Sel” inputs of the clock divider.

Create the Main 24-Bit Counter using a Single Process

In your `top.vhd`, create the main 24-bit counter using a single process. The counter should be clocked with the output of the `clock_div_2ton.vhd` subsystem (i.e., `Clock_div`) and be reset using the input port “Reset”. Consider creating two internal, 24-bit signals, one of type *unsigned* (`CNT_uns`) and the other of type *std_logic_vector* (`CNT`). Create the counter process to update only `CNT_uns`. This will allow you to use the “+” operator to increment `CNT_uns` on the rising edge of the clock. Outside of the process you can type cast `CNT_uns` to *std_logic_vector* within a signal assignment from `CNT_uns` to `CNT`. This will in effect create a 24-bit counter of type *std_logic_vector*. The reason that the counter needs to be of type *std_logic_vector* is because it will be assigned to a variety of other ports in the system that are also of type *std_logic_vector*.

[Connect the 24-Bit Counter Output to a Variety of I/O on the DE0-CV](#)

Now you are going to connect the output bits of the counter to the I/O on the DE0-CV as shown in [Figure 9.14](#). You will need to instantiate 6x versions of your `char_decoder.vhd` subsystem to drive the 6x HEX displays. The inputs to the `char_decoder.vhd` instances will be groups of 4-bits from the counter. The least significant 10-bits of the counter should be connected to the 10x red LEDs (LEDR). Finally, the least significant 8-bits of the counter should be connected to 8 pins of the GPIO_1 header.

[Assign the Pins for the Additional I/O used in this Exercise](#)

Since you copied this project from lab 9.2 and are using a subset of the ports, you should not have to import any signal assignments. Launch the Pin Planner tool and verify that all of your ports have the correct assignments. If they do not, correct them.

Note that it is OK for the Pin Planner to have locations for ports that are not being used in your entity. The synthesizer will automatically remove them from your design. Once the assignments are verified, close the Pin Planner tool.

[Compile your Design](#)

Compile your design and fix any errors that you encounter.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see a counter pattern on the LEDRs and HEX displays. You should be able to change the frequency of your counter by moving the SW(1) and SW(0) switches. Verify that your design works as expected by observing the LED I/O on the DE0-CV board. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design. You should show that the counter frequency can be altered using SW(1) and SW(0). **This video satisfies the requirements for deliverable #1.**

[9.3.5.2 Take a Logic Analyzer Measurement of your Counter](#)

Now you are going to take a logic analyzer measurement of the lower 8-bits of your counter, which are being driven to the pins on the GPIO_1 header. Connect the logic channels 0 → 7 of the Analog Discovery to the GPIO_1 header as shown in [Figure 9.14](#) and [Figure 9.15](#). Make sure to connect the ground of the Analog Discovery to the GPIO_1 header. Launch Waveforms and click on the Logic tool. In the logic analyzer tool, create a new bus called “CNT” and add channels DIO0 → DIO7 to the bus. Click “Add”.

On the DE0-CV board, set the frequency of your counter to 191 Hz using the SW(1) and SW(0) switches. While there are a variety of settings that will need to be configured, go ahead and click on the “Single” button to take a measurement. After the measurement, you will see a decimal value for CNT and also some edges on the signals (7:0). You are probably zoomed out too far to see anything meaningful. Adjust the zoom by selecting different values in the *Base* setting drop-down in the upper right corner of the screen. Each time you zoom out, you’ll need to click the “Single” button to re-run the analyzer and fill the screen. At 191 Hz, a setting of 0.01 s/div is a good zoom factor. Set the trigger to look for CNT=x”00”. Now when you click the “Single” button, you’ll notice that the center of the waveform screen is the transition between 255 to 0. Keep in mind that when running at a frequency of 191 Hz, the 8-bit counter will roll over to x”00” every 1.34 seconds. This means that the analyzer will only trigger every 1.34 seconds and you’ll see a noticeable delay between when the screen is updated and when it is waiting for a trigger. Use the HotTrack tool to

measure the pulse width of the counter. With your clock divider configured to provide a 191 Hz clock to your counter, the pulse width should be around 5.2 ms. Your measurement will now look like Figure 9.18.

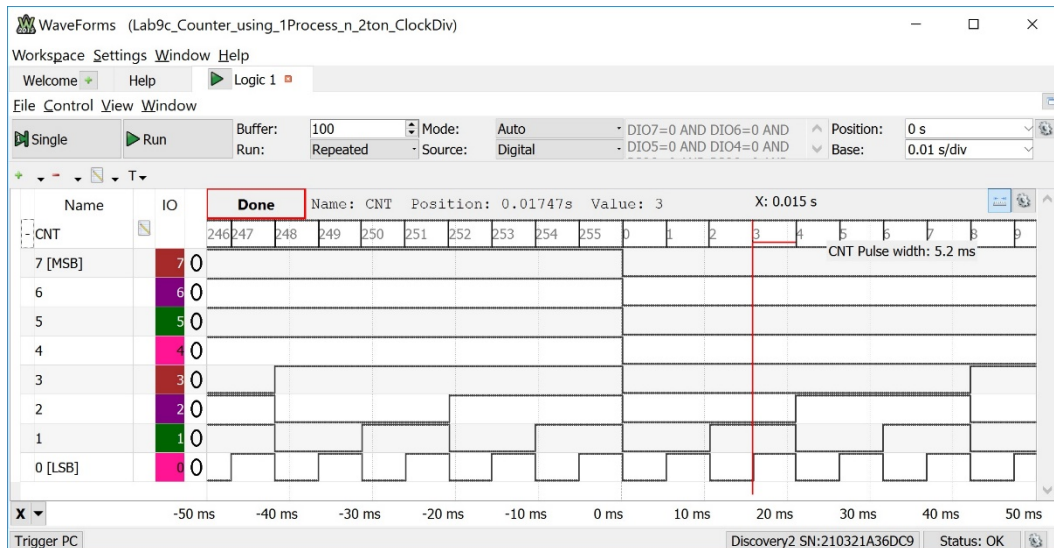


Figure 9.18
Logic Analyzer Measurement of CNT with Trigger set to x"00"

Take a screenshot of the logic analyzer measurement displaying the HotTracks measurement of your pulse width. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

Save your Analog Discovery workspace in your Quartus project directory so that you can recreate this measurement in the future if needed. Close Waveforms..

9.3.5.3 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your Waveforms and Quartus projects.

CONCEPT CHECK

Lab 9.3 After completing this lab exercise, can you:

- Implement a counter using a single process in VHDL?
- Implement a selectable, 2^n clock divider subsystem?
- Measure the frequency of the counter using a logic analyzer?

Lab 9.4: Precision Clock Divider and a BCD Counter

9.4.1 Objective

This objective of this lab is to provide more practice modeling counters in VHDL using single processes. In the first part of the lab, you will create a precision clock divider that is capable of outputting four different clock frequencies: 1 Hz, 10 Hz, 100 Hz, and 1 kHz. This will be accomplished using a counter process and selectable range checking. You will measure the frequency of the divided down clock using the Analog Discovery's oscilloscope tool. In the second part of the exercise, you will be designing a 6-symbol, binary coded decimal (BCD) counter that will be driven to the six character displays on the DE0-CV board. This will be accomplished using six separate, but interdependent processes.

9.4.2 Learning Outcomes

After completing this lab you should be able to:

- Implement a precise timing event using a counter modeled with a VHDL process and range checking.
- Create a multi-symbol BCD counter using interdependent VHDL counter processes.

9.4.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

9.4.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Oscilloscope measurement of the output of a precision clock divider (45% of the exercise).
2. Demonstration of a 6-symbol, BCD counter displayed on the character displays of the DE0-CV FPGA board (45% of exercise).
3. Provide your top.vhd design file (10% of exercise)

9.4.5 Lab Work & Demonstration

You are going to create a BCD counter that will drive the 6x HEX displays on the DE0-CV board. This system will count from 000000_{10} to 999999_{10} and then roll over. Note that the symbols are in decimal, not hexadecimal. You will be using *binary coded decimal* to implement the counter. Each symbol in the counter will be driven by its own process through its own *char_decoder.vhd* sub-system. You will also be creating a new *precision* clock divider (*clock_div_prec.vhd*) to slow down the incoming 50 MHz in order to clock the BCD counter at a slower rate. This new clock divider will allow the system to be clocked at frequencies of 1 Hz, 10 Hz, 100 Hz, and 1 kHz. The divided down clock will be observed on LEDR(0) and GPIO_1(0). To verify that the clock divider is producing the correct frequencies, the Analog Discovery's oscilloscope tool will be used to measure the clock on GPIO_1(0). [Figure 9.19](#) shows a block diagram of the system in this exercise.

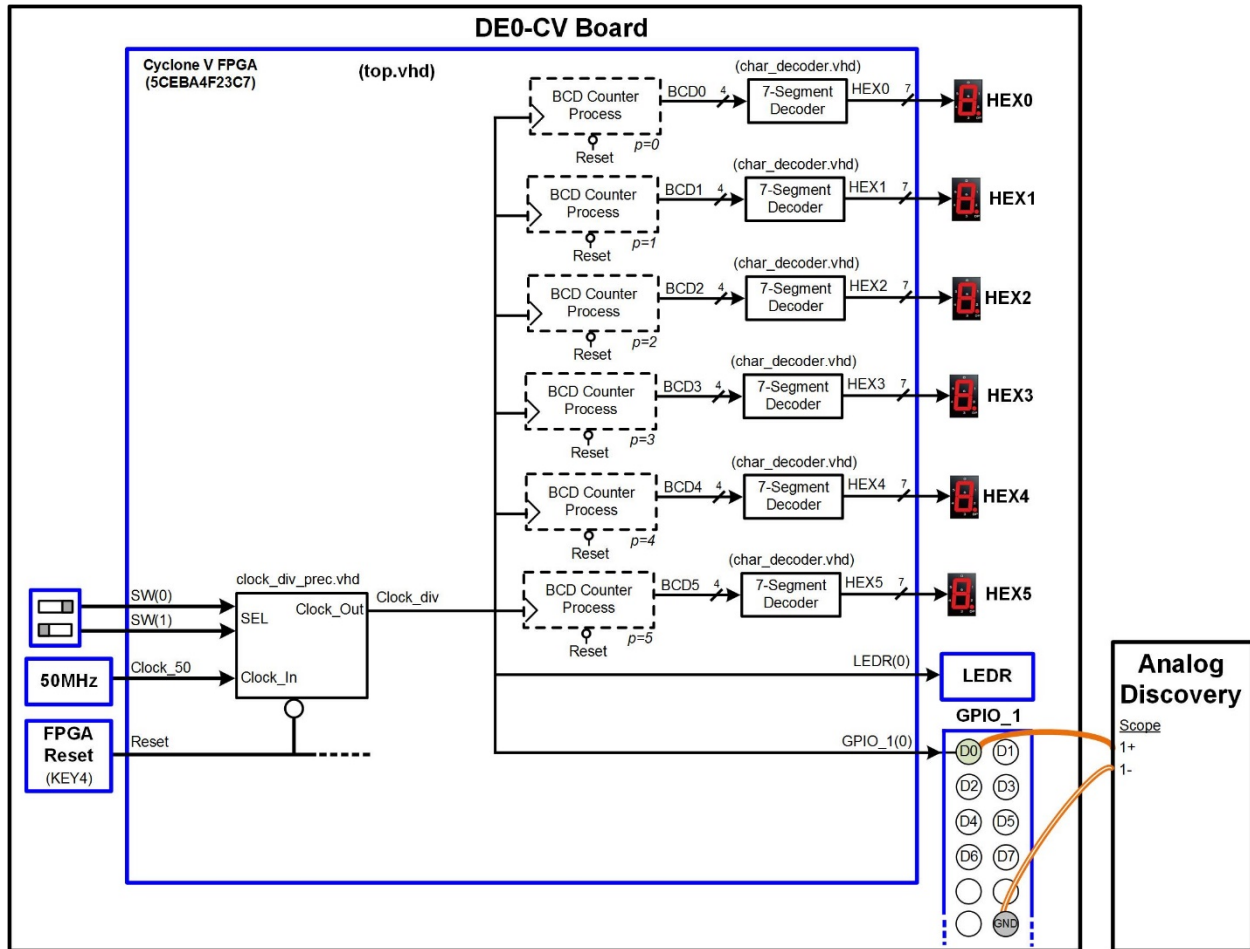


Figure 9.19
Block Diagram of the BCD Counter System

Figure 9.20 shows a picture of the I/O that will be used in this exercise.

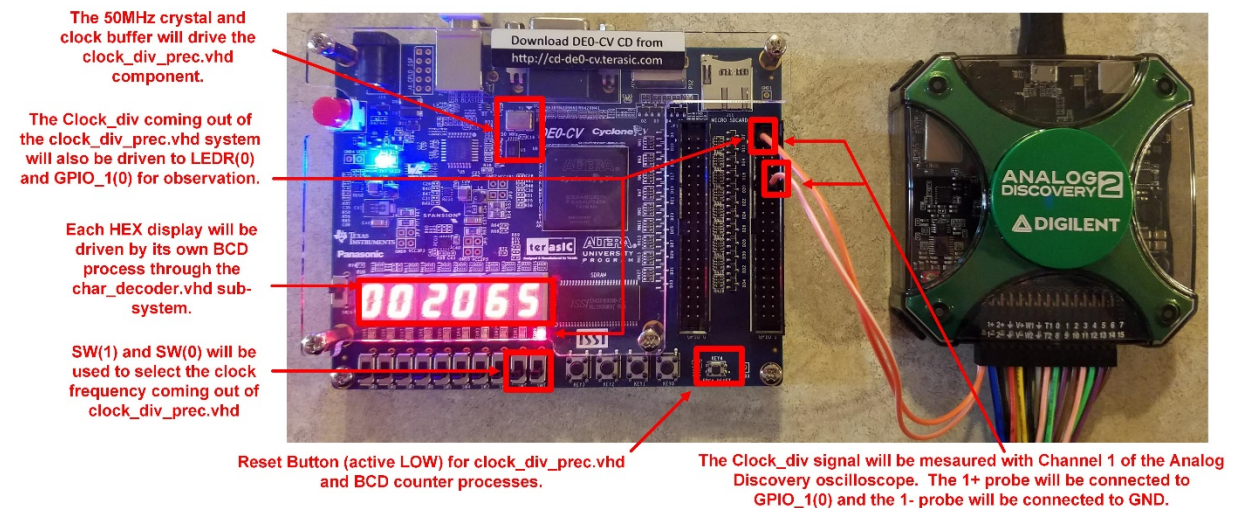


Figure 9.20
Picture of the BCD Counter System on the DE0-CV Board

9.4.5.1 Implement the Precision Clock Divider

Create a New Quartus Project by Copying Lab 9.3

Open lab 9.3 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_09p4_prec_clockdiv_n_bcd_counter”. Manually copy the pin_assignments.csv file into your new project directory. You will not need to modify the entity or packages in the top.vhd.

Create a Selectable, Precision Clock Divider

In prior labs we slowed down the system clock using a ripple counter architecture. The ripple counter architecture provided a simple way to slow down the clock using a series of D-flip-flop devices configured in toggle-flop configurations. The disadvantage of this approach is that the clock frequencies available are only in increments of 2^n . Using a 2^n clock divider approach makes it difficult to get an exact clock frequency such as 1 Hz or 10 Hz.

Another approach to creating a clock divider is using a single process counter and range checking. You can create precise timing events by simply counting up to a certain value and then setting the counter back to zero. Each time the counter reaches its maximum range, you can perform a task such as toggling a bit. This allows you to create timing events that have a precision of $\pm \frac{1}{2}$ of the period of the incoming clock.

As an example, let’s say you wanted to create a divided down clock with a frequency of 5 Mhz. This clock has a period of $T_{div}=200$ ns. Using a precision clock divider approach, we want to design a counter that toggles the output clock signal every 100 ns. Remember that to *toggle* a signal means if it was a 1, it will be changed to a 0, and if it was a 0, it will be changed to a 1. In the DE0-CV system, the incoming clock is 50 MHz, which has a period of $T_{in}=20$ ns. If you create a counter based on the incoming clock, it will increment every 20ns. If you create a counter that increments up to 5 and then is set back to 0, the setback event will occur every 5×20 ns=100ns. Each time the counter reaches its maximum value and needs to be manually set back to zero, you can also have it toggle the output clock signal. This will result in an output signal with a HIGH time of 100ns, a LOW time of 100ns, and an overall period of 200ns. In this way, you have created a divided down clock with a frequency of 5 MHz. Figure 9.21 shows a graphical depiction of this example.

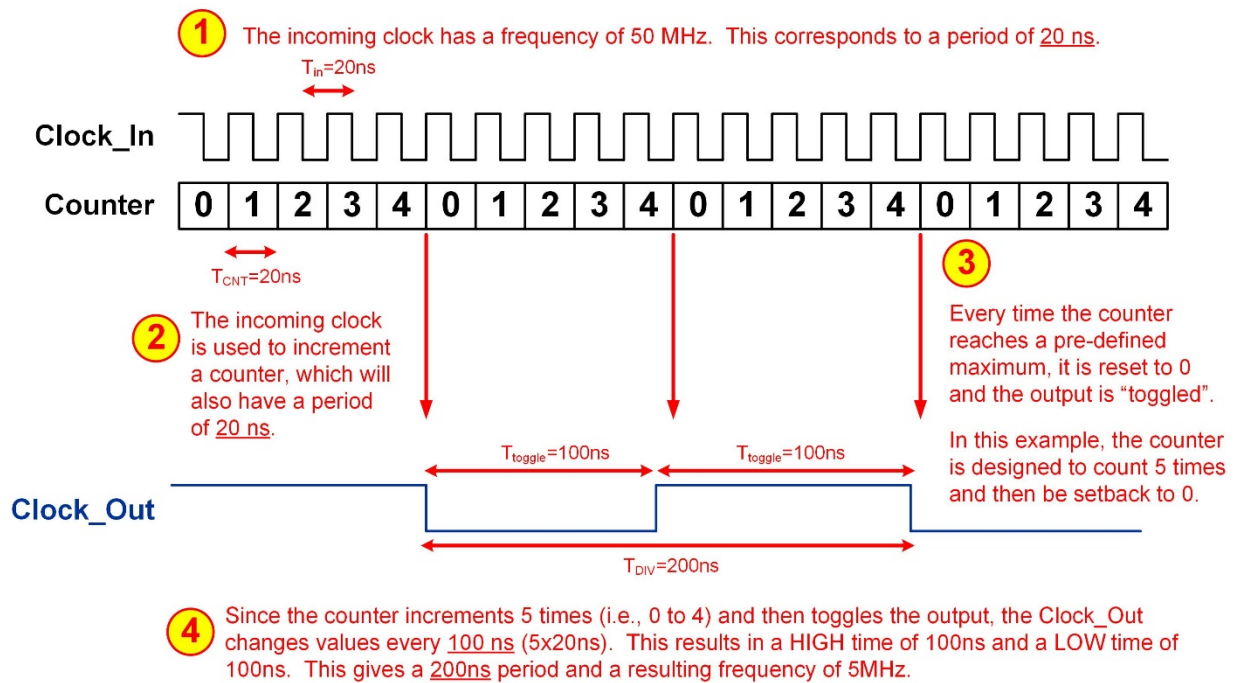


Figure 9.21
Graphical Depiction of the Operation of the Precision Clock Divider

You are going to create a precision clock divider that outputs four different clock frequencies: 1 Hz, 10 Hz, 100 Hz, and 1 kHz. This will be accomplished using the counter approach described above. The first step is to determine the maximum range of the your counter to accomplish these frequencies. Using the approach above, determine the upper values of your counter to achieve the four frequencies. Remember that when the counter reaches its maximum value, it will *toggle* the clock frequency. This means your maximum value represents how long the clock will be HIGH or LOW. Said another way, you are trying to find the number of times that the period of a 50 MHz clock (i.e., 20ns) will go into *half* of the desired clock period.

- **Maximum Value of Counter to Achieve Divided Clock of 1 Hz** = _____
- **Maximum Value of Counter to Achieve Divided Clock of 10 Hz** = _____
- **Maximum Value of Counter to Achieve Divided Clock of 100 Hz** = _____
- **Maximum Value of Counter to Achieve Divided Clock of 1k Hz** = _____

Create a new VHDL file in Quartus called `clock_div_prec.vhd`. Include the packages `std_logic_1164` and `numeric_std`. Use the following entity definition.

```
entity clock_div_prec is
  port (Clock_in : in std_logic;
        Reset    : in std_logic;
        Sel      : in std_logic_vector (1 downto 0);
        Clock_out : out std_logic);
end entity;
```

You will now enter the architecture for the precision clock divider. Your counter will output one of four different clock frequencies (1 Hz, 10 Hz, 100 Hz, and 1 kHz) depending on the values of *Sel*. You will create a process that increments a counter on the rising edge of *Clock_In* (note that *Clock_In* will be connected to *Clock_50* in the `top.vhd`). Your counter should increment up to a maximum value and then be setback to 0. Every time your counter reaches its maximum value, you should toggle *Clock_out*. The maximum value that your counter will increment to depends on the inputs *Sel*(1) and *Sel*(0). You calculated the maximum values for these four frequencies above.

Some tips:

- You are going to have a single process to implement your counter. This process will have *Clock_in* and *Reset* in the sensitivity list. Consider having a second process that will be used to assign the maximum value for the counter based on *Sel*. You will need to create an internal signal to hold the maximum value. In this way, you can simply compare your current count value to this new signal as if it was a constant. Anytime *Sel* changes, the signal will be updated independent of your main counter process. The process to make the maximum value assignment will only have *Sel* in the sensitivity list.
- Consider using the type *integer* for your counter. This way you can set the maximum value in integer format instead of hex. Since the internal counter doesn't get assigned to any ports, it does not have to be of type `std_logic_vector`.
- In your main counter process, you will want to update *Clock_out* whenever the system is reset, or when the counter reaches its maximum value. Since *Clock_out* is an output port, it can't exist on the right-hand-side of an assignment. This means you can't directly do the assignment: `Clock_out <= not Clock_out`. Consider creating an internal signal to hold the value of the divided clock. This will allow the signal to exist on the right-hand-side of the assignment. Outside of the process, you can then assign this internal signal to the output port *Clock_out*.

Save your design. Back in your `top.vhd` file, declare your `clock_div_prec.vhd` design as a component and instantiate it. Note that since you copied lab 9.2, you already have a component declaration for `clock_div_2ton.vhd` with the exact same entity definition. All you need to do is change the name of this component to `clock_div_prec` to use your new divider system.

[Connect the Clock div to LEDR\(0\) and GPIO_1\(0\)](#)

In `top.vhd`, connect the *Clock_div* signal coming out of `clock_div_prec.vhd` to *LEDR*(0) and *GPIO_1*(0). Save your design.

Compile your Design

Compile your design and fix any errors that you encounter.

Download and Test Your Design

Open the programmer tool and download your design to the FPGA. When you set SW(1) and SW(0) to either the 1 Hz or 10 Hz setting, you should see LEDR(0) blinking. When you change to the 100 Hz or 1 kHz setting, the LED will be blinking too fast to see and it will appear on.

Take an Oscilloscope Measurement of the Divided Down Clock

Configure SW(1) and SW(0) to output the 1 kHz clock rate. Now you are going to use the oscilloscope within the Analog Discovery to verify the frequency of Clock_Div on the GPIO_1(0) pin. An *oscilloscope* is an instrument that displays an electrical signal graphically. It is one of the most commonly used instruments to debug electrical circuits. The Analog Discovery contains two oscilloscope channels. Each channel has two wires (1+/1- and 2+/2-). The - channels will always be connected to ground for the exercises in this manual. Connect the probe wire labeled 1+ to the GPIO_1(0) pin on the GPIO_1 header. Connect the probe wire labeled 1- to a GND pin on the GPIO_1 header. Refer to [Figure 9.19](#) for the pin locations. Plug in the Analog Discovery to your computer using the USB cable. Launch the Waveforms application.

