Lab Manual for

COE 203: Digital Design Lab

Table of Contents

1. Prototyping of Logic Circuits using Discrete Components

Objectives

- Introduction to ICs, logic families, 74xx, 54xx
- Learn how to read Data sheets, IC diagrams etc.
- Partial familiarization with FPGA board, pins, switches, LEDs and power supply connections.
- Implementation of a simple combinational circuit using ICs

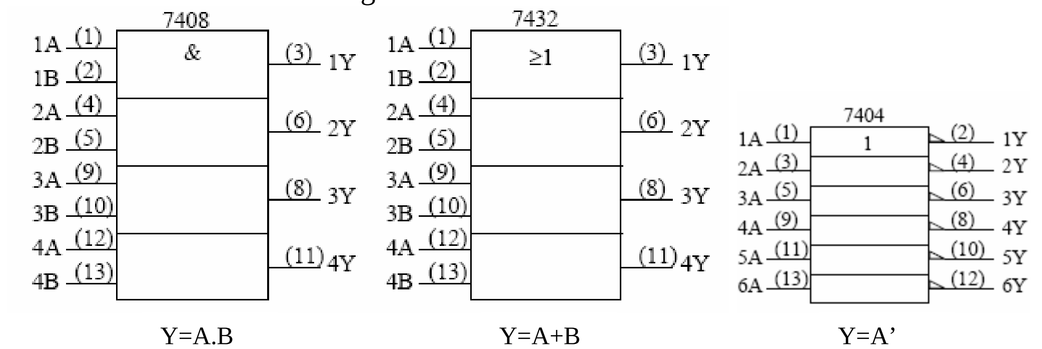
Material

- ICs – 7400, 7432, 7408 etc
- Wire Stripper
- Prototyping board with power and ground connections

Logic Gates and Integrated Circuits

In COE 203, you are required to work on digital circuits. Digital circuits are hardware components that are implemented using transistors and interconnections in complex semiconductor devices called *integrated-circuits*. Digital circuits work in binary logic domain which uses two discrete values, **TRUE** (High) and **FALSE (Low)**. We can also refer to these values as **1(High)** and **0 (Low)**.

Logic-Gates are basic building blocks of digital circuits. Using these building blocks, complex functions or larger digital circuits can be built. Examples of the basic logic gates are **AND, OR, NOT, NAND and NOR**. A complex gate such as an **XOR** gate can be built out of these basic gates. Each logic gate is actually implemented in part of an integrated circuit (IC), with each gate made using several transistors. For the most part we do not need to concern ourselves with the actual circuitry inside each gate, only the interconnections between individual gates. Usually each IC package contains several individual gates. The 7408 chip implements 4 two input AND gates and is commonly referred to as a “Quad two input AND” chip. Similarly the 7432 chip is a “Quad 2 input OR” chip while the 7404 chip is a “Hex Inverter” since it contains 6 inverters. The IEEE standard logic symbols for each of these devices are shown in Figure 1.



Y=A.B Y=A+B Y=A’

Figure 1: Standard Logic Symbols

Each of the chips in Figure 1 is available in 14 pin Dual-In-Line packages or DIPs. In a popular logic family called TTL (Transistor-Transistor Logic), the low logic level is assigned to 0V and the high logic level is assigned to 5V. Each IC or chip has an ID number that can be referenced in IC Data Book. From the book, you can get the pin configuration of that chip. The pin numbers assigned to each logic signal are shown inside brackets in the figure. The pins are numbered as shown in Figure 2. Pin 1 is usually identified as the pin to the left of an indentation or cutout in one end of the chip that is visible when the chip is viewed from the top. Occasionally, it is also identified by a printed or indented dot placed just next to it.

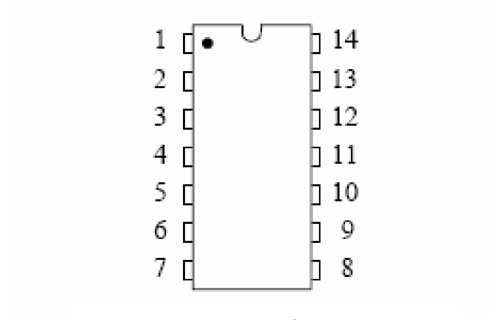


Figure 2: Identifying Pin 1

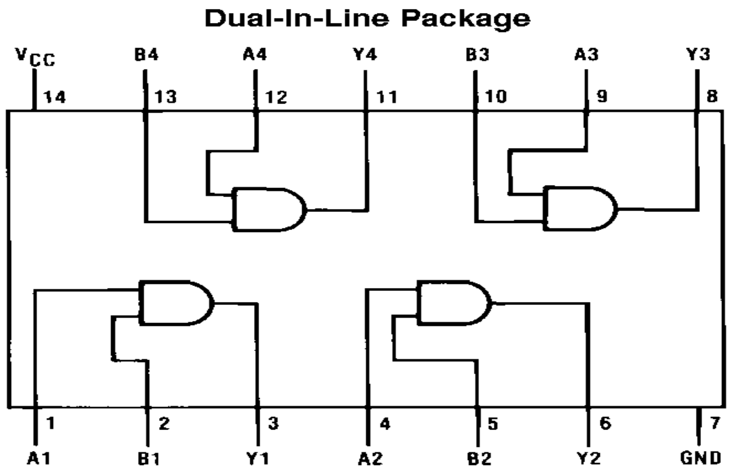


Figure 3: A Quad 2-input AND gate chip

Figure 3, shows an IC which contains four 2-input AND gates. You will notice that pins 7 and 14 show no connections. In 14 pin DIP packages, pin 7 is usually connected to ground (Gnd), and pin 14 is usually connected to the power supply (Vdd). These connections must be made or the chip will not work. Take care not to connect pin 7 to power and pin 14 to ground, or to connect the outputs of two or more gates together. In complex ICs more than one pin can be dedicated for power (VDD) as well as ground. In the simpler gates that we will be using in this experiment, the ICs require only one pin for power (VDD) and another for ground (GND). The power supply (VDD) voltage is typically +5Volts, 3.3 Volts or 2.5 Volts. The ground is typically connected to 0 Volts.

The Prototyping Board

The circuit is constructed on the breadboard section of the prototyping board. A breadboard is used to rapidly create an experimental or prototype circuit. It consists of an array of holes in which wires or component leads can easily be inserted. Rows of five or six holes are electronically connected to form a single node as shown in Figure 4. When a component lead is inserted into one of the holes, anything inserted into any of the remaining four holes will be connected to that lead. Nodes can be connected to each other using wires with 1/4" of insulation stripped from both ends. The holes are spaced 100 mms (0.1 inch) apart, which is the standard spacing of the pins on a DIP package. The breadboard has a groove down the center separating one side from the other. When inserting a chip into the breadboard, make sure it fits into the central groove as shown in the figure; otherwise the pins on opposite sides of the chip will be connected. Press the chip down until it touches the surface of the breadboard. Devices inserted on the breadboard can be connected to components on the prototyping board by using wires between the device and the J1 connector.