In the Waveforms “Welcome” tab, click on the “Scope” tool. A new tab will appear called “Scope 1”. Go to this tab and you’ll see a measurement screen. Both channels of the oscilloscope will be displayed on this screen by default. Turn off channel 2 by unchecking the box next to its zoom control on the right of the screen.

An oscilloscope sets the zoom using *divisions*. The vertical axis is always voltage, and is measured in *volts/division*, or V/div. Each line on the measurement screen is a division. The offset of the measurement can also be configured. For channel 1, set the zoom controls to:

- Offset = -1.45 V
- Range = 0.5 V/div

On the right side of the screen you’ll also see zoom controls for *time*. Again, both a scaling per division is given (base) and a horizontal offset (position). Oscilloscope are usually used for signals that are fast enough that they can’t be observed with the human eye. In this part, we will be measuring a 1 kHz signal so we need to configure the time zoom accordingly. Configure the time control to:

- Position = 0 s
- Base = 1 ms/div

When signals are too fast to be seen with the human eye, simply displaying what the oscilloscope is measuring on the screen would result in the entire screen being lit up. To handle displaying fast repetitive signals, an oscilloscope uses a *trigger*. A trigger represents an event that occurs on the incoming signal, such as a rising edge passing through a certain voltage level. When this occurs, the oscilloscope positions all of its recorded data on the screen with the trigger moment located at time=0s. As the oscilloscope continues to run, it will continually trigger and overwrite the data on the screen with the new set of data positioned with the trigger at time=0s. If the signal is repetitive, the resulting screen will show a steady waveform in which the characteristics of the signal can be determined. We want to setup the trigger so that every time Channel 1 has a rising transitions that passes through 1.7v, the oscilloscope will trigger. Along the top of the measurement screen there are a variety of trigger settings. Configure these as follows:

- Mode = Auto
- Source = Channel 1
- Condition = Rising
- Level = 1.7 V

Now we are ready to take an oscilloscope measurement. Press the “Run” button (the green triangle). You will see the waveform in [Figure 9.22](#). Note that the background can be changed from *Dark* to *Light* using the setup gear button in the upper right corner of the waveform. You can also change the thickness of the line. Making the background light and the line thicker makes taking and printing screenshots easier.

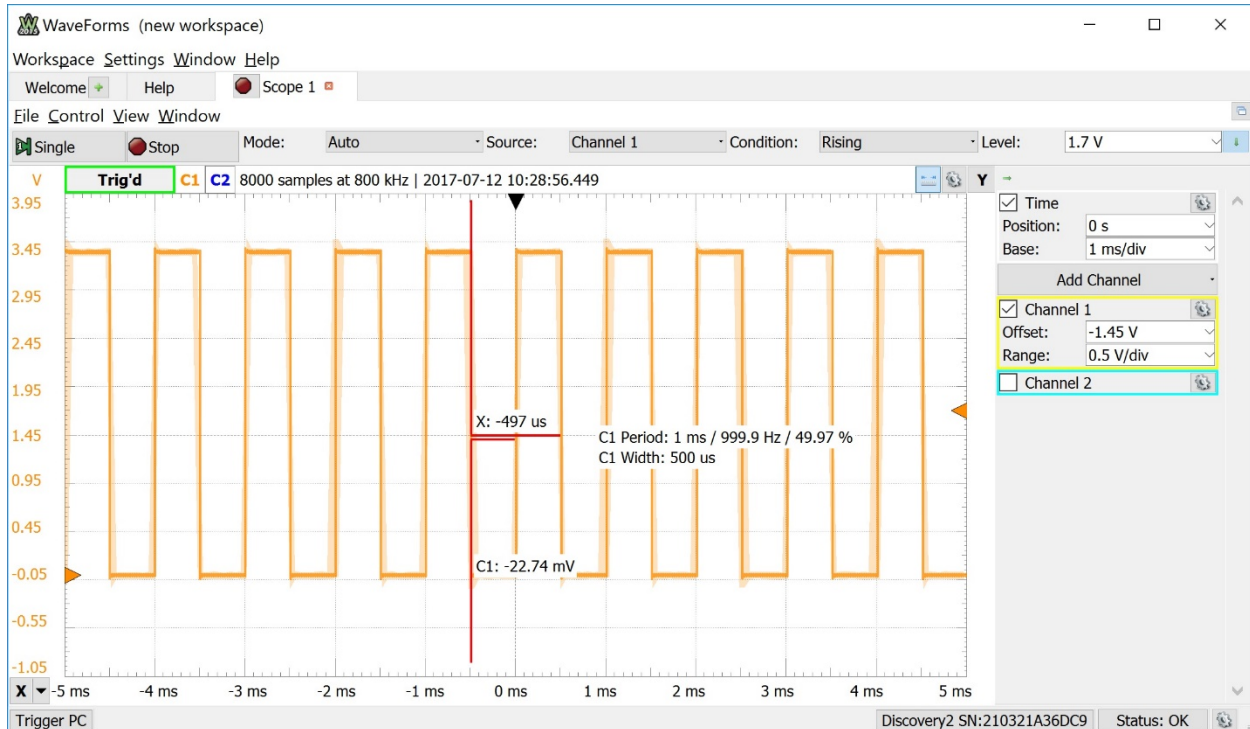


Figure 9.22
Oscilloscope Measurement of 1 kHz Divided Down Clock

Turn on the *HotTracks* measurement tool (the ruler in the upper right corner of the waveform pane). As you drag your mouse over the waveform, you'll see a variety of different measurements. When you place your marker in the vertical center of the waveform, you'll see the period measurement and corresponding frequency. If your clock divider is operating correctly, you should see a 1 kHz square wave. Verify that your clock divider works for the 100 Hz, 10 Hz and 1 Hz settings. When you change SW(1) and SW(0), you'll need to zoom in/out to get a visible square wave on the screen. Note that the 1 Hz measurement will take ~10 seconds to fill up the screen.

After verifying that all four clock frequencies operate as expected, change SW(1) and SW(0) back to the 1 kHz setting and take a screenshot showing that HotTracks is measuring a 1 kHz signal (+ or – a few Hz). Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #1.** Save your WaveForms workspace so you can recreate this measurement in the future.

9.4.5.2 Implement the 6-Digit BCD Counter

Now you are going to create the 6-digit BCD counter that will display count values from 000000_{10} to 999999_{10} on the HEX displays. A BCD code is a 4-bit value that represents a single decimal number between 0_{10} and 9_{10} . The difference between a BCD code and a standard unsigned 4-bit binary code is that the BCD code rolls over at 9_{10} instead of continuing up through the hex values A, B, C, D, E, and F. Note that to count up to 9_{10} still requires 4-bits, so a BCD number doesn't use all of the available codes that are possible. BCD codes are very popular for driving character displays because these types of displays typically only use decimal numbers, not hex. When using BCD codes to drive character displays, each character is driven by its own 4-bit BCD code.

One of the challenges of using BCD codes is that when the code is incremented, the circuitry must be able to handle rolling over back to 0_{10} (0000_2) after it reaches 9_{10} (1001_2). This is different from an unsigned 4-bit counter that rolls over from F_{16} (1111_2) to 0_{16} (0000_2) on its own. A single digit BCD counter can be easily created using a single process with range checking in VHDL. The counter increments on every rising edge of the clock, but has a nested if/else clause that checks whether it has reached its maximum value. When it has, it sets the value back to zero instead

of letting it go higher than 9. Refer to section 9.4.2 in the textbook to see an example of a counter implemented with an integer data type and range checking.

Another complexity of BCD counters comes when more than one digit is used. Each digit is driven with its own process, but each higher position process depends on the lower position values relative to its own position. There are two considerations when implementing the process for a digit that isn't in the lowest position. The first consideration is when to set it back to 0. It does not simply setback to 0 when it reaches its maximum value. Instead, it is only setback to 0 when itself and all lower position digits are at their maximum value. As an example, let's look at a 2-digit number. When this number reaches 90_{10} , it does not set the digit in $p=1$ back to zero, even though it has reached its maximum value of 9_{10} . Instead, it is setback to zero when it reaches 99_{10} because itself and all lower position digits have reached their maximum values. This is handled within the process in the same way that the maximum range is handled, except that the condition for when to set it back to 0 must include all lower position values. This can be handled by including a Boolean AND operation within the *if* clause for the range checking.

The second consideration is that the higher order digit doesn't increment on every edge of the triggering clock. Instead, it only increments when all of the lower position digits have reached their maximum value. As an example, let's again consider a 2-digit decimal number. For the first 9 counts, only the least significant digit is incrementing ($00_{10} \rightarrow 01_{10} \rightarrow 02_{10} \rightarrow \dots$). Only when the least significant digit reaches its maximum value of 9_{10} and is setback to 0 does the digit in the higher position increment ($08_{10} \rightarrow 09_{10} \rightarrow 10_{10}$). This means the process implementing the incrementing behavior of the counter must have an additional *if* clause that checks whether the lower position bits are at their maximum values. Only if they are is the higher order counter incremented.

Implement the 6x separate processes for the BCD counter. It is recommended that you implement and test each process before moving to the next higher order digit. The counter result of each process will be driven into the `char_decoder.vhd` components in order to drive each HEX display.

[Compile your Design](#)

Compile your final design and fix any errors that you encounter.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see a counter pattern on the HEX displays. You should be able to change the frequency of your counter by moving the SW(1) and SW(0) switches. Fix any errors you discover. Take a short video (<5 s) showing the proper operation of your design. You should show that the counter frequency can be altered using SW(1) and SW(0). **This video satisfies the requirements for deliverable #2.**

[9.4.5.3 Save a Copy of your top.vhd for your Records](#)

Locate the `top.vhd` file for this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your Waveforms and Quartus projects.

CONCEPT CHECK

Lab 9.4 After completing this lab exercise, can you:

- Implement a counter using a single process in VHDL?
- Implement a selectable, 2^n clock divider subsystem?
- Measure the frequency of the counter using a logic analyzer?

Chapter 10: Memory

Lab 10.1: ROM Memory

10.1.1 Objective

This objective of this lab is to gain experience modeling read only memory (ROM) in VHDL. You will design a synchronous, 64x8 ROM array and pre-populate it with known values. You will then create an address counter that will cycle through all 64 addresses in order to continually read the contents of the ROM. You will observe the address and output of the ROM on the HEX displays and using the logic analyzer.

10.1.2 Learning Outcomes

After completing this lab you should be able to:

- Implement a synchronous, 64x8 ROM array with pre-populated values.
- Implement a system that continually reads the contents of the ROM using an address counter.
- Observe the ROM system operation using a logic analyzer.

10.1.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

10.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstration of a ROM system that continually reads the values of the array and displays the input address and output data on the HEX displays (70% of exercise).
2. A logic analyzer measurement showing the operation of the ROM system (20% of exercise).
3. Provide your top.vhd design file (10% of exercise).

10.1.5 Lab Work & Demonstration

You are going to create a synchronous, 64x8 ROM array (`rom_64x8_sync.vhd`) and populate it with known values. The ROM component will be instantiated in your `top.vhd` where you will drive its address input with an address counter process. The address counter and ROM will be clocked with a divided down clock coming from your `clock_div_prec.vhd` component. The address will be displayed on the HEX5 and HEX4 displays through your `char_decoder.vhd` component. The 8-bit output of the ROM array will be displayed on the HEX1 and HEX0 displays through your `char_decoder.vhd`. The HEX3 and HEX2 displays will be turned *off* by driving constant “1111111” vectors to each display. You will drive the address, output of the ROM, and divided clock to the GPIO_1 header for observation with the logic analyzer. [Figure 10.1](#) shows a block diagram of the ROM system.

10.1.5.1 Implement the ROM System

[Create a New Quartus Project by Copying Lab 9.4](#)

Open lab 9.4 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_10p1_ROM_memory”. Manually copy the pin_assignments.csv file into your new project directory.

[Modify the Entity for this Exercise](#)

In this exercise, you will be using almost the same I/O from lab 9.4 with the exception that the GPIO_1 port will be expanded to 15 bits. Modify the size of the GPIO_1 port. When done, the package and entity portion of your design should look like [Figure 10.3](#).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5
6 entity top is
7   port (Clock_50 : in  std_logic;
8         Reset    : in  std_logic;
9         SW       : in  std_logic_vector (3  downto 0);
10        LEDR    : out std_logic_vector (9  downto 0);
11        HEX0    : out std_logic_vector (6  downto 0);
12        HEX1    : out std_logic_vector (6  downto 0);
13        HEX2    : out std_logic_vector (6  downto 0);
14        HEX3    : out std_logic_vector (6  downto 0);
15        HEX4    : out std_logic_vector (6  downto 0);
16        HEX5    : out std_logic_vector (6  downto 0);
17        GPIO_1  : out std_logic_vector (14 downto 0));
18 end entity;
19

```

Figure 10.3
Entity for ROM System

[Create a Synchronous, 64x8 ROM](#)

You are going to create a model for a synchronous, 64x8 ROM array. This will require creating a new VHDL file in Quartus. Your file should be called “rom_64x8_sync.vhd”. Since there are 64 address locations, the ROM will require 6x address lines ($2^6=64$). The output *data_out* will be 8-bits wide. The output should be updated with the contents located at the provided address on the rising edge of the clock. Since this is a memory device, there is not a reset like in a D-flip-flop storage device. Refer to example 10.3 in the textbook for details on how to model a ROM in VHDL. [Figure 10.4](#) shows the entity definition and contents that should be placed in the ROM array.

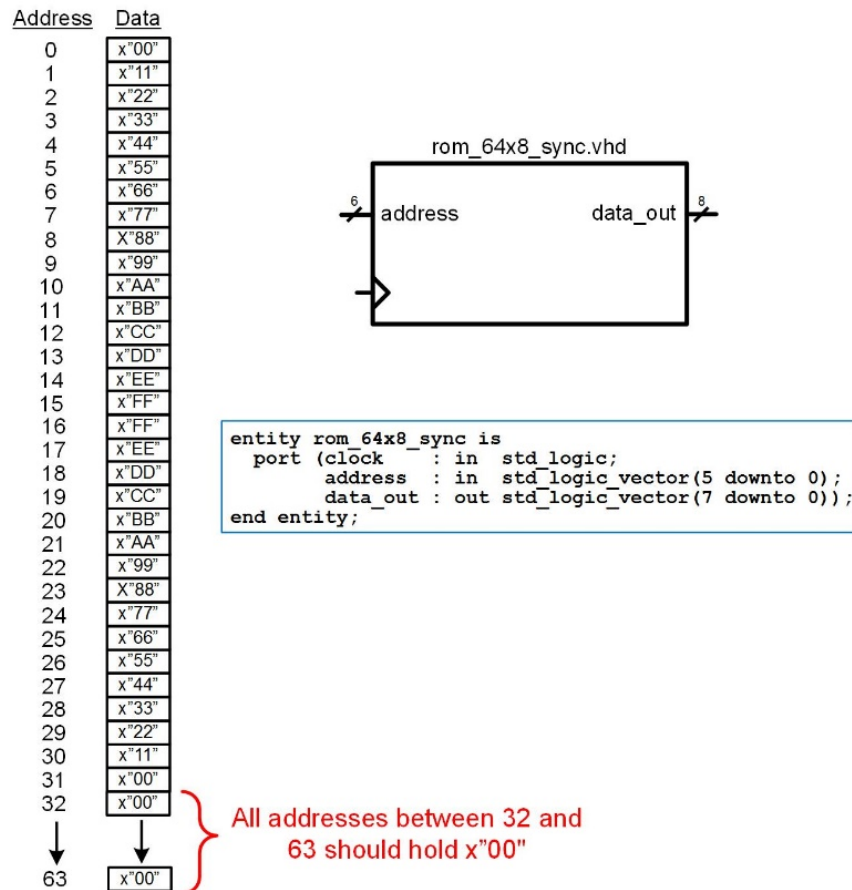


Figure 10.4
ROM Array Contents and Entity Definition

Back in your `top.vhd` file, declare your `rom_64x8_sync.vhd` design as a component and instantiate it. You should create two internal signal vectors called `address` and `ROM_data_out` to connect to the component. You will drive the component's clock with `Clock_div` coming out of your `clock_div_prec.vhd`.

Create the Address Counter

In your `top.vhd`, create an address counter using a single process. The counter should increment from 0 to 63 and then roll over continuously. The counter will increment on the rising edge of `Clock_div` and reset based on the input port `Reset`.

Connect the ROM Signals to a Variety of I/O on the DE0-CV

First, you are going to observe the operation of the ROM on the HEX displays. The address will be displayed on HEX5 and HEX4. Since the address counter is only 6-bits wide, the upper two bits will need to be concatenated with "00" when connected to the HEX5 `char_decoder.vhd`. The `ROM_data_out` vector will be displayed on HEX1 and HEX0. This will require 4x instantiations of your `char_decoder.vhd` component. You should turn off the HEX3 and HEX2 displays by driving them directly with 1's.

In the second part of this exercise, you'll observe the ROM operation on the logic analyzer. Connect `ROM_data_out` to the GPIO_1 header pins D0 → D7. Connect `address` to the GPIO_1 header pins D8 → D13. Connect `Clock_div` to the GPIO_1 header pin D14.

[Assign the Pins for the Additional I/O used in this Exercise](#)

You will need to provide pin locations for the additional GPIO_1 signals used in this exercise. Locate these in the DE0-CV user's manual. Update your pin_assignments.csv file. Import the assignments into Quartus.

[Compile your Design](#)

Compile your design and fix any errors that you encounter.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see the address and ROM_data_out on the HEX displays. You can set the clock frequency to 1 Hz to verify that each location in the ROM array contains the correct data. Keep in mind that addresses 32 → 63 contain 0's. You'll notice that the data is one clock behind the address. This is because it takes a finite amount of time for the address to be produced after the clock edge so it is not visible to other systems on that same edge. It is on the *next* clock edge that the ROM is able to see the address that was produced and output the corresponding data. Verify that your design works as expected including the reset function. Fix any errors you discover. Set the clock to **10 Hz**. Take a short video (<5 s) showing the proper operation of your design. **This video satisfies the requirements for deliverable #1.**

10.1.5.2 [Take a Logic Analyzer Measurement of your ROM System](#)

Now you are going to take a logic analyzer measurement of the ROM system. Connect the logic channels 0 → 14 of the Analog Discovery to the GPIO_1 header as shown in [Figure 10.1](#) and [Figure 10.2](#). Make sure to connect the ground of the Analog Discovery to the GPIO_1 header. Launch Waveforms and click on the Logic tool. In the logic analyzer tool, add new busses for “address” and “ROM_data_out” and assign the logic channels accordingly. Display the address in **decimal** format. Add a new signal for Clock_div and assign the logic channel accordingly.

On the DE0-CV board, set the frequency of your counter to **1 kHz** using the SW(1) and SW(0) switches. While there are a variety of settings that will need to be configured, go ahead and click on the “Single” button to take a measurement. After the measurement, you will see values for address, ROM_data_out, and Clock_div. Set the Position to 0.02 s and the Base to 5 ms/div. Set the trigger to address="000000". Run the logic analyzer continuously. You should see the contents of the ROM being read out. Your measurement should look like [Figure 10.5](#).

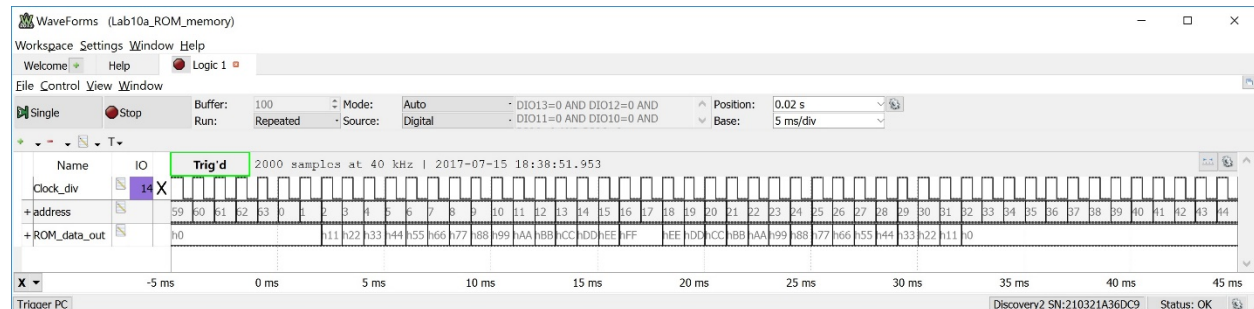


Figure 10.5
Logic Analyzer Measurement of ROM System

Take a screenshot of the logic analyzer measurement. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

Save your Analog Discovery workspace in your Quartus project directory so that you can recreate this measurement in the future if needed. Close Waveforms.

10.1.5.3 [Save a Copy of your top.vhd for your Records](#)

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #3.**

After you are done, close your Quartus project.

CONCEPT CHECK

Lab 10.1 After completing this lab exercise, can you:

- Implement a synchronous, 64x8 ROM array with pre-populated values?
- Implement a system that continually reads the contents of the ROM using an address counter?
- Observe the ROM system operation using a logic analyzer?

Lab 10.2: Read/Write Memory

10.2.1 Objective

This objective of this lab is to gain experience modeling read/write memory (R/W) in VHDL. You will design a synchronous, 32x8 R/W memory. You will then create an address counter with enable that will provide the 32 addresses for the array. You will create a control finite state machine that will facilitate writing information to the memory and also reading information to verify it is storing correctly. You will observe all of the critical signals using the logic analyzer.

10.2.2 Learning Outcomes

After completing this lab you should be able to:

- Implement a synchronous, 32x8 R/W array.
- Implement an address counter with enable.
- Implement a FSM that controls how information is written to, and read from, the R/W array.
- Observe the R/W system operation using a logic analyzer.

10.2.3 Parts Needed

- DE0-CV FPGA board.
- Analog Discovery 2.

10.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstration of your R/W system displaying read and write operations on the HEX displays and red LEDs (50% of exercise).
2. A logic analyzer measurement showing a write operation to the R/W array (20% of exercise).
3. A logic analyzer measurement showing a series of read operations from the R/W array (20% of exercise).
4. Provide your top.vhd design file (10% of exercise).

10.2.5 Lab Work & Demonstration

You are going to create a synchronous, 32x8 R/W array (`rw_32x8_sync.vhd`). The R/W component will be instantiated in your `top.vhd` where you will drive its inputs using the slider switches SW(9 down to 2). A FSM will control writing to, and reading from, the R/W array. An address counter with enable will provide the address to the R/W array. The address will be displayed on the HEX5 and HEX4 displays through your `char_decoder.vhd` component. The 8-bit output of the R/W array will be displayed on the HEX1 and HEX0 displays through your `char_decoder.vhd`. The HEX3 and HEX2 displays will be turned *off* by driving constant “1111111” vectors to each display. You will drive the address, output of the R/W array, the divided clock, the address counter enable, and the memory write signals to the GPIO_1 header for observation with the logic analyzer.

When KEY(0) is pressed, the information on the slider switches will be written to the next address in the R/W array. The control FSM will enable the address counter for one increment and then assert the write signal for the memory. In this way, information can be populated into the R/W array by setting values on the switches and continually pressing KEY(0). When KEY(1) is pressed, the address counter will run freely in order to output the data at each location within the R/W array. In this way, the contents of the array can be verified to ensure that the write operations were successful. All critical signals will be connected to pins on the GPIO_1 header for observation with the logic analyzer. [Figure 10.6](#) shows a block diagram of the R/W system.

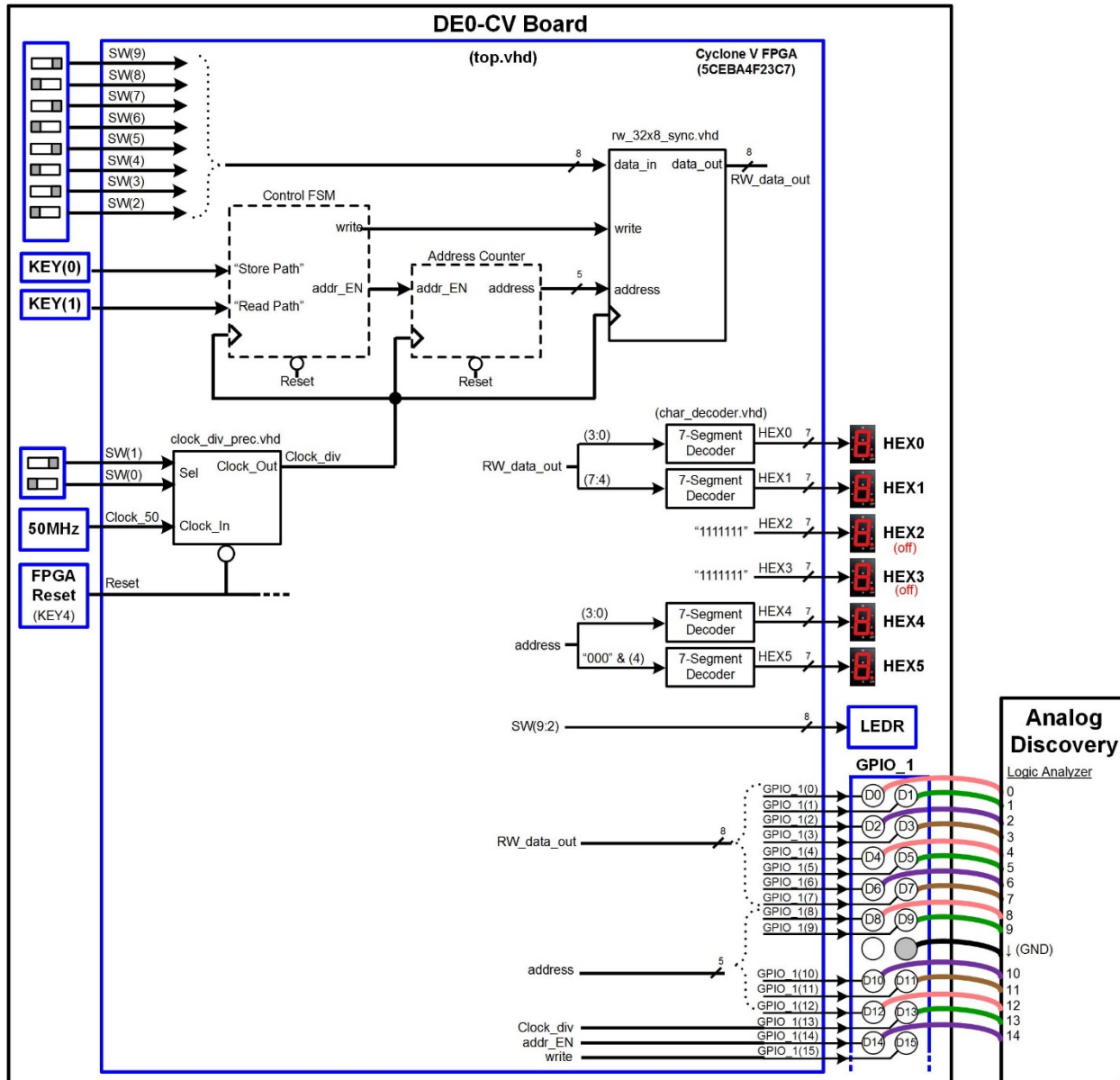


Figure 10.6
Block Diagram of the R/W Memory System

Figure 10.7 shows a picture of the R/W memory system implemented on the DE0-CV board.

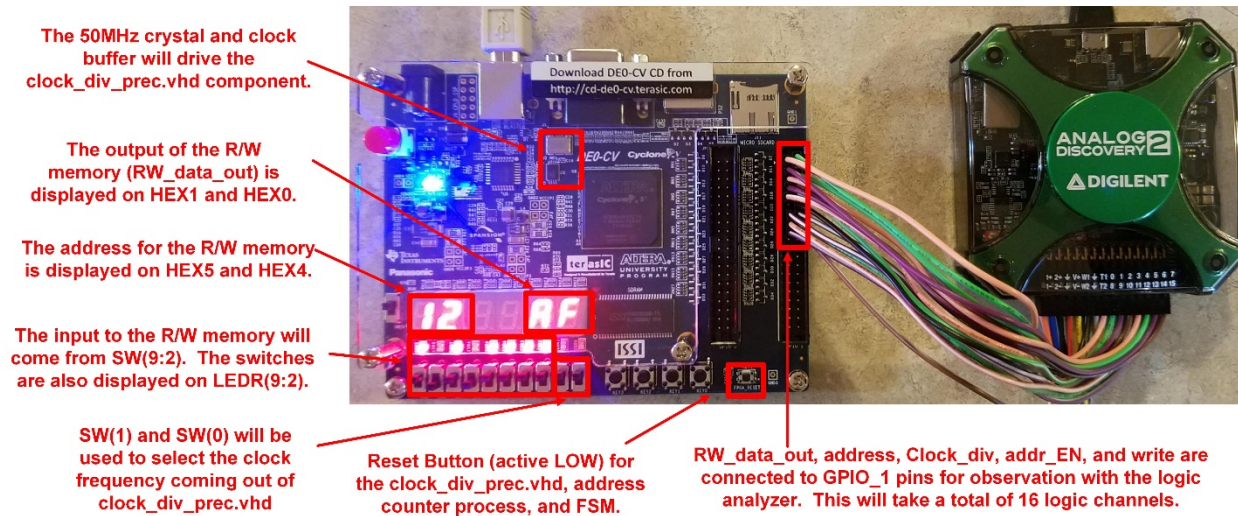


Figure 10.7
Picture of the R/W Memory System on the DE0-CV Board

10.2.5.1 Implement the R/W Memory System

Create a New Quartus Project by Copying Lab 10.1

Open lab 10.1 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project "Lab_10p2_RW_memory". Manually copy the `pin_assignments.csv` file into your new project directory.

Modify the Entity for this Exercise

In this exercise, you will add a new input port called KEY. This will be used to read from the active LOW push button switches on the DE0-CV board. This port will be 2-bits wide to support reading from the KEY(0) and KEY(1) buttons. You will also need to expand the width of the SW port to 10-bits to handle reading from all 10x slider switches. Finally, you will need to expand the GPIO_1 port to 16-bits to handle the logic analyzer measurement of all of the critical signals in the system. When done, the package and entity portion of your design should look like [Figure 10.8](#).

```

Text Editor - C:/Users/k91h784/Desktop/Logic_Lab/Lab_10p2_RW_...
File Edit View Project Processing Tools Window Help Search altera.com
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5
6 entity top is
7   port (Clock_50 : in  std_logic;
8         Reset    : in  std_logic;
9         SW       : in  std_logic_vector (9 downto 0);
10        KEY      : in  std_logic_vector (1  downto 0);
11        LEDR    : out std_logic_vector (9  downto 0);
12        HEX0    : out std_logic_vector (6  downto 0);
13        HEX1    : out std_logic_vector (6  downto 0);
14        HEX2    : out std_logic_vector (6  downto 0);
15        HEX3    : out std_logic_vector (6  downto 0);
16        HEX4    : out std_logic_vector (6  downto 0);
17        HEX5    : out std_logic_vector (6  downto 0);
18        GPIO_1  : out std_logic_vector (15 downto 0));
19 end entity;
20
21
0% 00:00:14

```

Figure 10.8
Entity for R/W Memory System

Create a Synchronous, 32x8 Read/Write Memory

You are going to create a model for a synchronous, 32x8 R/W memory array. This will require creating a new VHDL file in Quartus. Your file should be called “rw_32x8_sync.vhd”. Since there are 32 address locations, the R/W memory will require 5x address lines ($2^5=32$). The system will have an 8-bit input called *data_in* and an 8-bit output called *data_out*. There is also an input called *write*. When write is asserted, the information on *data_in* will be stored to the input address location on the rising edge of the clock. The system will continually output the data at the provided address on *data_out* on every rising edge of clock. Refer to example 10.5 in the textbook for details on how to model a ROM in VHDL. Figure 10.9 shows the ports and entity definition for the R/W memory array.

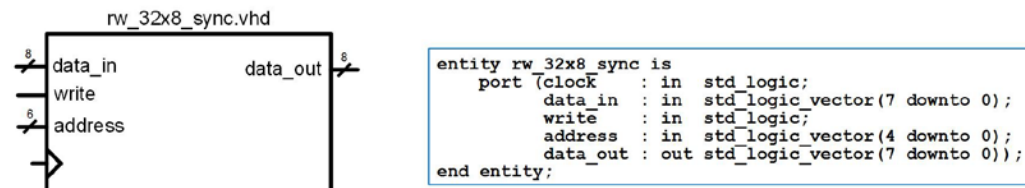


Figure 10.9
R/W Ports and Entity Definition

Back in your top.vhd file, declare your rw_32x8_sync.vhd design as a component and instantiate it. You should create an internal signal called *write* and two internal signal vectors called *address* and *RW_data_out* to connect to the component. The *data_in* port of the rw_32x8_sync.vhd component will be connected to SW(9 downto 2). You will drive the component’s clock with *Clock_div* coming out of your clock_div_prec.vhd.

Create the Address Counter

In your top.vhd, create an address counter using a single process. The counter should have a synchronous, active HIGH enable. Create an internal signal called *addr_EN* to serve as the enable line. When *addr_EN* is asserted, the counter will increment on the rising edge of *Clock_div*. When *addr_EN* is deasserted, the counter should hold its current value. The counter will increment from 0 to 31 and then roll over. The counter should have an asynchronous, active LOW reset.

Create the Control FSM

The R/W operation will be facilitated by a control FSM. The state diagram for the FSM is given in Figure 10.10. You will implement the FSM using the three process, behavioral modeling approach described in Chapter 8 of the textbook. The FSM will have two primary functions. The first is to handle storing the values of SW(9 downto 2) to the R/W memory. When KEY(0) is pressed, the FSM will increment the address counter by asserting *addr_EN* for one clock cycle. It will then assert *write* for one clock cycle. This will have the effect of storing the values on the switches to the *next* address in R/W memory. When running the clock at 1 kHz, you will not be able to press and release KEY(0) before the FSM returns to its START state and looks for another key press. This means that a single button press could result in hundreds of writes to memory. To avoid this behavior, the FSM will have a state that waits for KEY(0) to be released. In this way, each KEY(0) press only results in one write operation to R/W memory. This functionality is labeled the “Store Path” in Figure 10.10. Note that the push buttons are active LOW, meaning that when not pressed, they produce a logic 1 and when pressed they produced a logic 0.

The second function that the FSM will handle is reading out the contents of memory. When KEY(1) is pressed, the FSM will assert *addr_EN* for as long as the button is held down. In this way, the address counter will increment on every rising edge of clock. This will result in the R/W array outputting the contents of each address on the rising edge of clock. This functionality is labeled the “Read Path” in Figure 10.10.

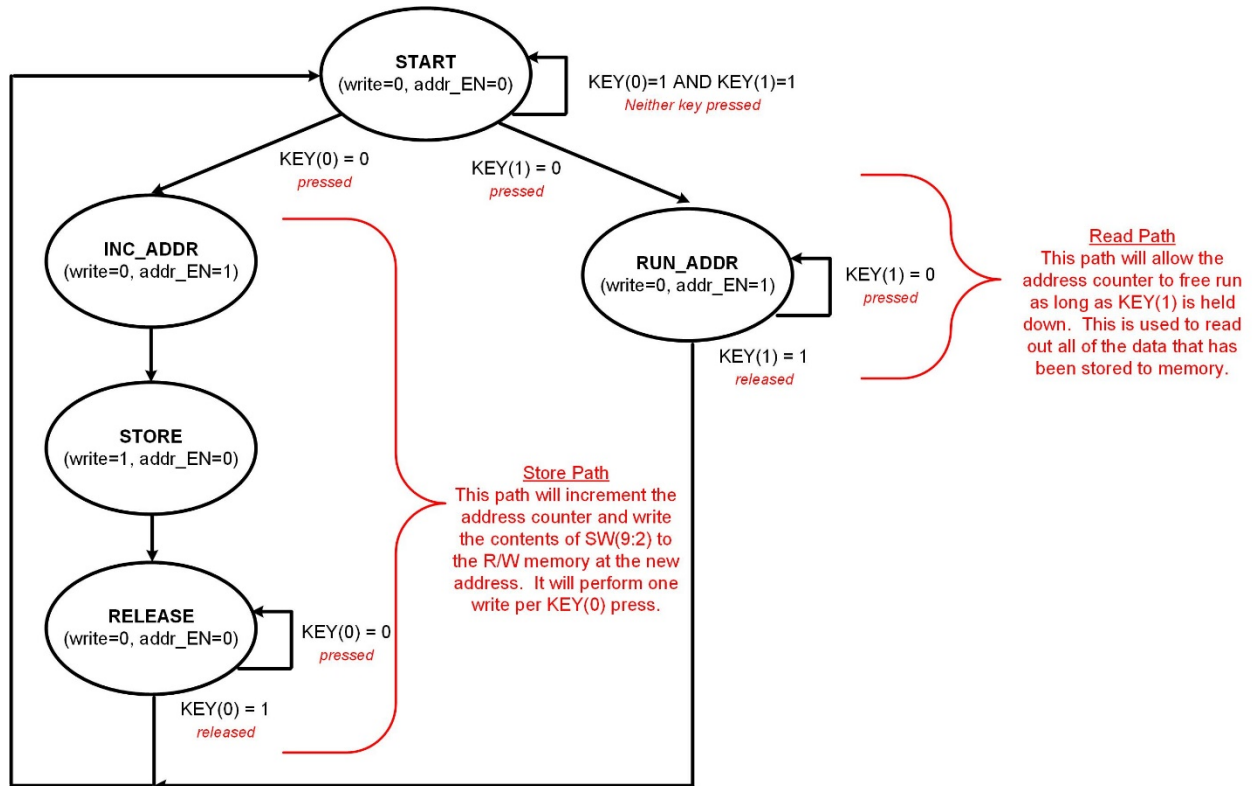


Figure 10.10
State Diagram for R/W Memory Control FSM

[Connect the R/W Signals to a Variety of I/O on the DE0-CV](#)

First, you are going to observe the operation of the R/W memory address and data out on the HEX displays. The address will be displayed on HEX5 and HEX4. Since the address counter is only 5-bits wide, the upper bit will need to be concatenated with “000” when connected to the HEX5 char_decoder.vhd. The RW_data_out vector will be displayed on HEX1 and HEX0. This will require 4x instantiations of your char_decoder.vhd component. You should turn off the HEX3 and HEX2 displays by driving them directly with 1’s.

You will also observe the R/W memory operation on the logic analyzer. Connect RW_data_out to the GPIO_1 header pins D0 → D7. Connect address to the GPIO_1 header pins D8 → D12. Connect Clock_div, addr_EN, and write to the GPIO_1 header pins D13, D14, and D15 respectively.

Connect SW(9 down to 2) to LEDR(9 down to 2) so that the input data to the R/W is visible.

[Assign the Pins for the Additional I/O used in this Exercise](#)

You will need to provide pin locations for the KEY(0) and KEY(1) inputs and the additional GPIO_1(15) pin used in this exercise. Locate these in the DE0-CV user’s manual. Update your pin_assignments.csv file. Import the assignments into Quartus.

[Compile your Design](#)

Compile your design and fix any errors that you encounter.

[Download and Test Your Design](#)

Open the programmer tool and download your design to the FPGA. You should now see the address and RW_data_out on the HEX displays. You should also see the values of the upper eight slider switches on the upper 8 red LEDs. After the program is downloaded, the system will show address=x”00” and RW_data_out=”00”.

Set the clock frequency to **10 Hz** so that the system runs fast enough to quickly respond to a button press but slow enough so that the values can be observed on the HEX displays. Set SW(9 downto 2)=x"AA" and press KEY(0). You will see the address increment to x"01" and RW_data_out change to x"AA". You have just stored x"AA" at address location x"01" in the R/W array. There are a few things to consider about this operation. First, there is a delay between when the address increments and when RW_data_out updates. This is because it takes a few clock cycles to complete the write. The FSM first increments the address to x"01" by asserting *addr_EN*. It then asserts *write* to store the information. The data appears on RW_data_out one clock cycle after it was written because the write and read can't occur simultaneously in the memory system.

Change the slider switches to x"EE" and press KEY(0). You will see the address increment to x"02" and RW_data_out change to x"EE". You have just stored x"EE" at address location x"02" in the R/W array. Continue to write different data to the R/W array by changing the switches and pressing KEY(0). Record the values that you are storing so that you can verify they were correctly written during the read operation. You don't need to fill up the array, but you should write at least 10 values so that you have something significant to view during the read operation (next).

Now press the "FPGA_RESET" button to put the address back to x"00". Remember that a reset has no impact on the values within the R/W memory array. Resets only alter sequential circuits created with D-flip-flops (i.e., the FSM state memory, the address counter, and clock_div_prec.vhd). This means that at any time you can put the address counter back to x"00" by pressing reset. Press and hold KEY(1). You will see the address begin to increment and the associated data be displayed. Verify that the values displayed are what you stored. If necessary, you can slow down the clock to 1 Hz so you can see the values more clearly. You will notice that as soon as the RW_data_out is updated, the address counter increments to the next value. This is the correct functionality; however, this means you will not see the current address location corresponding to the displayed data at the same time. The address will always be one more than the location being displayed.

Set the clock back to 10 Hz, press reset, and take a short video (<5 s) showing the values on the HEX displays when holding down KEY(1). Since the only way for data to get into the R/W array is through the store operation associated with a KEY(0) press, a video showing a sequence of read operations will inherently prove data was written. The only constraint on this demonstration is that x"AA" must reside at address x"01" and x"EE" must reside at address x"02". **This video satisfies the requirements for deliverable #1.**

10.2.5.2 Take a Logic Analyzer Measurement of your R/W System During a "Store"

Now you are going to take a logic analyzer measurement of the R/W system during a store. Connect the logic channels 0 → 15 of the Analog Discovery to the GPIO_1 header as shown in [Figure 10.6](#) and [Figure 10.7](#). Make sure to connect a ground of the Analog Discovery to a ground on the GPIO_1 header. Launch Waveforms and click on the Logic tool. In the logic analyzer tool, add new busses for "address" and "RW_data_out" and assign the logic channels accordingly. Set the format for these busses to **Hexadecimal**. Add new signals for Clock_div, addr_EN, and write and assign the logic channels accordingly. Configure the following logic analyzer settings:

- Set the trigger to "Rising Edge" of addr_EN. Leave all other signals as "Don't Care".
- Buffer = 100
- Run = Repeated
- Mode = Normal (This is critical! If left at Auto, measurement will not trigger correctly)
- Source = Digital
- Position = 0.02 s
- Base = 5 ms/div

Press the "Run" button on the logic analyzer tool. The analyzer will display "Armed" as it waits for the trigger (i.e., a rising edge on addr_EN).

On the DE0-CV board, set the frequency of your counter to **1 kHz** using the SW(1) and SW(0) switches. Press the reset button to put the address back to x"00". Now set SW(9 downto 2)=x"AA" and press KEY(0). The logic analyzer will trigger. You should see the measurement in [Figure 10.11](#).

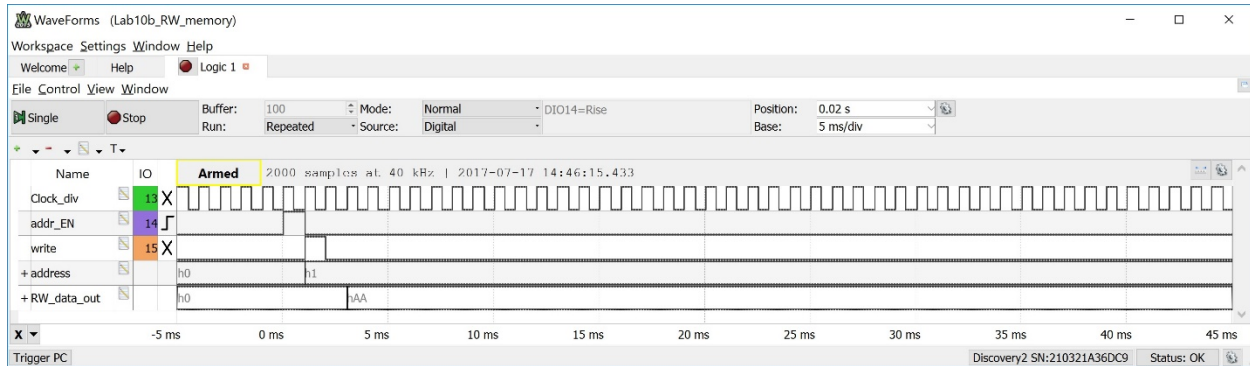


Figure 10.11
Logic Analyzer Measurement of R/W System – Storing x"AA" to Address Location x"01"

Take note of the synchronous behavior of the system. First, observe how the address does not update until one clock cycle *after* `addr_EN` is asserted. This is because the `addr_EN` is produced by the FSM, which is triggered by the edge of a clock. The counter process does not see the enable being asserted on the same clock edge because it takes a small amount of the time for `addr_EN` to reach its final asserted value. Instead, it is seen on the *next* clock edge. Even though it appears that `addr_EN` is deasserted when the next clock edge occurs, it is actually still present due to the small amount of delay that exists as it moves to its deasserted state.

Second, notice how the data x"AA" does not appear on the R/W component until *two* clock cycles after `write` is asserted. The first clock period delay is due to the same behavior described for the `addr_EN` signal. The second clock period delay is due to the output not being able to display the data on the same edge that it is stored on.

Now change SW(9 downto 2)=x"EE" and press KEY(0). The logic analyzer will trigger. You should see the measurement in [Figure 10.12](#).

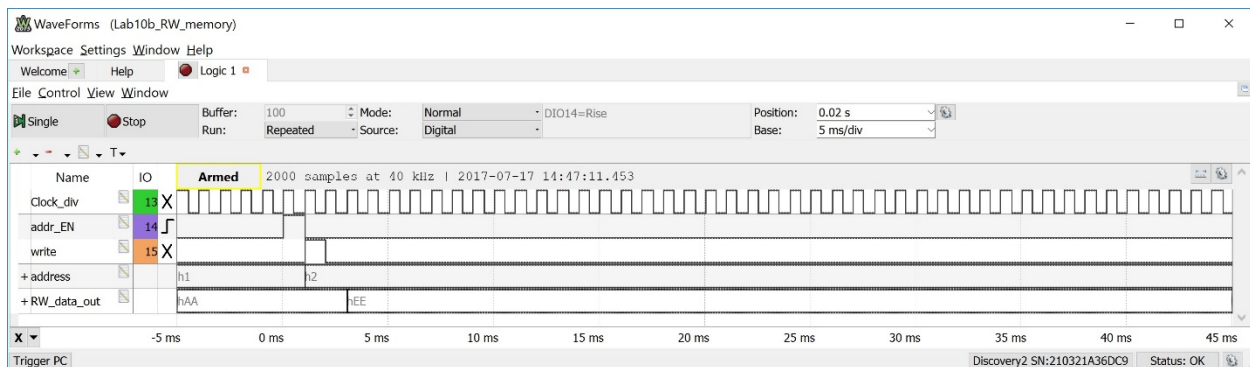


Figure 10.12
Logic Analyzer Measurement of R/W System – Storing x"EE" to Address Location x"02"

Now store x"BB" to address location x"03". Take a screenshot of this logic analyzer measurement. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

10.2.5.3 Take a Logic Analyzer Measurement of your R/W System During a "Read"

Now you are going to take a logic analyzer measurement of the R/W system during a read. Press the reset button to put the address back to x"00" and then press the KEY(1) button once. The logic analyzer will trigger. Note that the clock of the system is running at 1 kHz. This means that no matter how fast you try to remove your finger from KEY(1), it will still be slow enough that hundreds of values will be read out of the R/W array due to having `addr_EN` asserted while KEY(1) is pressed. You should see a measurement similar to [Figure 10.13](#). Your values will differ depending on what you stored in the address locations beyond x"03".

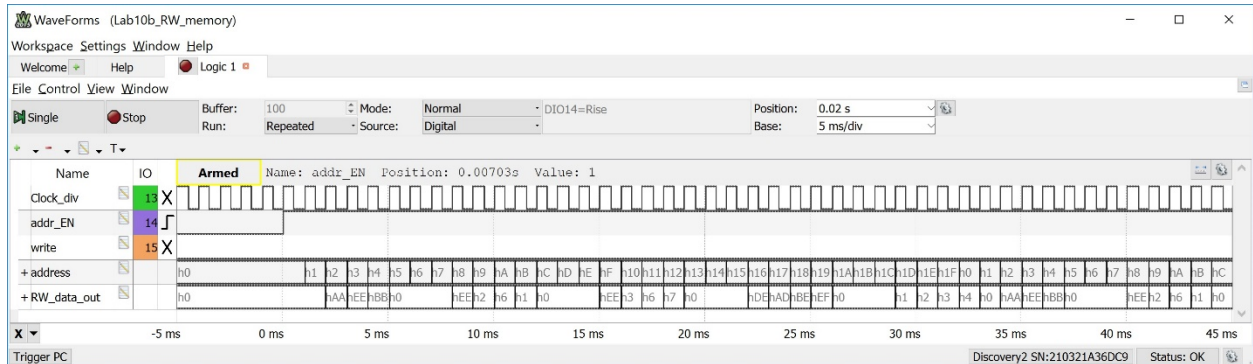


Figure 10.13
Logic Analyzer Measurement of R/W System – Reading Contents of Array

Take a screenshot of this logic analyzer measurement. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #3.**

Save your Analog Discovery workspace in your Quartus project directory so that you can recreate this measurement in the future if needed. Close Waveforms.

10.2.5.4 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #4.**

After you are done, close your Quartus project.

CONCEPT CHECK

Lab 10.2 After completing this lab exercise, can you:

- Implement a synchronous, 32x8 R/W array?
- Implement an address counter with enable?
- Implement a FSM that controls how information is written to, and read from, the R/W array?
- Observe the R/W system operation using a logic analyzer?

Chapter 11: Programmable Logic

Lab 11.1: Details of an FPGA

11.1.1 Objective

This objective of this lab is to gain experience with some of the most used reporting and viewing options associated with modern digital design tools. You will use the Quartus tool to view the resource utilization and maximum frequency reports for an FPGA design. You will also view the RTL, state machine, and chip planner views of the design.

11.1.2 Learning Outcomes

After completing this lab you should be able to:

- Determine the device utilization of a VHDL design implemented in Quartus.
- Determine the maximum clock frequency of a VHDL design implemented in Quartus.
- View the RTL interpretation of a VHDL design when implemented in Quartus.
- View the state diagram interpretation of a VHDL FSM implemented in Quartus.
- View the chip planner view of a VHDL design implemented in Quartus.

11.1.3 Parts Needed

- None.

11.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. A screenshot of the [device utilization](#) report for a VHDL design in Quartus (20% of exercise).
2. A screenshot of the [maximum clock frequency](#) report for a VHDL design in Quartus (20% of exercise).
3. A screenshot of the [RTL view](#) for a VHDL design in Quartus (20% of exercise).
4. A screenshot of the [state diagram view](#) for a VHDL FSM in Quartus (20% of exercise).
5. A screenshot of the [chip planner view](#) for a VHDL design in Quartus (20% of exercise).

11.1.5 Lab Work & Demonstration

You are going to investigate a variety of reporting and viewing options available in Quartus for a VHDL design implemented on the Cyclone V FPGA. This can be done completely in Quartus so the DE0-CV board is not needed.

11.1.5.1 *View the Device Utilization Report*

[Create a New Quartus Project by Copying Lab 10.2](#)

Open lab 10.2 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_11p1_FPGAs”. Manually copy the pin_assignments.csv file into your new project directory. While you will not be changing the pin assignments, it is good design practice to keep a current version of the pin assignments in your project directory. This project is a good representation of a typical digital system as it contains dedicated combinational logic (char_decoder.vhd), register transfer logic consisting of both D-flip-flops and combinational logic (clock_div_prec.vhd & the address counter), a finite state machine (control FSM), and a memory block (rw_32x8_sync.vhd).

[View the Logic Utilization Reports](#)

Quartus produces a variety of utilization reports. These reports become important as your design gets large enough that you begin reaching the maximum size of an FPGA as they may indicate a larger device is needed. Additionally, as your design reaches the maximum size for a device, the synthesis step can start taking an excessive amount of time to converge. This is another indication you should consider a larger device.

There are two general types of utilization reports. The first report is in the **Analysis & Synthesis** step. In this step, Quartus has synthesized your VHDL designed and performed the first mapping of logic into its available resources, however, it has not yet done the final placement and routing of your design into the FPGA. Quartus does this first step to ensure your design has a chance of fitting within the selected FPGA before going to the next, more detailed implementation step.

In the task pane of Quartus, expand the *Analysis & Synthesis* step and double click on *View Report*. You will see a “Compilation Report – top” window appear in the main window. On the left side of report window, there is a table of contents. This table of contents contains all of the reports for the implementation. It can sometimes be difficult to find what you are looking for in the table of contents. By double clicking on the *View Report* step in the task pane, it will automatically bring you to the corresponding location in the table of contents. By default, double clicking on *Analysis & Synthesis* → *View Report* in the task pane will take you to the *Analysis & Synthesis Summary* report. In the table of contents, four items down from the summary report is the *Resource Usage Summary*. Click on this report. You should see the report in [Figure 11.1](#).

The screenshot shows the 'Compilation Report - top' window. On the left is a 'Table of Contents' tree with 'Analysis & Synthesis' expanded to 'Resource Usage Summary'. The main window displays the 'Analysis & Synthesis Resource Usage Summary' table.

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	57
2		
3	Combinational ALUT usage for logic	99
1	-- 7 input functions	0
2	-- 6 input functions	13
3	-- 5 input functions	3
4	-- 4 input functions	22
5	-- <=3 input functions	61
4		
5	Dedicated logic registers	63
6		
7	I/O pins	82
8	Total MLAB memory bits	0
9	Total block memory bits	256
10		
11	Total DSP Blocks	0
12		
13	Maximum fan-out node	Reset~input
14	Maximum fan-out	63
15	Total fan-out	773
16	Average fan-out	2.31

Figure 11.1
Analysis & Synthesis - Resource Usage Summary

This report gives a variety of useful information including the number of LUTs and registers needed, the number of I/O pins, and the number of memory bits required. This report does not provide information about what percentage of resources of your selected FPGA are being consumed. To see the percentage, you need to look at the usage report after it has been placed and routed.

In the Task pane of Quartus, expand the **Fitter (Place & Route)** step and double click on *View Report*. You will see the *Fitter Summary Report* in [Figure 11.2](#).

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Tue ... 18 07:31:22 2017
Quartus Prime Version	17.0.0 Build 595 0...17 SJ Lite Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone V
Device	5CEBA4F23C7
Timing Models	Final
Logic utilization (in ALMs)	58 / 18,480 (< 1 %)
Total registers	74
Total pins	82 / 224 (37 %)
Total virtual pins	0
Total block memory bits	256 / 3,153,920 (< 1 %)
Total RAM Blocks	1 / 308 (< 1 %)
Total DSP Blocks	0 / 66 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 11.2
Fitter Summary

This summary shows the number of resources being used by the design on the Cyclone V 5CEBA4F23C7 FPGA. Notice for this design, the *Logic Utilization*, *Total block memory bits*, and *Total RAM Blocks* usage are all under 1%. This gives an indication of how vast the resources available are on a modern FPGA.

A more detailed breakdown of the resource utilization can be found in *Fitter* → *Resource Section* → *Resource Usage Summary*. Open this report and get a feel for the level of detail that is provided by Quartus for the resource usage. While, this level of detail is interesting, usually the fitter summary information will suffice for determining how close a design is to exceeding a device's maximum capacity.

Go back to the *Fitter Summary* and take a screenshot of the report (the window similar to [Figure 11.2](#)). Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #1.**

11.1.5.2 View the Maximum Frequency Report

In synchronous systems, one of the important performance specifications is the clock speed. Quartus provides an estimate for the maximum clock frequency based on all of the delays in your design and the setup/hold specifications of its D-flip-flops. In the task pane, expand the **TimeQuest Timing Analyzer** and double click on the *View Report* step. In the main compilation report, expand the *TimeQuest Timing Analyzer* section to see the available reports.

You will see four models for estimating delay. These four models contain delay for the devices on the FPGA that consider the process (slow vs. fast) and temperature (85C vs. 0C). The worst case delay for a digital system is always when the manufacturing process yields the slowest transistors and they are run at the maximum temperature. The maximum frequency is reported for this case under the *Slow 1100mV 85C Model*. Expand this model and click on the *Fmax Summary*. You'll see the estimated maximum frequency for any clocks in your system. Your report will look like [Figure 11.3](#).

Compilation Report - top

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
 - Summary
 - Parallel Compilation
 - Clocks
 - Slow 1100mV 85C Model
 - Fmax Summary
 - Timing Closure Recommendations
 - Setup Summary
 - Hold Summary
 - Recovery Summary
 - Removal Summary
 - Minimum Pulse Width Summary
 - Metastability Summary
 - Slow 1100mV OC Model

	Fmax	Restricted Fmax	Clock Name	Note
1	167.48 MHz	167.48 MHz	Clock_50	
2	210.75 MHz	210.75 MHz	clock_div_prec:Clk_div	

This panel reports FMAX for every clock in the design, regardless of the user-specified clock periods. FMAX is only computed for paths where the source and

Figure 11.3
Fmax Summary for Slow 1100mV 85C Model

Notice that this report gives Fmax for all clocks in the system. In this design, there are two clocks, Clock_50 and CLK_div (CLK_div is the signal name within clock_div_prec.vhd). Notice that CLK_div can run faster than Clock_50. This is likely due to the additional path that the Click_50 signal must traverse to enter the FPGA and be routed to the clock_div_prec.vhd logic.

Take a screenshot of the Fmax summary. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #2.**

11.1.5.3 Run the RTL Viewer

The RTL view of your design provides an initial mapping of your post-synthesis logic into resources on the FPGA. This can be useful to see how Quartus is interpreting your design. In the task pane, expand the **Analysis & Synthesis** step. Then expand the *Netlist Viewers* step and double click on “RTL Viewer”. The RTL viewer will load and then appear. You should see the screen in [Figure 11.4](#).

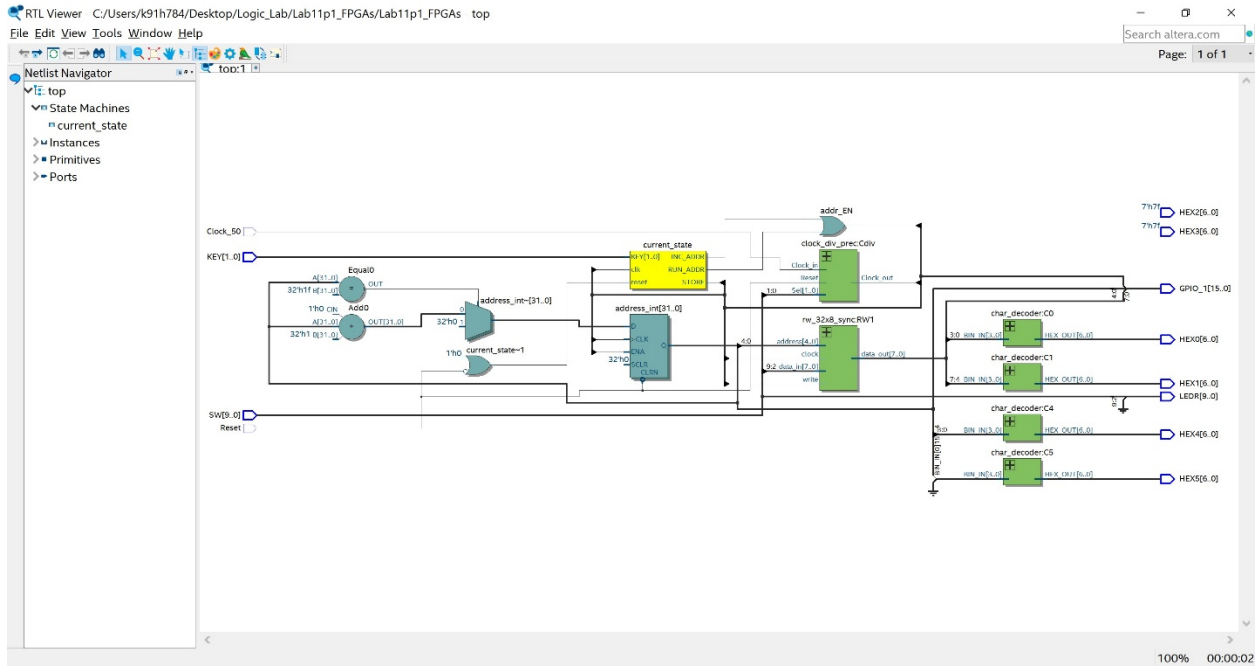


Figure 11.4
RTL View of the Top Level of R/W Memory System

In this view you are able to see the major groupings of logic. Any block with a + can be expanded into in order to see more details of the design. Expand the char_decoder to see how the combinational logic is mapped by clicking on the “+” on the “char_decoder:C0” block. You will see the RTL view in [Figure 11.5](#).

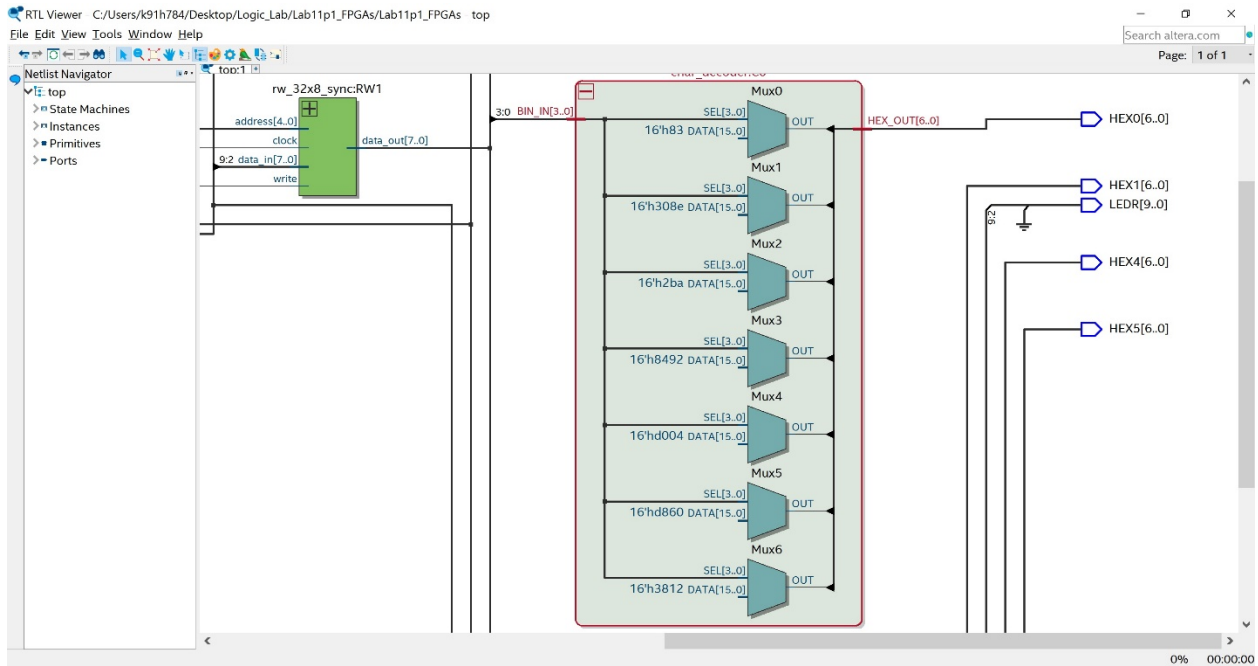


Figure 11.5
RTL View of char_decoder:C0

Notice that each bit of the 7x output bits of the char_decoder is driven by a multiplexer. This is showing the lookup table (LUT) approach used to implement combinational logic on FPGAs. Click the ← icon in the upper left corner of the RTL viewer to go back to the top level.

Now expand the rw_32x8_sync:RW1 block. You will see the RTL view in [Figure 11.6](#).

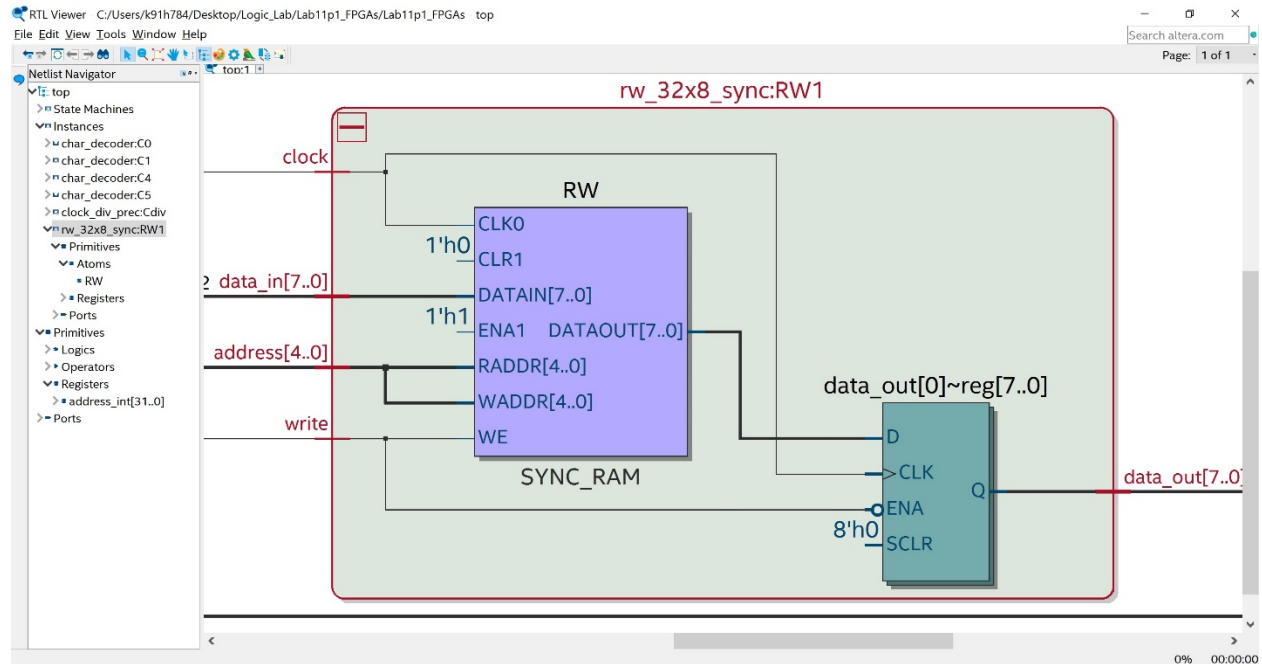


Figure 11.6
RTL View of rw_32x8_sync:RW1

The RTL view of the R/W system shows two items of interest. First, the block labeled “RW” is a built in *block RAM* (BRAM) within the FPGA. Notice that this block actually has more functionality than our design uses. This includes a clear (CLR1) and an enable (ENA1). When a VHDL model is created that only uses a subset of the features of a built-in block within the FPGA, Quartus will automatically wire the unused inputs to logic values so that they don't impact the desired functionality. In this case, Quartus wired CLR1=0 and ENA1=1 so they have no effect on our design.

The second item to notice is that to get the rising edge sensitivity for the R/W memory that we designed into our VHDL model, Quartus needed to attach 8x D-flip-flops to the output of the BRAM. Notice how the FPGA D-flip-flops also have additional functionality beyond what we typically include in our D-flip-flop model. This includes an enable (ENA) and a synchronous clear (SCLR). The synchronous clear is different from a reset in that it is only acknowledged on the rising edge of the clock. The reset on the FPGA is considered a *global* net, so it is not shown as it is implicitly connected to all D-flip-flops on the device. Click the ← icon in the upper left corner of the RTL viewer to go back to the top level.

Take a screenshot of the top level RTL view (similar to [Figure 11.4](#)). Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #3.** Close the RTL viewer.

11.1.5.4 Run the State Machine Viewer

Quartus will recognize any finite state machine that are modeled in the design and provide a state diagram view of their behavior. Back in the task pane of Quartus, expand the **Analysis & Synthesis** step. Then expand the *Netlist Viewers* step and double click on “State Machine Viewer”. The state machine viewer will load and then appear. You should see the screen in [Figure 11.7](#).

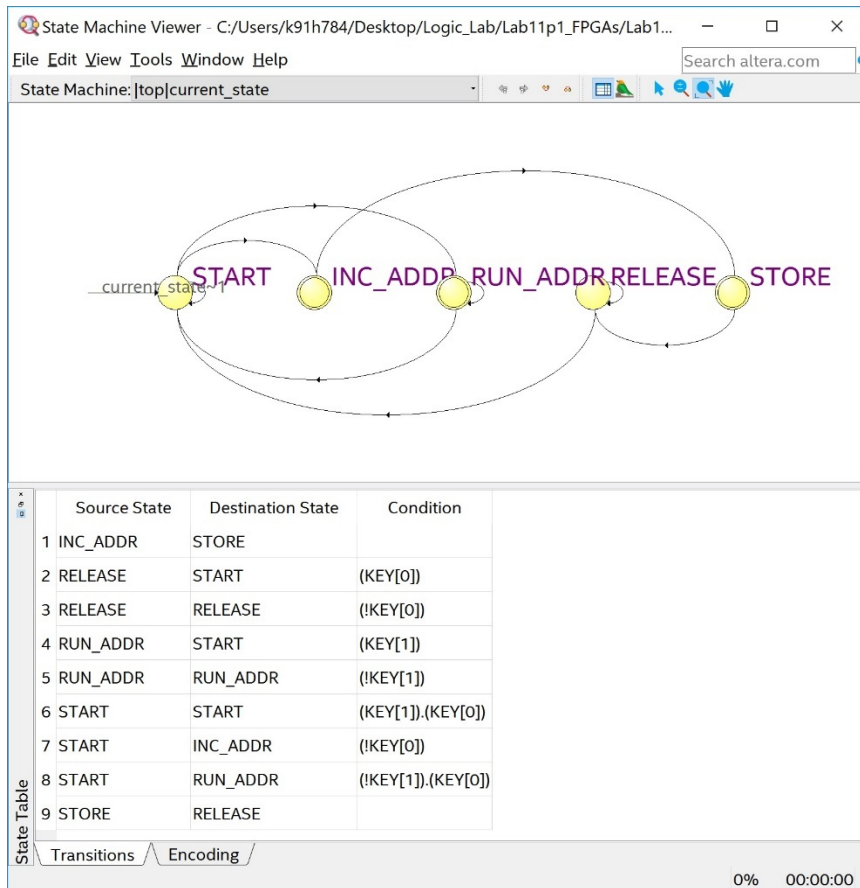


Figure 11.7
State Machine View of Control FSM

This view shows the state diagram that Quartus produced based on your FSM VHDL model. This is a useful tool as it allows you to see how Quartus interpreted the behavior you intended to model in VHDL. If the state diagram does not match your desired behavior, you have an issue in your VHDL that must be fixed. You can also view the encoding scheme that the synthesizer chose for your FSM states by clicking on the “Encoding” tab.

Take a screenshot of the state machine view. Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #4.** Close the state machine viewer.

11.1.5.5 Run the Chip Planner Tool

Another useful view of the implementation of a VHDL design is the *chip planner*. This gives an approximation of the location on the device that your logic is mapped into. Back in the task pane of Quartus, expand the **Fitter (Place & Route)** step. Double click on “Chip Planner”. The chip planner tool will load and then appear. You should see the window in [Figure 11.8](#).

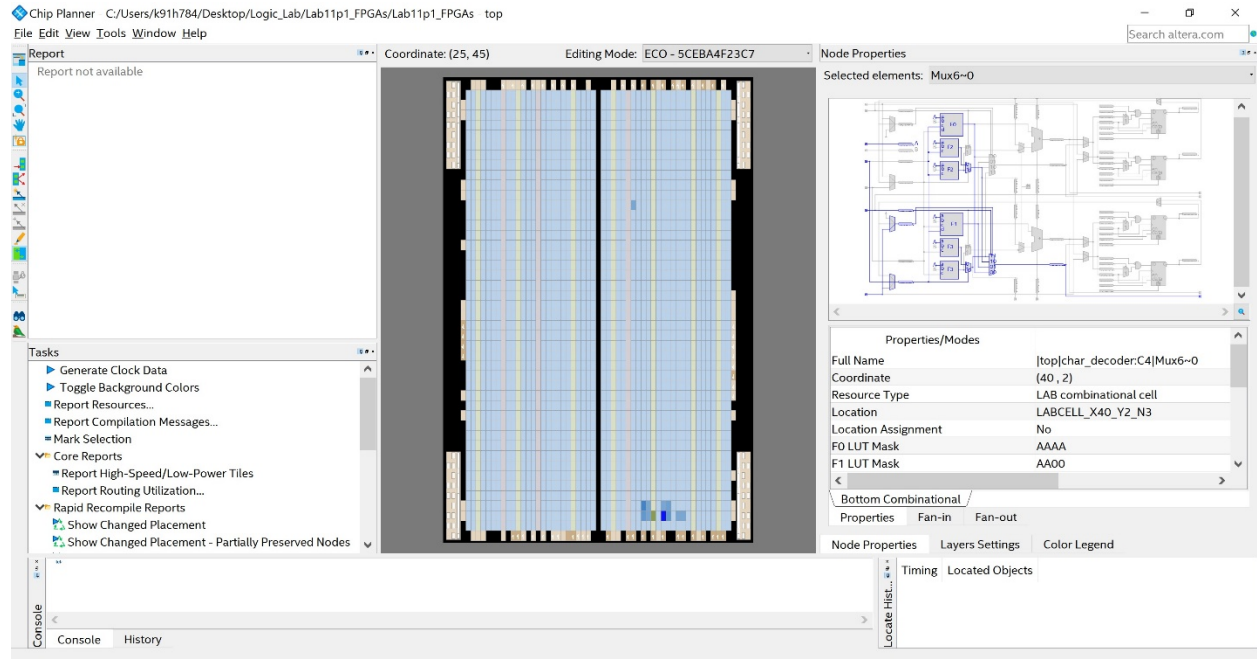


Figure 11.8
Chip Planner View of Top Level of R/W Memory System

The chip planner shows all of the available logic resources and memory blocks. The logic resources are grouped into what Quartus called *logic array blocks (LABs)*. These are the light blue rectangles in the chip planner view. There are also columns of BRAMs (light red) and dedicated hardware blocks for digital signal processing (light yellow). Each of these blocks can be selected to view the available resources.

The chip planner will highlight any block that is used by the design. In [Figure 11.8](#) you'll notice there are a handful of highlighted blocks in the lower right corner of the chip plan. Zoom in on this region using either the zoom icon or by using the mouse scroll wheel while holding down the CNTL key. As you zoom in, you'll start to see that each block contains more resources that are also highlighted if used. Zoom in until you see a block that is used and click on it. The details of the internal resources will show up in the upper right pane of the chip planner window. You should see a window similar to [Figure 11.9](#).

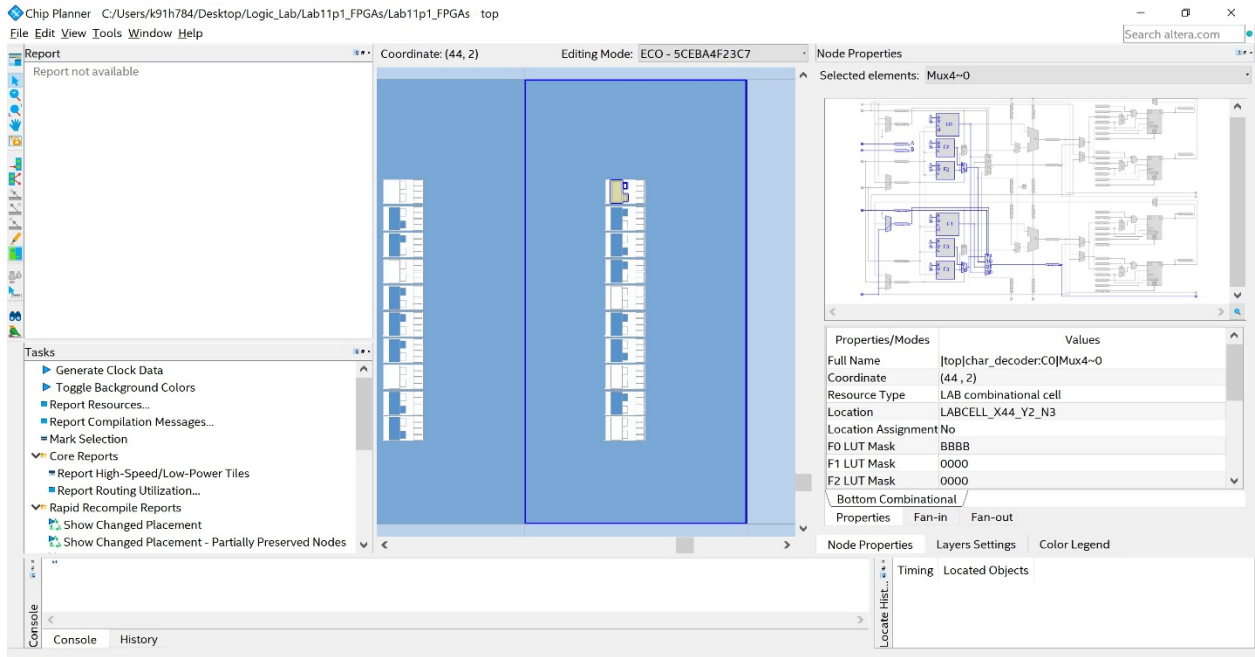


Figure 11.9
Chip Planner View of Top Level of R/W Memory System (Zoomed)

Double click on one of the used LABs. A new window will appear that shows the resources within a LAB. You should see the window in Figure 11.10.

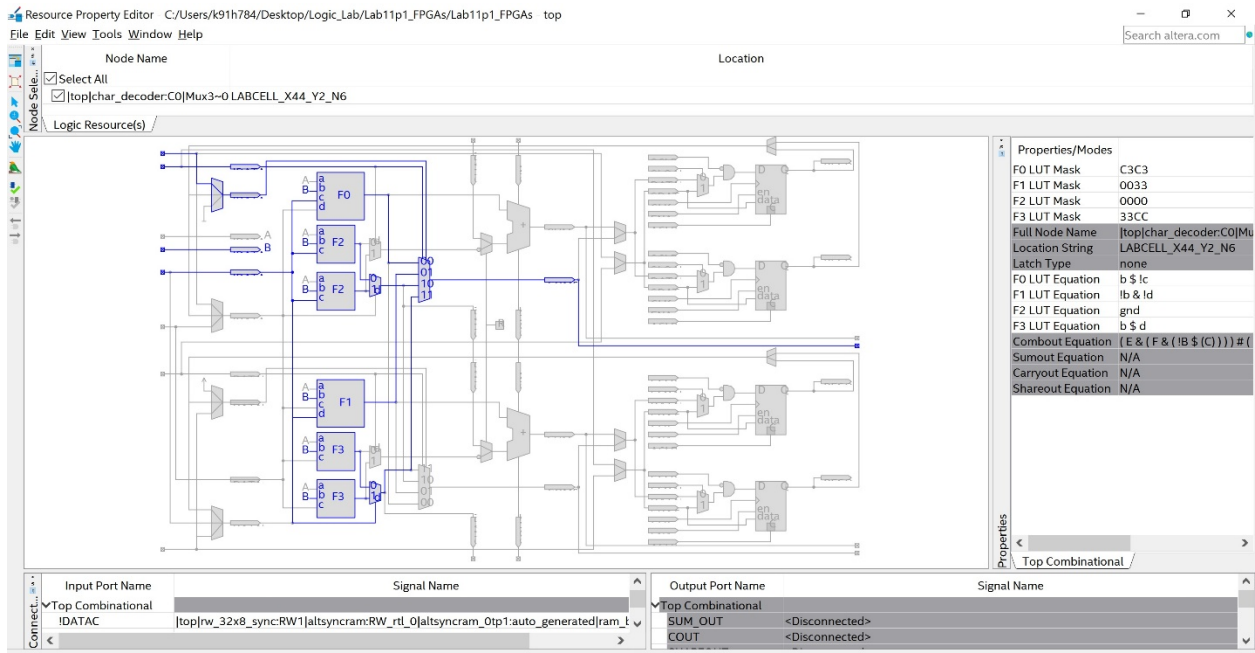


Figure 11.10
Chip Planner Details of Logic Array Block

Within the LAB, you'll notice that the resources used are again highlighted. Take notice of all of the resources available. Each LAB contains multiple LUTs (labeled F0, F1, etc.), multiple D-flip-flops, and numerous multiplexers to

handle how the resources are configured. Understanding the FPGA resources at this level can sometimes be exploited to create more optimized designs.

Take a screenshot of the chip planner view of the LAB (your window similar to [Figure 11.10](#)). Save the image in JPG format with a descriptive file name. **This image satisfies the requirements for deliverable #5.** Close all of the chip planner windows. Save your project and close Quartus.

After you are done, close your Quartus project.

CONCEPT CHECK

Lab 11.1 After completing this lab exercise, can you:

- Determine the device utilization of a VHDL design implemented in Quartus?
- Determine the maximum clock frequency of a VHDL design implemented in Quartus?
- View the RTL interpretation of a VHDL design when implemented in Quartus?
- View the state diagram interpretation of a VHDL FSM implemented in Quartus?
- View the chip planner view of a VHDL design implemented in Quartus?

Chapter 12: Arithmetic Circuits

Lab 12.1: Unsigned Adders

12.1.1 Objective

This objective of this lab is to give experience implementing adders that are modeled in VHDL. You will create a 4-bit **unsigned** adder. The inputs for your adder will come from the slider switches on the DE0-CV FPGA board. The inputs, sum, and carry out will be displayed on the HEX character displays. The inputs and carry will also be displayed on the red LEDs.

12.1.2 Learning Outcomes

After completing this lab you should be able to:

- Design a 4-bit, unsigned adder using the “+” operator from the `numeric_std` package.
- Type cast between types `unsigned` and `std_logic_vector`.

12.1.3 Parts Needed

- DE0-CV FPGA board.

12.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstration of an unsigned adder that displays its inputs, sum, and carry on the displays of the DE0-CV board (90% of exercise).
2. Provide your `top.vhd` design file (10% of exercise).

12.1.5 Lab Work & Demonstration

You are going to design a 4-bit, unsigned adder in VHDL. The two 4-bit inputs will come from SW(7:4) and SW(3:0). The two inputs will be displayed on the HEX2 and HEX0 displays through your `char_decoder.vhd` and also on the red LEDs. The 4-bit sum will be displayed on HEX4. The carry will be displayed on HEX5 and also on LEDR(9). The block diagram for the 4-bit unsigned adder system is shown in [Figure 12.1](#).

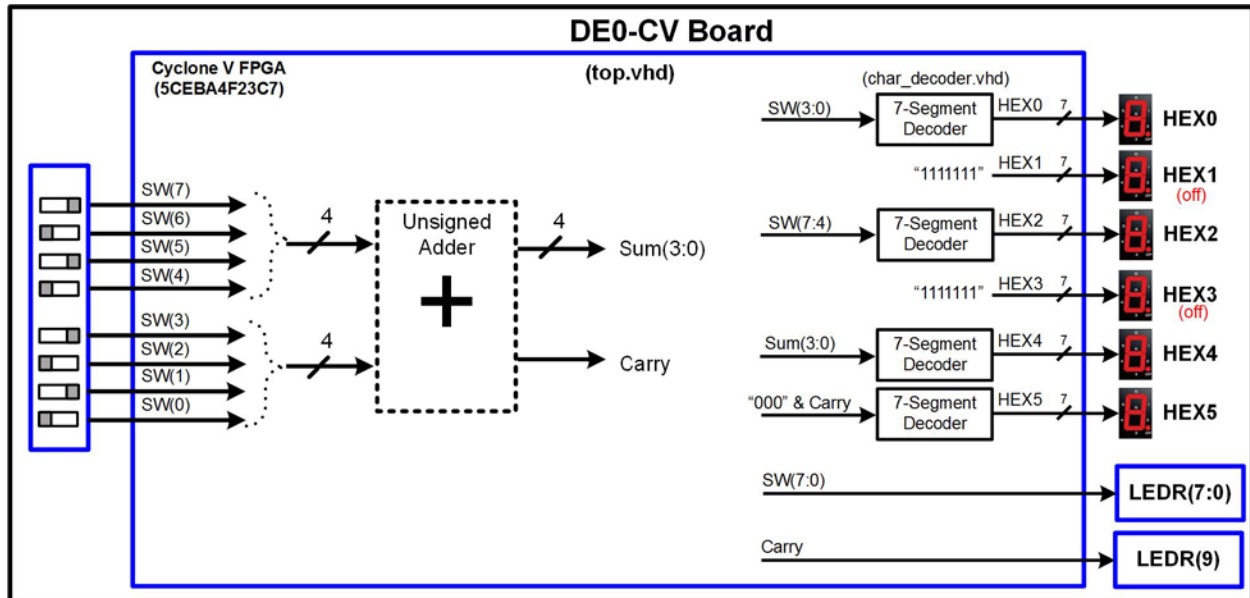


Figure 12.1
Block Diagram for the 4-Bit Unsigned Adder System

Figure 12.2 shows a picture of the I/O on the DE0-CV board used in this exercise.

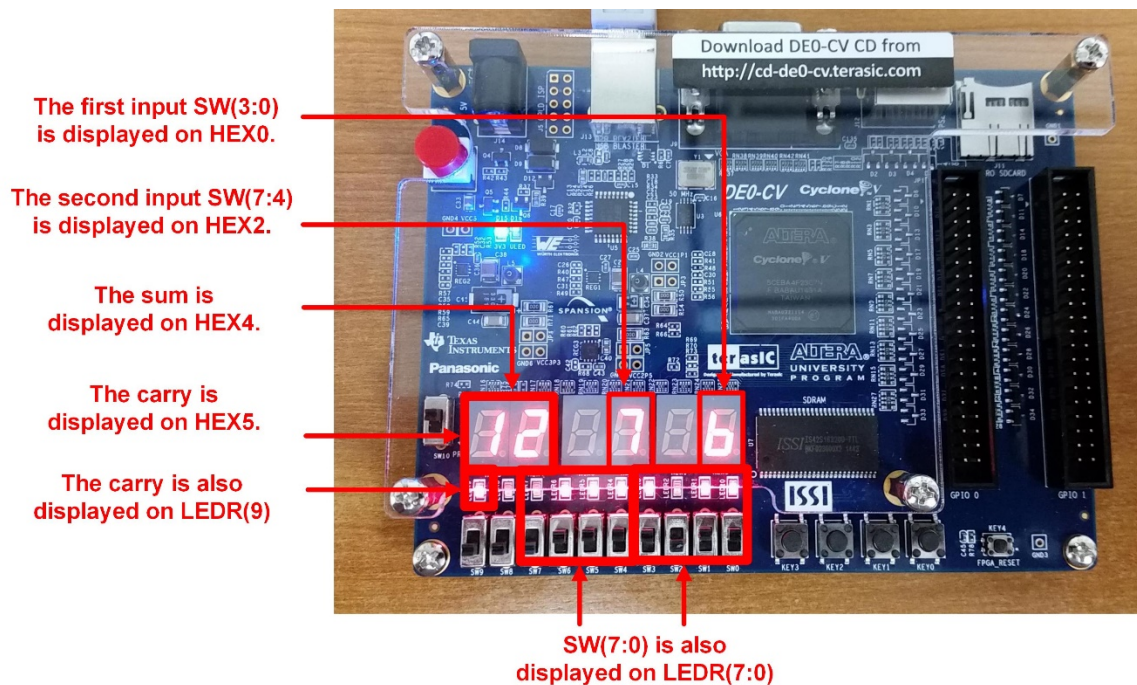


Figure 12.2
Picture of the Unsigned Adder System on the DE0-CV Board

12.1.5.1 Implement the Adder System

[Create a New Quartus Project by Copying Lab 11.1](#)

Open lab 11.1 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_12p1_Unsigned_Binary_Adder”. Manually copy the pin_assignments.csv file into your new project directory. While you will not need to update any pins in this exercise, it is good practice to keep the most up to date assignment file in your project directory. You can delete all of the VHDL from 11.1 that is not used in this exercise.

[Modify the Entity for this Exercise](#)

Alter the entity to match the ports that will be used in this exercise. This will involve eliminating many of the ports from lab 11.1. When done, the package and entity portion of your design should look like [Figure 12.3](#). Note that you will not be using LEDR(8), but it is easier to declare the entire 10-bits of LEDR in one vector. You can simply drive LEDR(8) with ‘0’ to turn it off in this exercise.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity top is
6   port (SW      : in  std_logic_vector(7 downto 0);
7         LEDR    : out std_logic_vector(9 downto 0);
8         HEX0    : out std_logic_vector(6 downto 0);
9         HEX1    : out std_logic_vector(6 downto 0);
10        HEX2    : out std_logic_vector(6 downto 0);
11        HEX3    : out std_logic_vector(6 downto 0);
12        HEX4    : out std_logic_vector(6 downto 0);
13        HEX5    : out std_logic_vector(6 downto 0));
14 end entity;
15

```

Figure 12.3
Entity for the Unsigned Adder System

[Create the Unsigned Adder Logic](#)

In your top.vhd, design the logic for the unsigned adder. You can create the adder logic using either a process or a sequential signal assignment.

Hint 1: Use the “+” operator from the numeric_std package to perform the addition. Be aware the “+” operation is only supported for types *unsigned* and *signed* in this package. This means you’ll need to create internal vectors of type *unsigned* to implement the inputs and output of your adder. You can first type cast your inputs SW(7:4) and SW(3:0) into your new unsigned vectors. You can then perform the addition with a resulting sum and carry that are of type unsigned. You’ll then need to type cast back the sum from unsigned to std_logic_vector to drive the HEX displays.

Hint 2: The carry bit can be created as simply the 5th bit in an addition; however, the size of the inputs and output need to be the same width for the “+” operator. If you concatenate the 4-bit inputs with a ‘0’, they will become 5-bits wide and their values won’t change (this only works for unsigned numbers). If you declare your internal, unsigned signal for the sum to also be 5-bits, the addition can be performed directly with the “+” operator. The 5-bit sum can then be used to drive the HEX displays. You will drive the lower 4-bits (i.e., the 4-bit sum) to HEX4 and the 5th bit (i.e., the carry) to HEX5 through your char_decoder.vhd components. The carry will need to be concatenated with “000” to make it match the input requirements of your char_decoder.vhd.

Compile your Design

Compile your design and fix any errors that you encounter. Note that you don't need to alter any pin assignments since you are using a subset of the entity from lab 11.1. Although it is always a good idea to verify the assignments in pin planner.

Download and Test Your Design

Open the programmer tool and download your design to the FPGA. You should now see the inputs on HEX0 and HEX2 in addition to the red LEDs. You should see the sum and carry on HEX4 and HEX5. The carry should also be displayed on LEDR(9). Cycle through the majority of input codes on the slider switches and ensure that the adder is producing the correct result.

Take a short video (<5 s) showing the proper operation of your design. You should show a 2-3 additions with at least one asserting the carry bit. **This video satisfies the requirements for deliverable #1.**

12.1.5.2 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #2.**

After you are done, close your Quartus project.

CONCEPT CHECK

Lab 12.1 After completing this lab exercise, can you:

- Design a 4-bit, unsigned adder using the "+" operator from the numeric_std package?
- Type cast between types *unsigned* and *std_logic_vector*?

Lab 12.2: Signed Adders

12.2.1 Objective

This objective of this lab is to give more experience implementing adders that are modeled in VHDL. You will create a 4-bit **signed** adder. The inputs for your adder will come from the slider switches on the DE0-CV FPGA board. The inputs and sum will be displayed on the HEX character displays. This will require creating a new decoder that will drive two HEX displays for each 4-bit input, one for the sign of the number and one for the magnitude (`twos_comp_decoder.vhd`). The inputs will also be displayed on the red LEDs. You will also display whether *two's complement overflow* occurred by asserting LEDR(9).

12.2.2 Learning Outcomes

After completing this lab you should be able to:

- Design a 4-bit, signed adder using the “+” operator from the `numeric_std` package.
- Type cast between types *signed* and *std_logic_vector*.
- Create a decoder to display the decimal value of a 4-bit signed number on two character displays.
- Create the logic to determine if two's complement overflow occurred.

12.2.3 Parts Needed

- DE0-CV FPGA board.

12.2.4 Deliverables

The deliverable(s) for this lab are as follows:

1. Demonstration of a signed adder that displays its inputs and sum on the displays of the DE0-CV board and also indicates whether two's complement overflow occurred by asserting LEDR(9) (90% of exercise).
2. Provide your `top.vhd` design file (10% of exercise).

12.2.5 Lab Work & Demonstration

You are going to design a 4-bit, signed adder in VHDL. The two 4-bit inputs will come from SW(7:4) and SW(3:0) and will be displayed on the red LEDs. The inputs and sum will be displayed on the HEX displays on the DE0-CV board. Since these 4-bit values are two's complement codes, they contain negative numbers. In order to display a negative number using decimal symbols, two HEX displays are required, one for the sign and one for the magnitude. You will need to create a new decoder (`twos_comp_decoder.vhd`) that will take in a 4-bit two's complement code and drive two HEX displays with the corresponding decimal symbols. You are also going to create the logic to determine whether two's complement overflow occurred. If it has, you will assert LEDR(9). The block diagram for the 4-bit signed adder system is shown in [Figure 12.4](#).

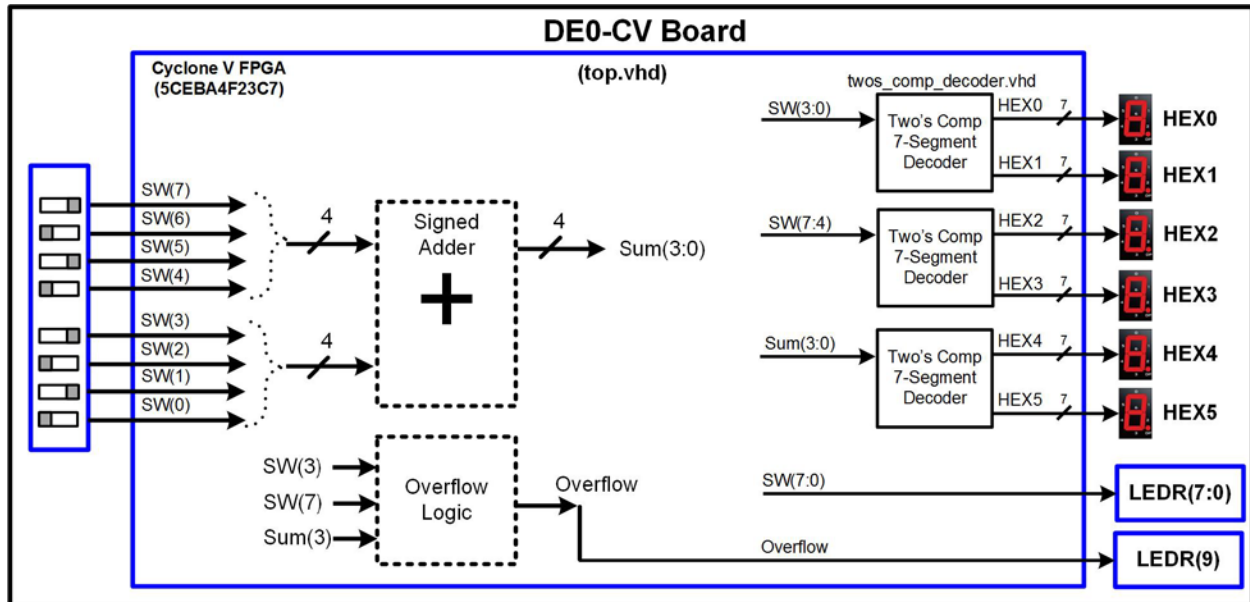


Figure 12.4
Block Diagram for the 4-Bit Signed Adder System

Figure 12.5 shows a picture of the I/O on the DE0-CV board used in this exercise.

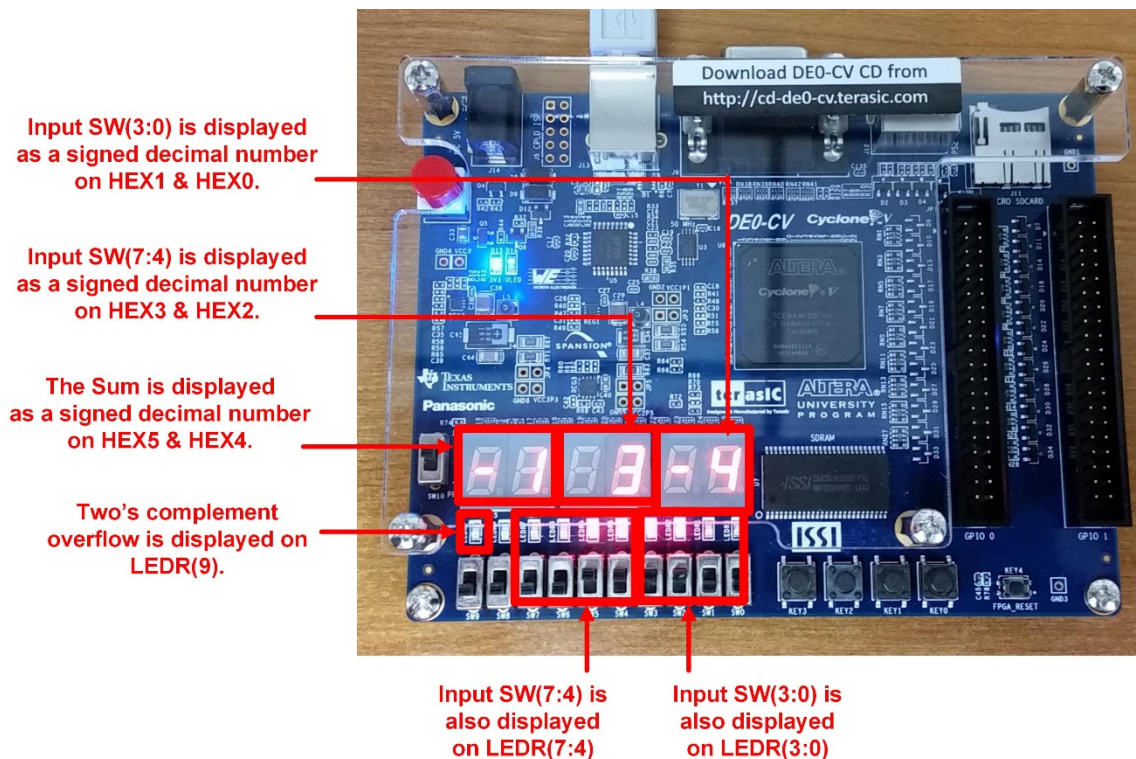


Figure 12.5
Picture of the Signed Adder System on the DE0-CV Board

12.2.5.1 Implement the Adder System

Create a New Quartus Project by Copying Lab 12.1

Open lab 12.1 in Quartus. Use the Copy Project feature to create the project for this lab. Name the folder and project “Lab_12p2_Signed_Binary_Adder”. Manually copy the pin_assignments.csv file into your new project directory. While you will not need to update any pins in this exercise, it is good practice to keep the most up to date assignment file in your project directory.

Create the Signed Adder Logic

In your top.vhd, design the logic for the signed adder. You can create the adder logic using either a process or a sequential signal assignment.

Hint: Use the “+” operator from the numeric_std package to perform the addition. Be aware the “+” operation is only supported for types *unsigned* and *signed* in this package. This means you’ll need to create internal vectors of type *signed* to implement the inputs and output of your adder. You can first type cast your inputs SW(7:4) and SW(3:0) into your new signed vectors. You can then perform the addition with a resulting sum that are of type *signed*. You’ll then need to type cast back the sum from *signed* to *std_logic_vector* to drive the HEX displays. Note that when performing an addition using two’s complement number, the carry is ignored. This means the inputs and sum can both be 4-bits and the carry is non-existent.

Create the Two’s Complement Decoder Sub-System

You are going to create a model for a two’s complement character decoder. This will require creating a new VHDL file in Quartus. Your file should be called “twos_comp_decoder.vhd”. The system will have a 4-bit input called *TWOS_COMP_IN*, which will be a two’s complement number. The system will have two, 7-bit outputs called *MAG_OUT* and *SIGN_OUT*, which will drive two HEX displays on the DE0-CV board. The port and entity definition are given in Figure 12.6.

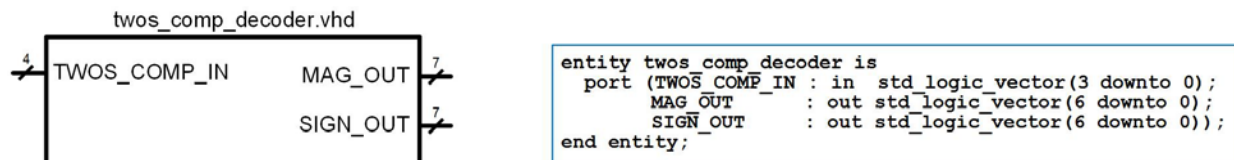


Figure 12.6
Port and Entity Definition for Two's Complement Character Decoder

Recall that the operation for adding signed numbers is identical to adding unsigned numbers. The only exception is that the carry bit is ignored. The main difference when using two’s complement numbers is how we interpret and display them. When the number is displayed on a character display, it takes two displays, one for the sign and one for the magnitude. For the 4-bit code in this exercise, the decimal symbols to be displayed are {-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7}. This type of decoder can be created in a similar manner to your unsigned decoder *char_decoder.vhd*. The difference is that the output will drive two, 7-bit ports and the values sent to displays will need to reflect the decimal symbols corresponding to the 4-bit input. Create the logic for this decoder. Save your file.

Back in your top.vhd, declare the *twos_comp_decoder.vhd* component. Instantiate it three times. The three inputs to the decoders will be SW(3:0), SW(7:0), and your internal sum. The outputs will be the six HEX displays. The details of the signal connections are given in Figure 12.4.

Create the Logic to Determine if Two’s Complement Overflow has Occurred

In your top.vhd, you need to create logic to determine if the sum resulted in two’s complement overflow. Two’s complement overflow refers to when the result of an operation falls outside of the range of numbers that can be represented using the number of bits in the output. For a 4-bit code, this means any result that is <(-8) or >(7). Recall that the conditions to detect overflow when adding two’s complement overflow are either:

- The sum of two positive numbers yields a negative number.
- The sum of two negative numbers yields a positive number.

These two conditions can be monitored by observing the MSB of the inputs and output of the sum since the MSB is always the sign bit for a two's complement code. If overflow occurs, you will assert LEDR(9). This logic can be implemented using a process and an if/then construct.

Compile your Design

Compile your design and fix any errors that you encounter. Note that you don't need to alter any pin assignments since you are using a subset of the entity from lab 12.1. Although it is always a good idea to verify the assignments in pin planner.

Download and Test Your Design

Open the programmer tool and download your design to the FPGA. You should now see the inputs on HEX3, HEX2, HEX1, and HEX0. You should see the sum on HEX5 and HEX4. The inputs and output should display the decimal values including the negative sign when appropriate. The inputs should also be displayed on the red LEDs and overflow should be indicated on LEDR(9).

Cycle through the majority of input codes on the slider switches and ensure that the adder is producing the correct result. When testing the overflow condition, you will want to provide two inputs that result in a sum that is outside of the range possible for a 4-bit two's complement code. Examples of these are:

- $7+7=14$. Overflow has occurred because 14 can't be represented with a 4-bit two's complement code.
- $(-8)+(-8)=(-16)$. Overflow has occurred because 14 can't be represented with a 4-bit two's complement code.

Take a short video (<5 s) showing the proper operation of your design. You should show conditions that have both positive and negative numbers on the inputs and output and conditions that show no overflow and with overflow. **This video satisfies the requirements for deliverable #1.**

12.2.5.2 Save a Copy of your top.vhd for your Records

Locate the top.vhd file for this exercise. **This file satisfies the requirements for deliverable #2.**

After you are done, close your Quartus project.

CONCEPT CHECK

Lab 12.2 After completing this lab exercise, can you:

- Design a 4-bit, signed adder using the "+" operator from the numeric_std package?
- Type cast between types *signed* and *std_logic_vector*?
- Create a decoder to display the decimal value for a 4-bit signed number on two character displays?
- Create the logic to determine if two's complement overflow occurred?

Chapter 13: Computer System Design

Lab 13.1: 8-Bit Computer Implementation

13.1.1 Objective

The objective of this lab is to gain experience implementing computer systems. You will design and implement a fully functional, 8-bit, computer system. The computer will be designed incrementally to verify that smaller portions of the computer are verified before moving onto the next step in the design. Ultimately, the computer will be implemented on the DE0-CV board.

13.1.2 Learning Outcomes

After completing this lab you should be able to:

- Design a VHDL model for an 8-bit computer system.
- Perform functional simulations to verify the proper execution of instructions by observing simulation waveforms.
- Implement the computer on an FPGA and verify the proper execution of instructions by observing signals on the various I/O of the DE0-CV board.

13.1.3 Parts Needed

- DE0-CV FPGA Board.

13.1.4 Deliverables

The deliverable(s) for this lab are as follows:

1. "VHDL Shell" - Provide the 8x VHDL design files that represent the structural *shell* for the microcomputer and a screenshot of your simulation transcript verifying that there are no broken connections (10% of exercise).
2. "Functional Simulation of the Four Basic Instructions" - Provide simulation waveforms that verify the proper execution of the LDA_IMM, LDA_DIR, STA_DIR, and BRA instructions (40% of exercise).
3. "Implementation of Basic Instructions on the FPGA" – Demonstration of the proper operation of the LDA_DIR, STA_DIR, and BRA instruction on the DE0-CV Board (20% of exercise).
4. "Additional Instruction Implementation" – Provide simulation waveforms and demonstration of implementation on the DE0-CV board of additional instructions you wish to include in your computer (30% of exercise).

13.1.5 Lab Work & Demonstration

13.1.5.1 VHDL Shell

In this part, you are going to create the *VHDL shell* for the computer system. This entails creating all of the VHDL files for the system, creating all of the entities, and connecting each sub-system within the hierarchy. You will then run a ModelSim simulation to load the design and verify that all connections are correct. If any of the connections are incorrect (i.e., port map naming mismatch), the simulation will fail when loading. A test bench is provided for this simulation (computer_TB.vhd). In this part of the exercise, you won't be viewing any simulation waveforms, you'll just be verifying that the simulation loaded correctly by observing messages in the transcript window. The structure for the computer.vhd, cpu.vhd, and memory.vhd are given in [Figure 13.1](#), [Figure 13.2](#), and [Figure 13.3](#) respectively.

Example: Top Level Block Diagram for the 8-Bit Computer System

The following is the top level block diagram for our 8-bit computer system example.

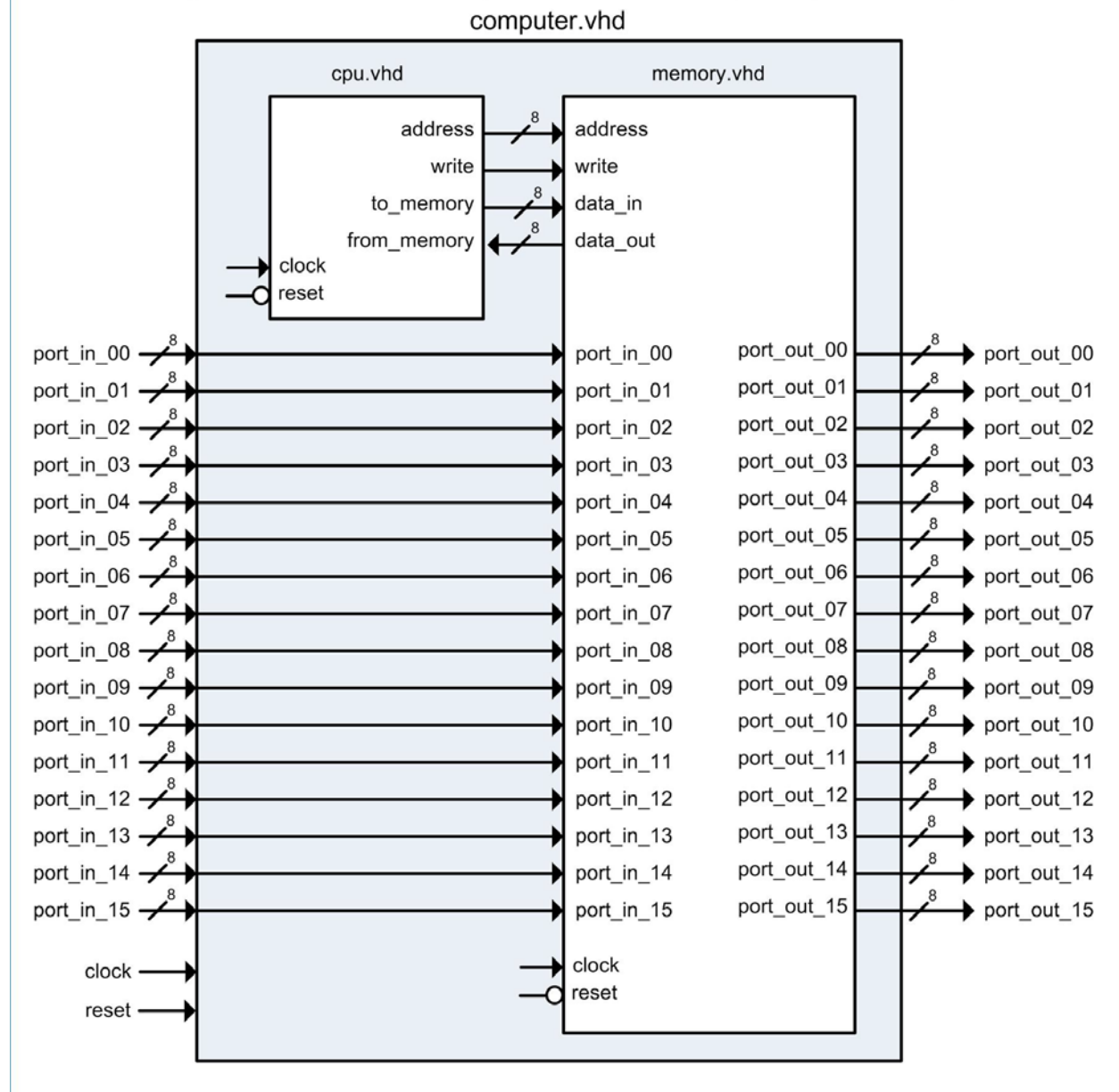


Figure 13.1
Structure of `computer.vhd`

Example: CPU Block Diagram for the 8-Bit Computer System

The following is the block diagram for the CPU of our 8-bit computer system example.

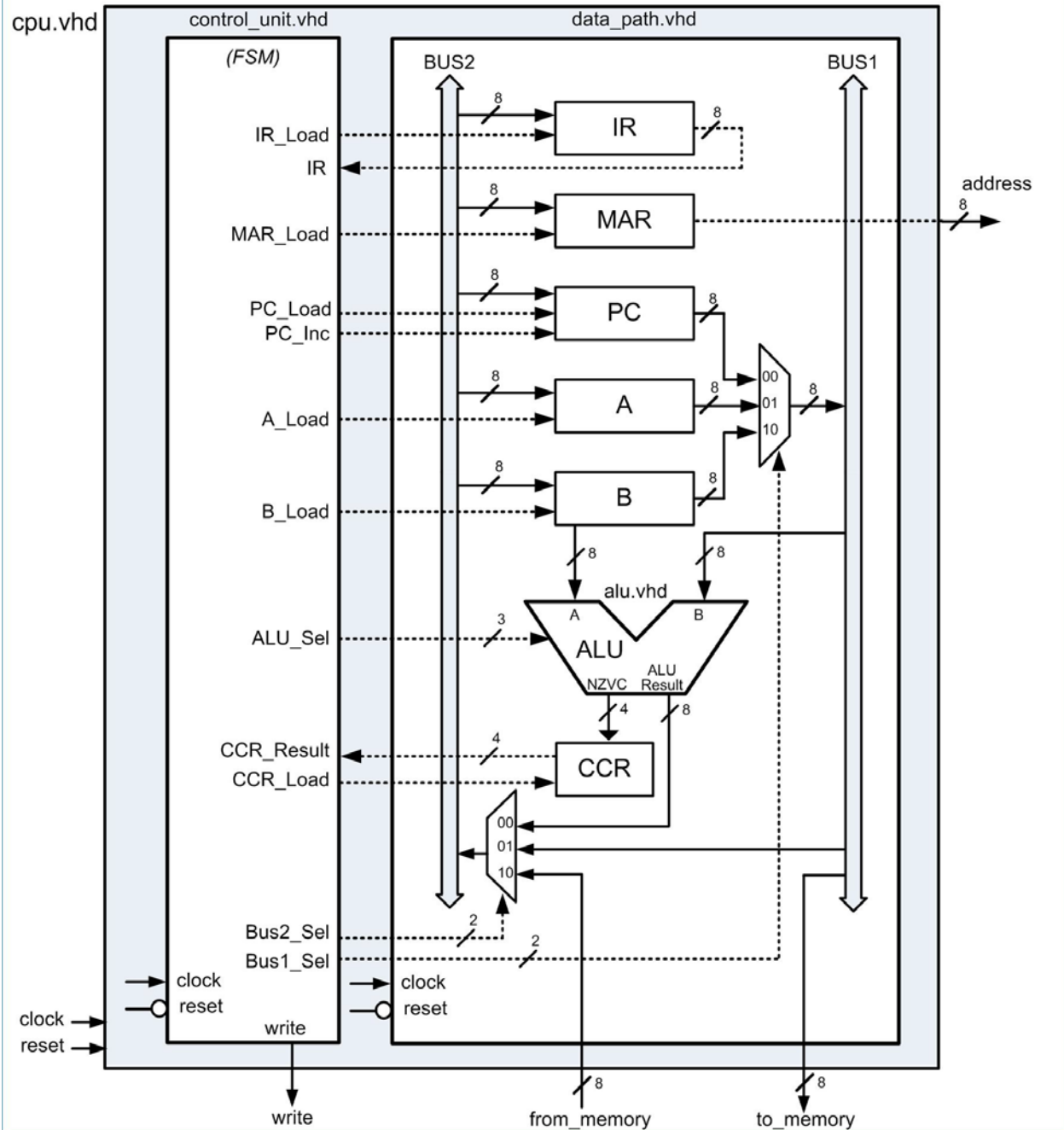


Figure 13.2 Structure of `cpu.vhd`

Example: Memory System Block Diagram for the 8-Bit Computer System

The following is the block diagram for the memory system of our 8-bit computer system example.

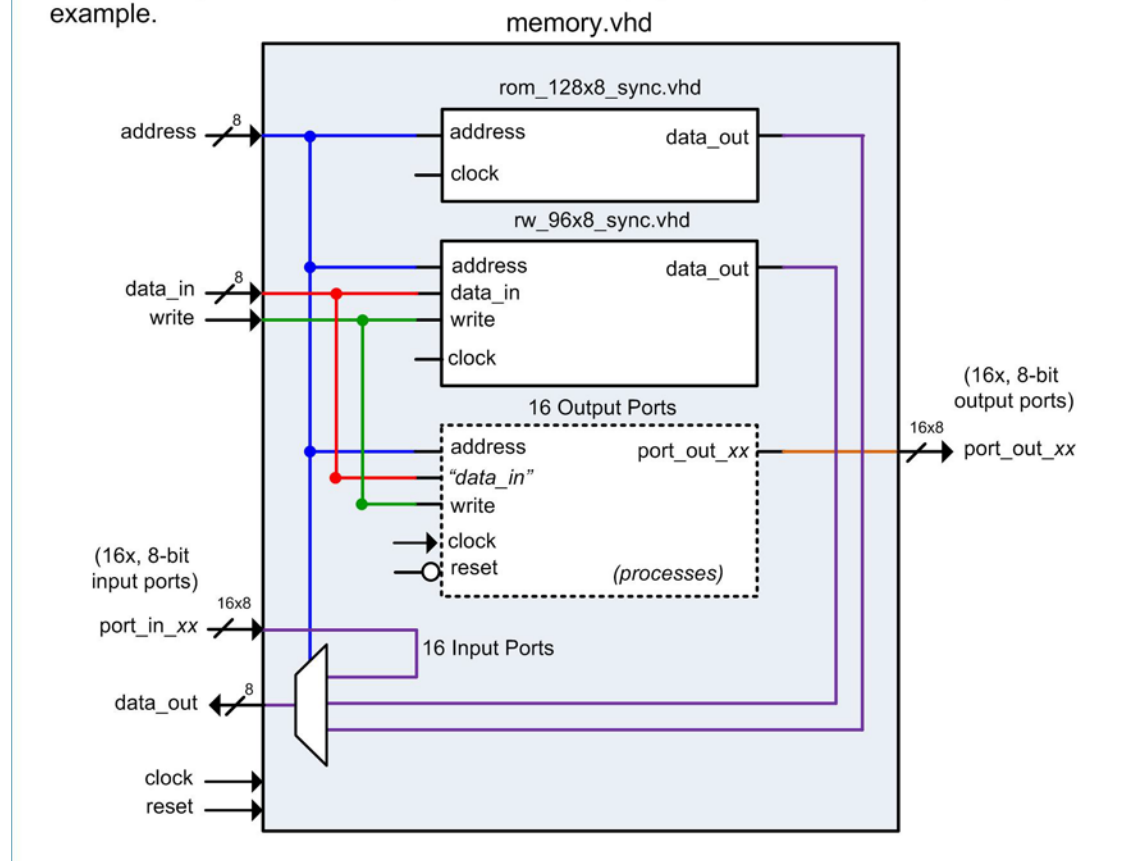


Figure 13.3
Structure of `memory.vhd`

Create the VHDL Shell for the Computer in ModelSim

Launch ModelSim and create a new project for the computer system. Design the structural shell of the computer system based on the above block diagrams. You will need to create 8x VHDL files named as follows:

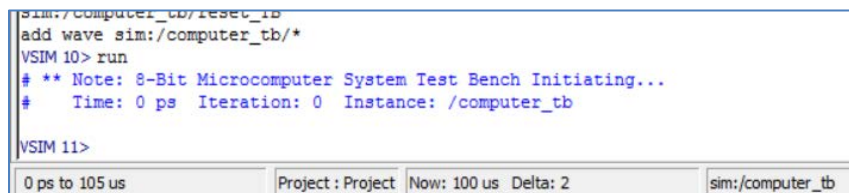
```
computer.vhd
├── cpu.vhd
│   ├── control_unit.vhd
│   ├── data_path.vhd
│   └── alu.vhd
└── memory.vhd
    ├── rom_128x8_sync.vhd
    └── rw_96x8_sync.vhd
```

In each of these files, you should enter the libraries, entity, and the architectural shell. Within the architecture shell, you should include all of your component declarations, associated signal declarations, a begin statement, and all component instantiations. Compile all the files and fix any errors you encounter.

Run a ModelSim Simulation to Verify the VHDL Shell Structure

Recall that all that is needed for a VHDL model to compile and simulate is the correct syntax for the entity and architectural shell. This means you can verify the entities, internal signal declarations, and component instantiations

before any behavioral modeling is inserted in the architecture. Compiling your models will reveal any syntax errors in your structure. Running a simulation will load all of your files and check that each component call has the appropriate port mappings. The simulation will not have any waveform outputs, but it will verify there are no port mismatches in your structure. This will allow you to get all of your connections correct before moving onto the behavioral modeling of the computer. The simulation will only run if all of the component port mappings are correct. Once the simulation runs successfully, you will get a message in the transcript window similar to [Figure 13.4](#). You will get a series of warnings, but that is OK for now. If there is a port mapping mismatch, you will get an error loading the design and text describing where the mismatch is.



```

sim:/computer_tb/reset_tb
add wave sim:/computer_tb/*
VSIM 10> run
# ** Note: 8-Bit Microcomputer System Test Bench Initiating...
# Time: 0 ps Iteration: 0 Instance: /computer_tb
VSIM 11>
0 ps to 105 us      Project: Project      Now: 100 us Delta: 2      sim:/computer_tb

```

Figure 13.4
Transcript Window for Computer System showing Successful Load

Run the simulation on your VHDL shell using the `computer_TB.vhd` test bench provided. Fix any errors you encounter. Take a screenshot of your transcript window indicating that the simulation loaded successfully (just like [Figure 13.4](#)). Save the screenshot in JPG format with a descriptive name. **This screenshot image and your eight VHDL files created in this part satisfy the requirements for deliverable #1.**

13.1.5.2 Functional Simulation of the Four Basic Instructions

In this part, you are going to design and simulate the behavior of the computer to execute the four basic instructions LDA_IMM, LDA_DIR, STA_DIR, and BRA. This will involve completing the memory system and the data path, and implementing the control unit FSM paths for these four instructions. You should refer to section 13.3 in the textbook for details on this implementation.

Modeling the Memory System

The first model to create is the memory system. This will involve creating the behavior for the ROM and R/W subsystems. It will also involve creating the 16 processes for the output ports. Finally, you'll need to model all of the signal routing within `memory.vhd` to handle the input ports and `data_out` port. When modeling this system, use the map in [Figure 13.5](#) for the address locations of the various memory types. After creating the model for the memory system, you will insert test programs into the program memory section of your system to execute the instructions. You will use the programs provided in exercise problems 13.3.1 and 13.3.2 from the textbook.

Example: Memory Map for a 256x8 Memory System

The following is a memory map for an example 8-bit computer system.

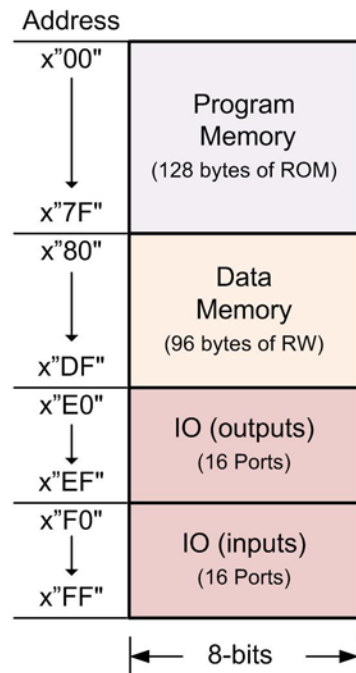


Figure 13.5
Memory Map for the 8-Bit Computer System

Modeling the Data Path

The next sub-system to model is the data path shown in [Figure 13.2](#). This sub-system is modeled using processes for each register and processes for the two multiplexers. You do not need to model any ALU or CCR functionality in this part of the exercise. You will do that later once you add arithmetic instructions.

Modeling the Control Unit: LDA_IMM Instruction

You are now going to implement the behavior to execute the *load register A using immediate addressing* (LDA_IMM) instruction. The state diagram for the control unit functionality for LDA_IMM is given in [Figure 13.6](#). After the model is complete, you can run a full computer simulation to verify the execution of this instruction. You should use the program provided in exercise problem 13.3.1 of the textbook, which continually loads constants into register A and then stores them to output ports. Since the first instruction in this program is LDA_IMM, you can run this program with only this instruction complete. The computer will hang after the first instruction, but you'll be able to verify that LDA_IMM is working properly. Your simulation waveforms should look like [Figure 13.7](#).

Example: State Diagram for LDA_IMM

The following is the state diagram for LDA_IMM. This load instruction will move information from memory into register A. Immediate addressing implies that the information to be put into A is provided as the operand of the instruction.

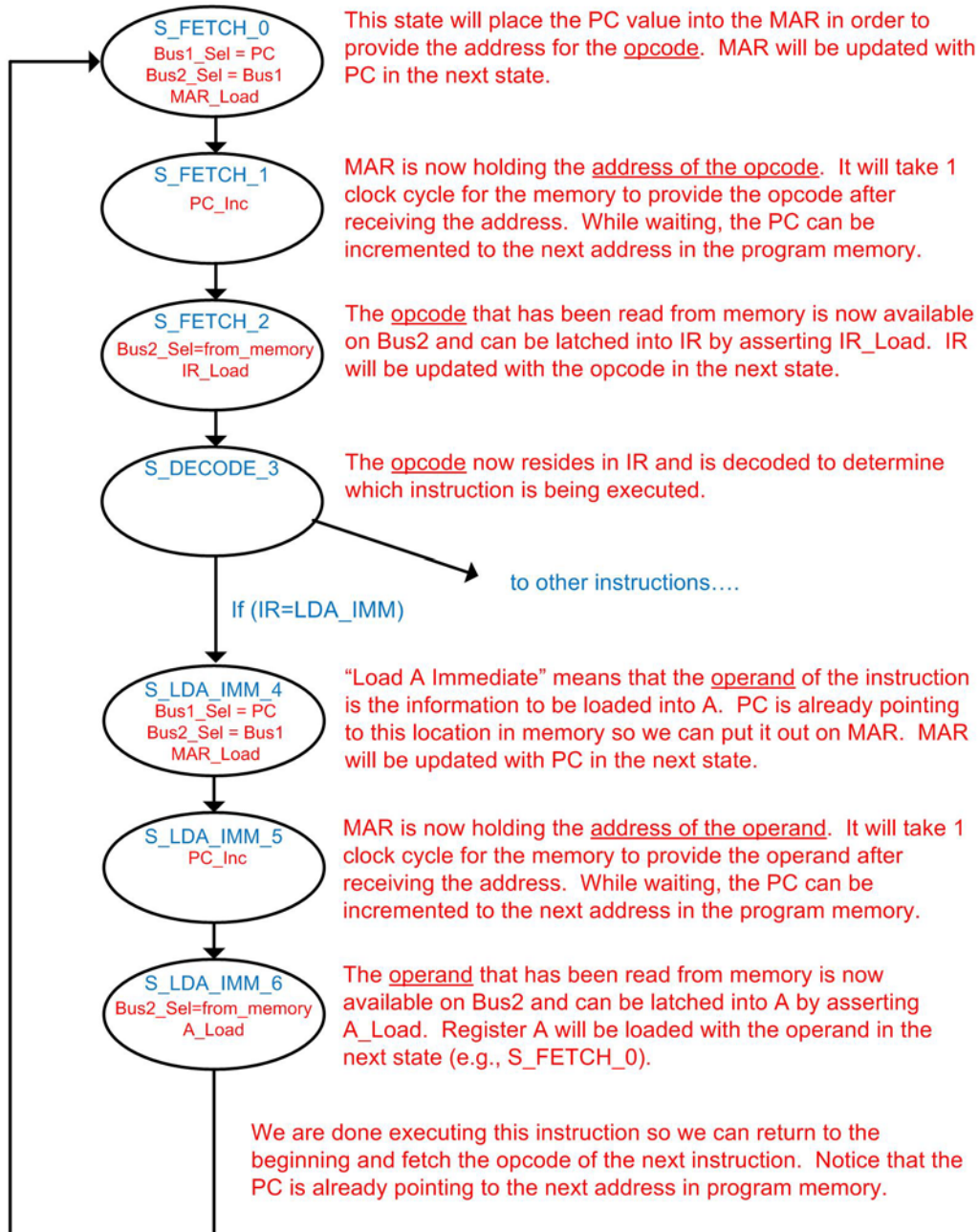


Figure 13.6
State Diagram for the LDA_IMM Instruction

Example: Simulation Waveform for LDA_IMM

Let's look at the timing diagram when executing the following load instruction located at addresses x"00" and x"01" in program memory. The opcode for this instruction is x"86".

LDA_IMM x"AA"

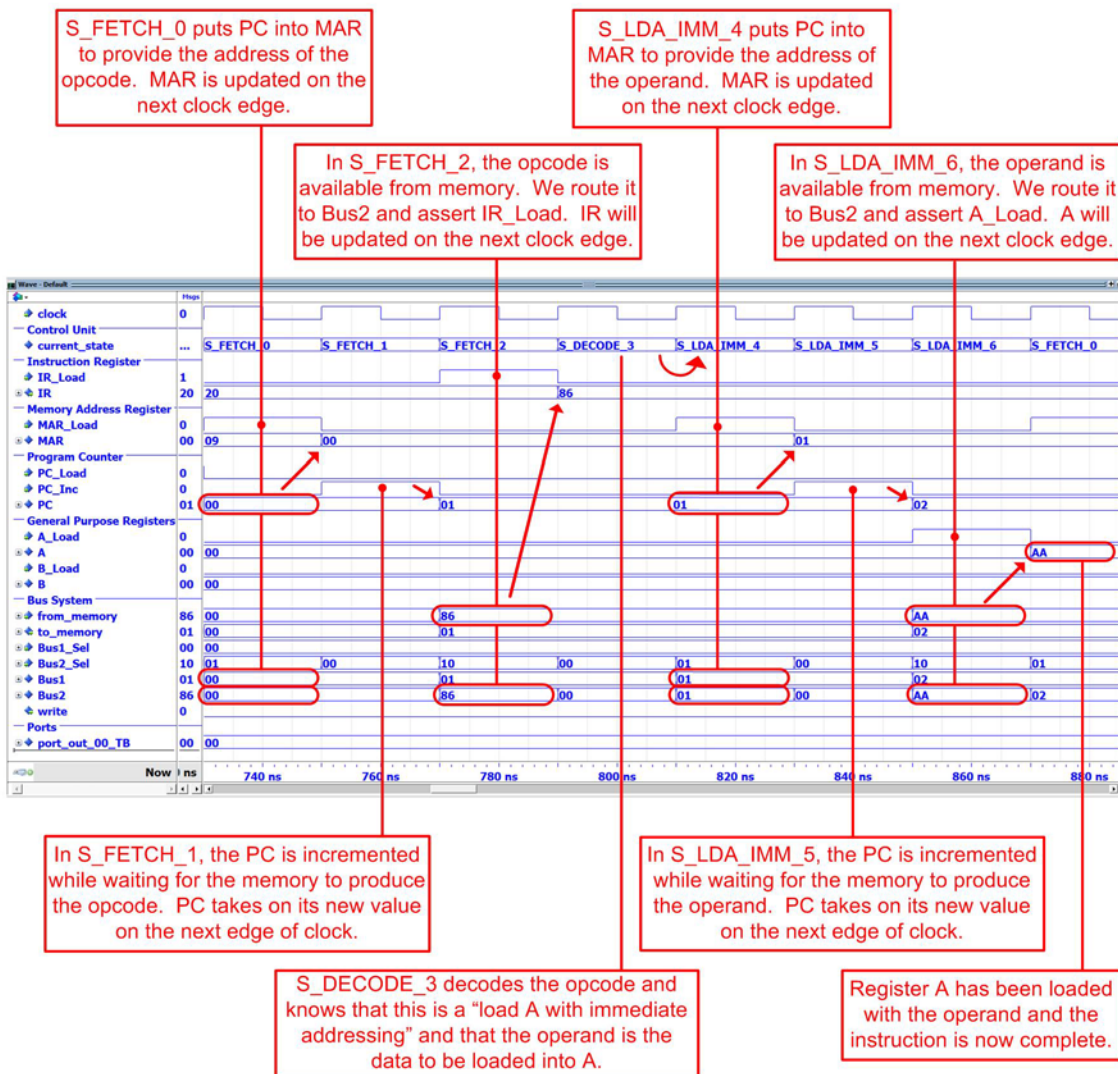


Figure 13.7
Simulation Waveforms for the LDA_IMM Instruction

Modeling the Control Unit: LDA_DIR Instruction

You are now going to implement the behavior to execute the *load register A using direct addressing* (LDA_DIR) instruction. The state diagram for the control unit functionality for LDA_DIR is given in Figure 13.8. After the model is complete, you can run a full computer simulation to verify the execution of this instruction. You should use the program provided in exercise problem 13.3.2 from the textbook, which continually loads values from input port 1 and stores them to output port 1. Since the first instruction in this program is LDA_DIR, you can run this program with only this instruction complete. The computer will hang after the first instruction, but you'll be able to verify that LDA_DIR is working properly. Your simulation waveforms should look like Figure 13.9.

Example: State Diagram LDA_DIR

The following is the state diagram for LDA_DIR. This load instruction will move information from memory into register A. Direct addressing implies that the information to be put into A is located at the address provided as the operand of the instruction.

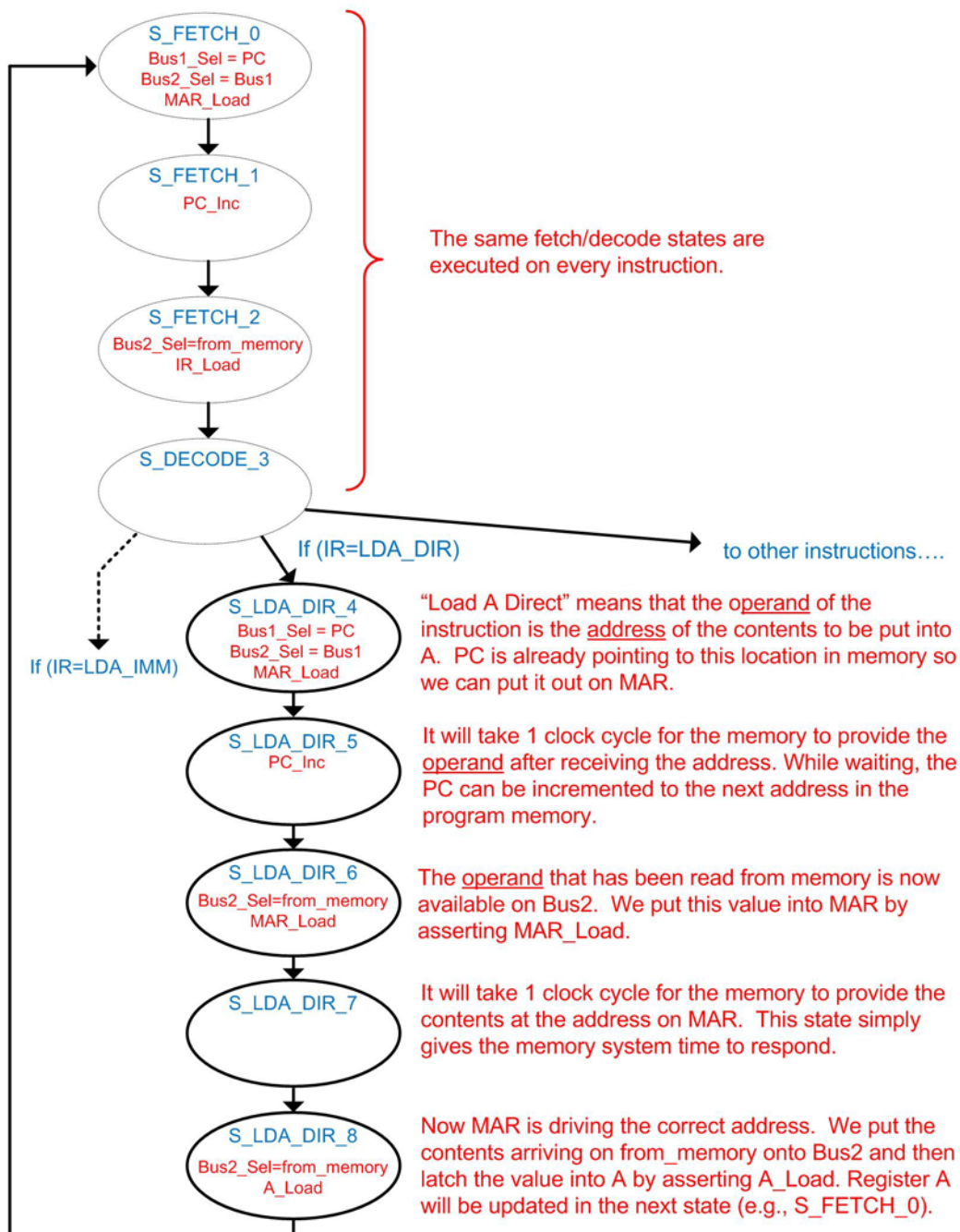


Figure 13.8
State Diagram for the LDA_DIR Instruction

Example: Simulation Waveform for LDA_DIR

Let's look at the timing diagram when executing the following load instruction located at addresses x"08" and x"09" in program memory. The opcode for this instruction is x"87". The address x"80" is in data memory, which in this example is already holding x"AA" prior to this instruction.

LDA_DIR x"80"

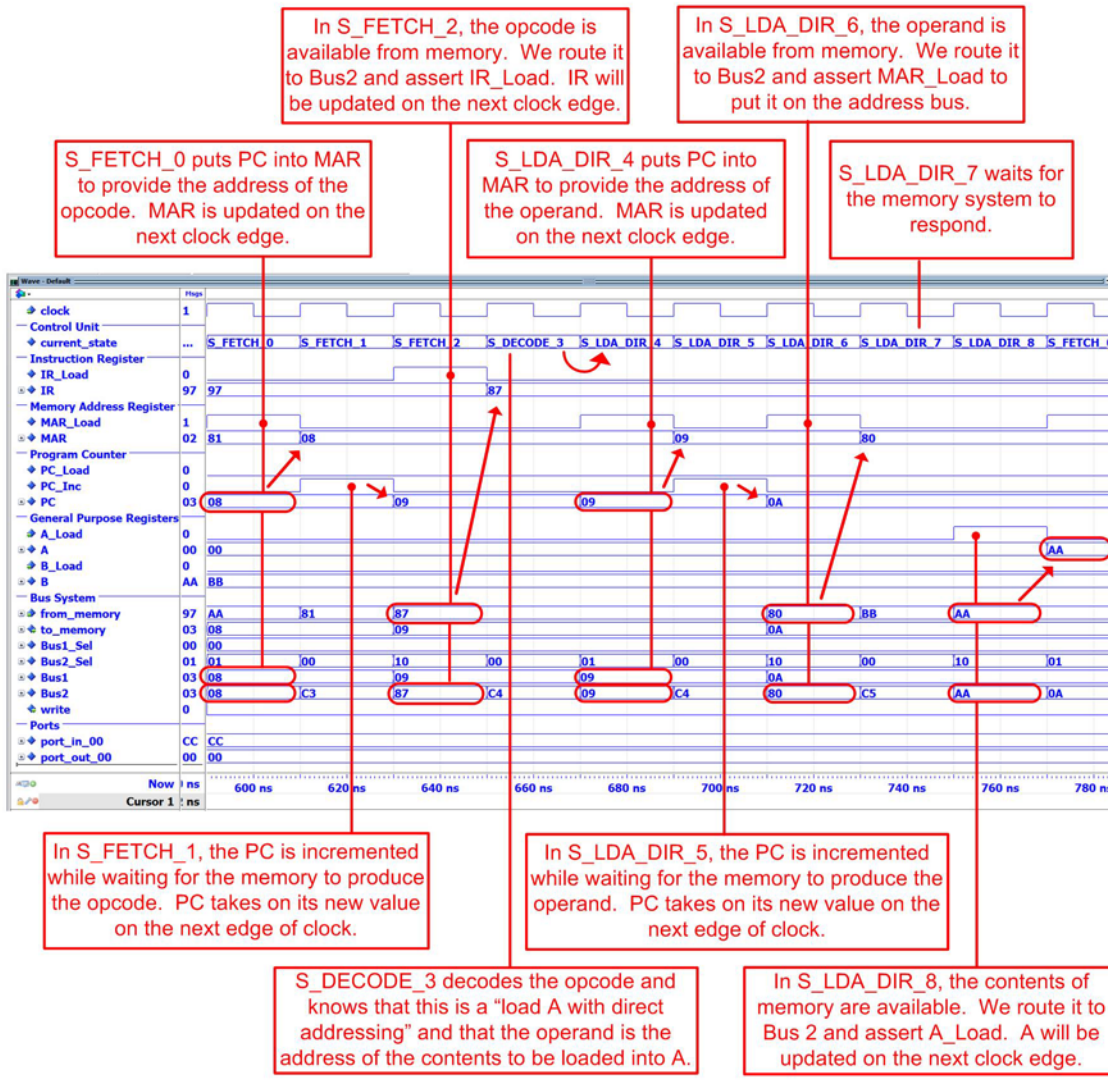


Figure 13.9
Simulation Waveforms for the LDA_DIR Instruction

Modeling the Control Unit: STA_DIR Instruction

You are now going to implement the behavior to execute the *store register A using direct addressing* (STA_DIR) instruction. The state diagram for the control unit functionality for STA_DIR is given in [Figure 13.10](#). After the model is complete, you can run a full computer simulation to verify the execution of this instruction. You should use the program provided in exercise problem 13.3.1, which continually loads constants into register A and then stores them to output ports. Since you have already implemented LDA_IMM, you'll be able to run the program far enough to test STA_DIR. Again, the computer will hang after the STA_DIR instruction, but you'll be able to verify that STA_DIR is working properly. Your simulation waveforms should look like [Figure 13.11](#).

Example: State Diagram for STA_DIR

The following is the state diagram for STA_DIR. This store instruction will move information from register A into memory. Direct addressing implies that the operand provides the address of where to store A to.

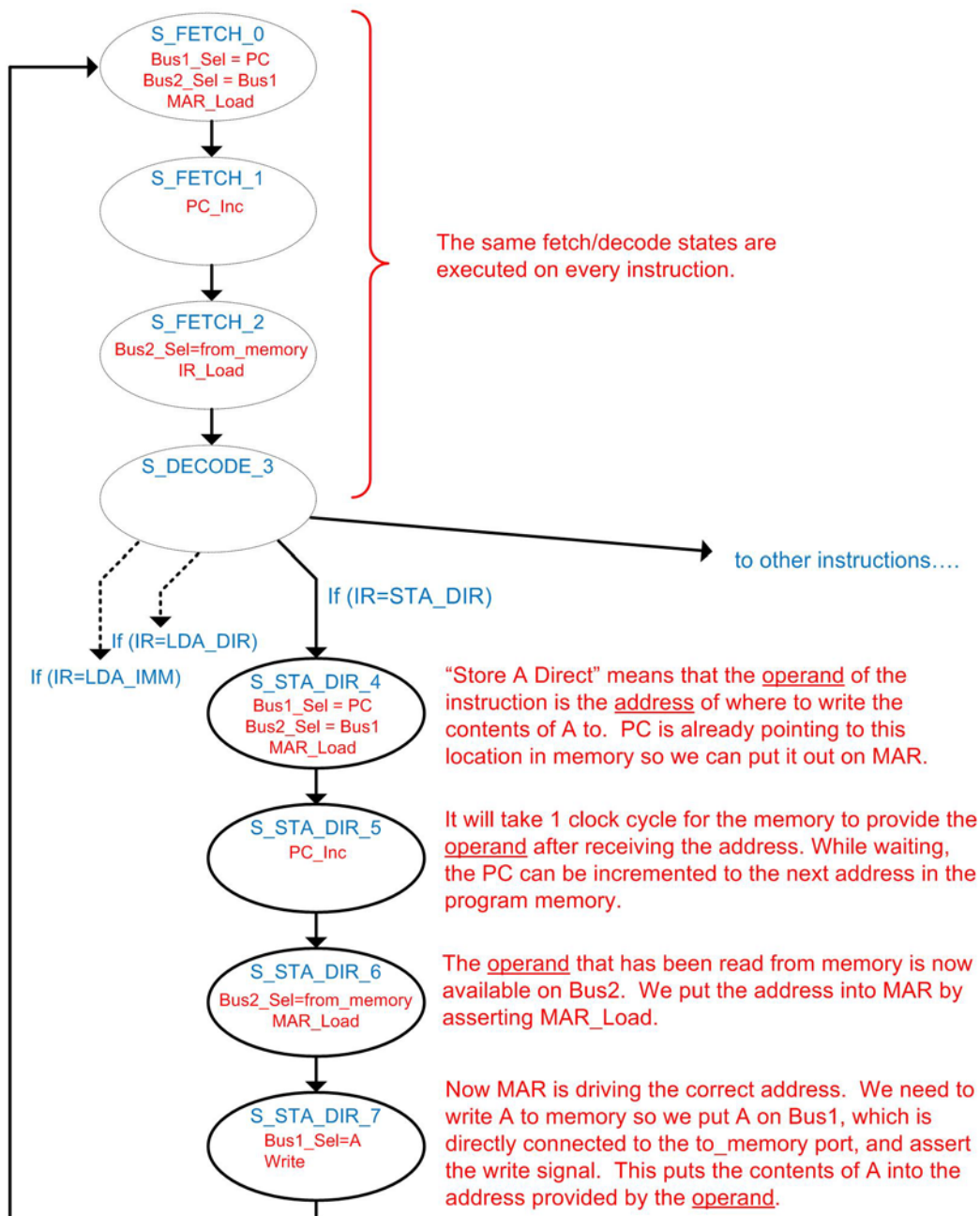


Figure 13.10
State Diagram for the STA_DIR Instruction

Example: Simulation Waveform for STA_DIR

Let's look at the timing diagram when executing the following store instruction located at addresses x"04" and x"05" in program memory. The opcode for this instruction is x"96". The address x"E0" is for port_out_00. A already contains x"CC".

STA_DIR x"E0"

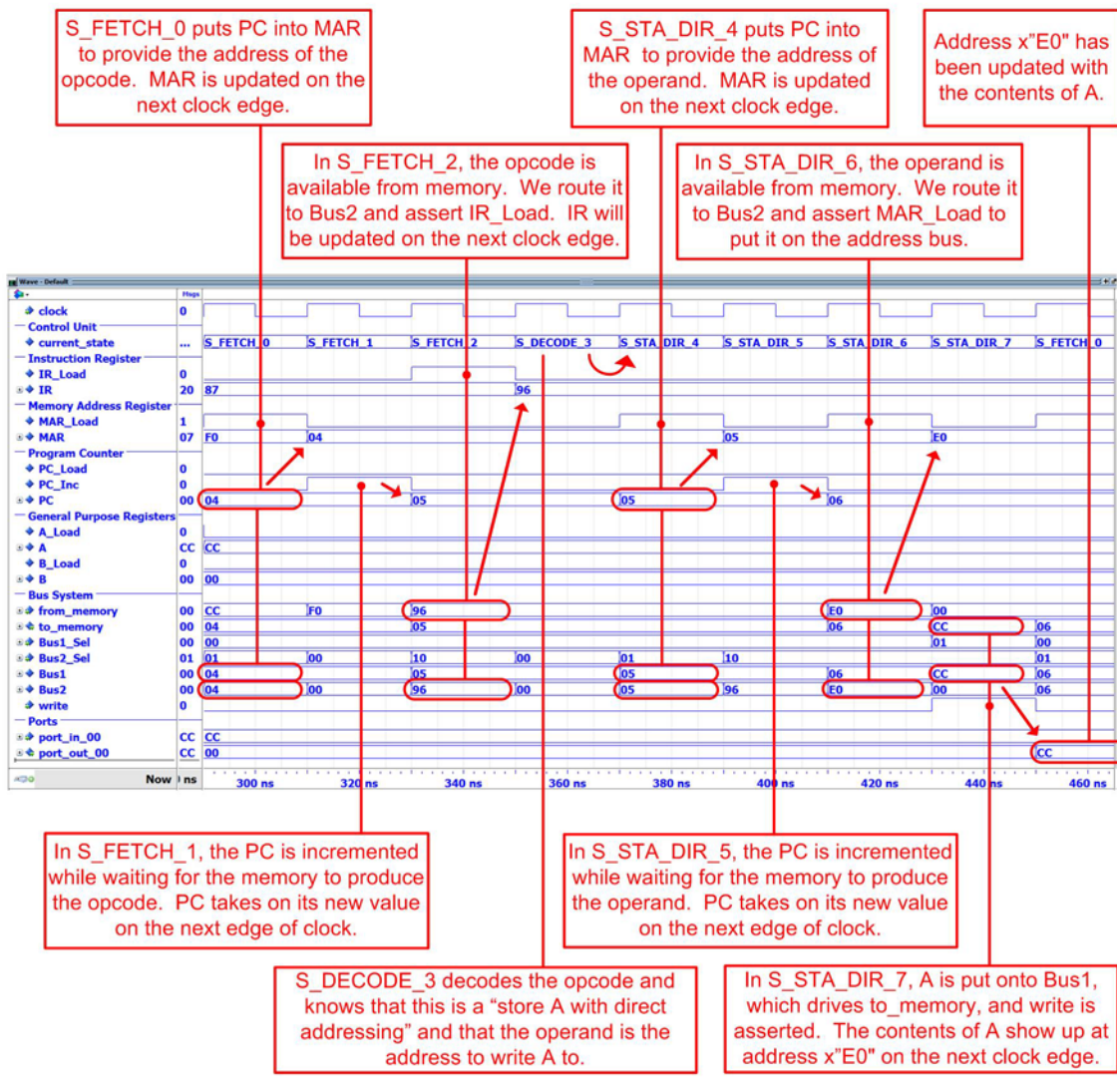


Figure 13.11
Simulation Waveforms for the STA_DIR Instruction

Modeling the Control Unit: BRA Instruction

You are now going to implement the behavior to execute the *branch always* (BRA) instruction. The state diagram for the control unit functionality for BRA is given in Figure 13.12. After the model is complete, you can run a full computer simulation to verify the execution of this instruction. You should use the program provided in exercise problem 13.3.1, which continually loads constants into register A and then stores them to output ports. Since you have already implemented LDA_IMM and STA_DIR, you'll be able to run the program far enough to test BRA. Once complete, this program should run indefinitely in a loop. Your simulation waveforms should look like Figure 13.13.

Example: State Diagram for BRA

The following is the state diagram for BRA. This instruction will load the program counter with the address supplied by the operand of the instruction. This has the effect of setting the address of the next instruction to be executed to a new location in program memory.

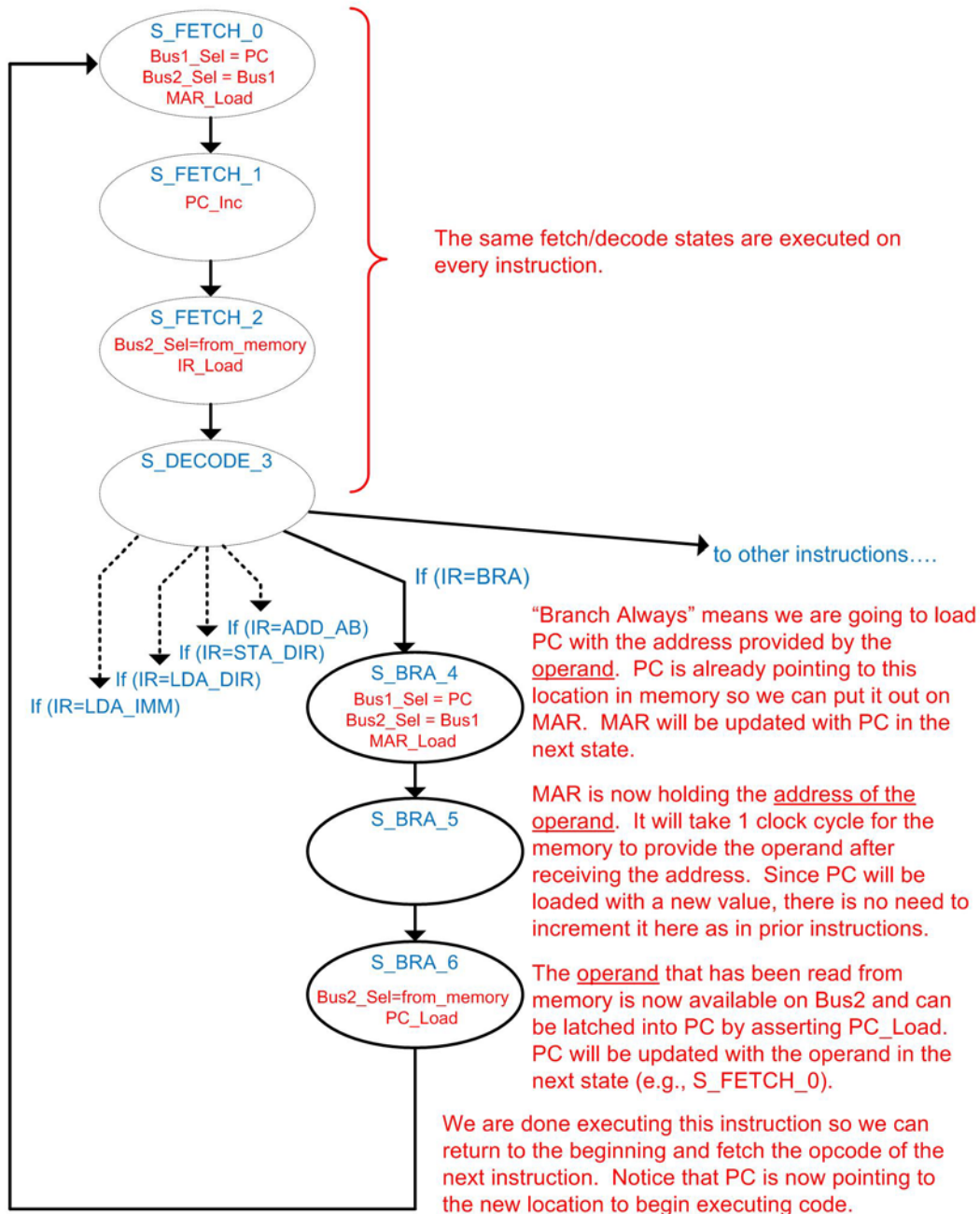


Figure 13.12
State Diagram for the BRA Instruction

Example: Simulation Waveform for BRA

Let's look at the timing diagram when executing the following branch always instruction located at addresses x"06" and x"07" in program memory. The opcode for this instruction is x"20".

BRA x"00"

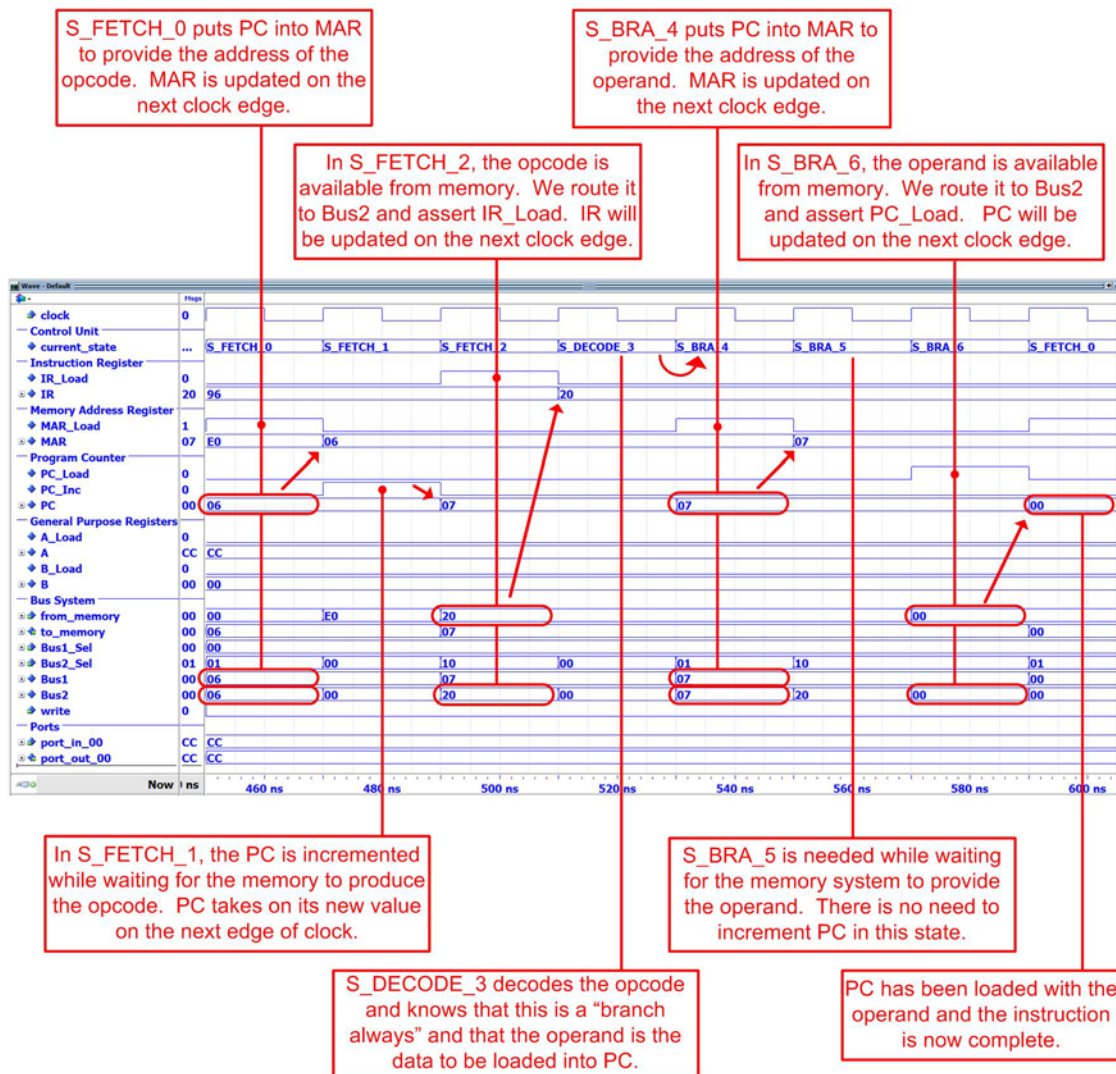


Figure 13.13
Simulation Waveforms for the BRA Instruction

Take screenshots of each of the four simulation waveforms verifying that each instruction is executing as expected. These should look very similar to [Figure 13.7](#), [Figure 13.9](#), [Figure 13.11](#), and [Figure 13.13](#). Save the screenshots in JPG format with descriptive names. **These four screenshot images and the three relevant VHDL files (memory.vhd, data_path.vhd, and control_unit.vhd) satisfy the requirements for deliverable #2.**

13.1.5.3 Implementation of Basic Instructions on the FPGA

You are now going to implement your computer system on the DE0-CV FPGA board. This will involve creating a new Quartus project and designing a `top.vhd` that instantiates your `computer.vhd` system. The purpose of the `top.vhd` is to connect the computer ports to the I/O on the DE0-CV board. It will also handle instantiating the `clock_div_prec.vhd` so that the computer can be run slow enough to observe its operation on the displays. Figure 13.14 shows the definition for the `top.vhd` that you will implement.

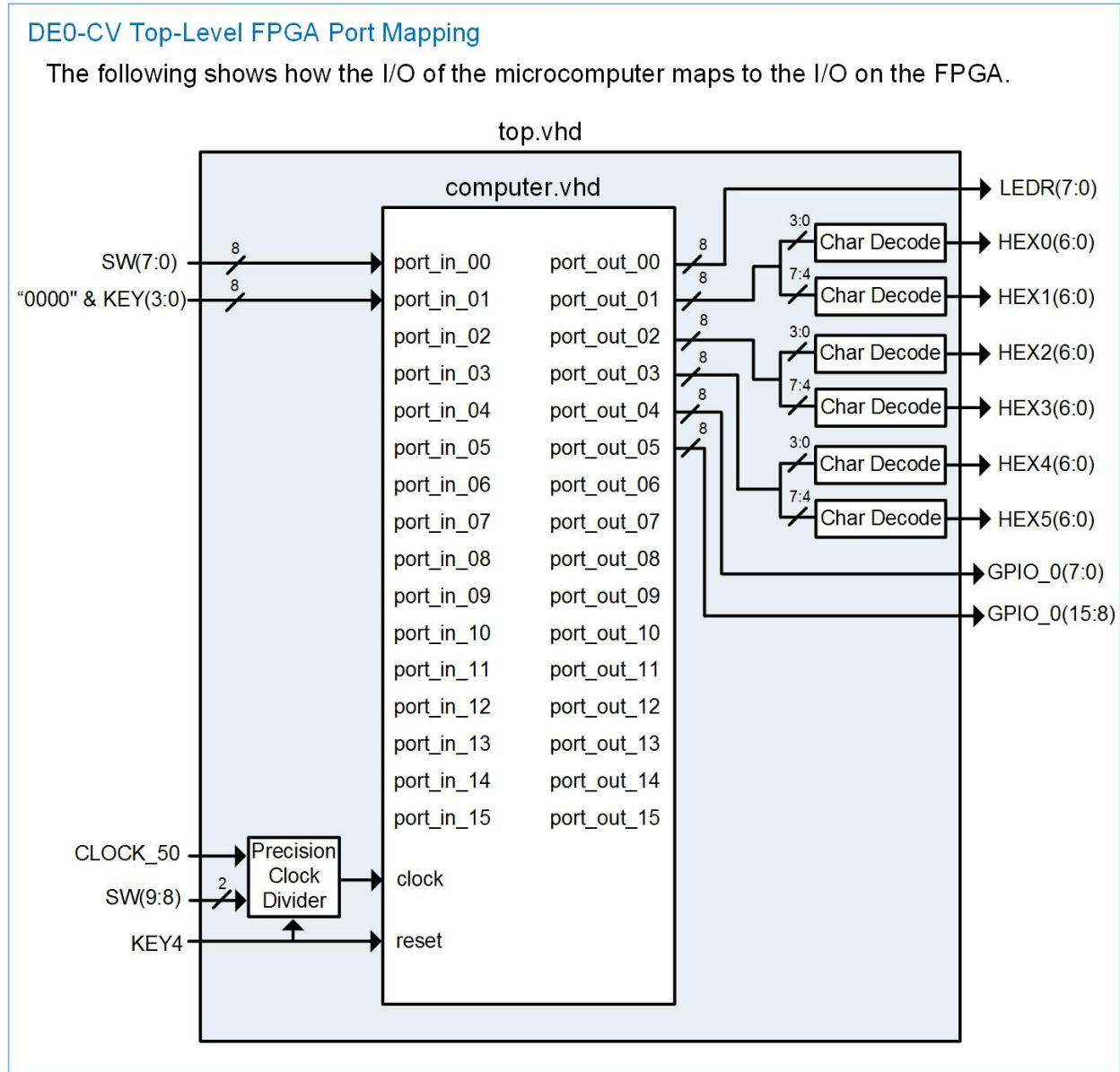


Figure 13.14
Top-Level for Computer System Implementation on the DE0-CV Board

You should create a program that continually reads from SW(7:0) and writes to LEDR(7:0), HEX(1:0), HEX(3:2), and HEX(5:4). You can accomplish this using the instructions you have already implemented in part 3. Note that the FPGA block diagram uses your clock_div_prec.vhd with select lines coming from SW(9:8) instead of SW(1:0) as in prior labs. Also notice that the character decoders are instantiated outside of the computer system.

Take a short video (<5 s) showing the proper operation of your design. You should run your clock at 10 Hz so that it is noticeable that a change on the slider switches doesn't appear on the LEDs and HEX displays instantaneously due to the way the computer instructions are executed as a series of states. **This video and your top.vhd file satisfy the requirements for deliverable #3.**

13.1.5.4 Implement Additional Instructions for the Computer System

You will now implement additional instructions to make the computer more functional. The instructions to implement are listed below. Note that the ALU instructions will require you to add paths through the control unit FSM in addition to behavior in the alu.vhd and CCR in the data_path.vhd. Also note that the conditional branch instructions base their operation on the CCR. This means your test program must perform an ALU instruction that alters the appropriate CCR flag just before the conditional branch instruction is executed.

Load & Store Instructions

- LDB_IMM (2% of exercise)
- LDB_DIR (2% of exercise)
- STB_DIR (2% of exercise)

ALU Instructions

- ADD_AB (2% of exercise)
- SUB_AB (2% of exercise)
- AND_AB (2% of exercise)
- OR_AB (2% of exercise)
- INCA (2% of exercise)
- INCB (2% of exercise)
- DECA (2% of exercise)
- DECB (2% of exercise)

Conditional Branch Instructions

- BEQ (2% of exercise)
- BCS (2% of exercise)
- BVS (2% of exercise)
- BMI (2% of exercise)

For each of instruction that is implemented, you must provide a **state diagram**, **ModelSim simulation waveform**, and a **demonstration of the instruction used on the DE0-CV Board** to satisfy the deliverable for part 4.

CONCEPT CHECK

Lab 13.1 After completing this lab exercise, can you:

- Design a VHDL model for an 8-bit computer system?
- Perform functional simulations to verify the proper execution of instructions by observing simulation waveforms?
- Implement the computer on an FPGA and verify the proper execution of instructions by observing signals on the various I/O of the DE0-CV board?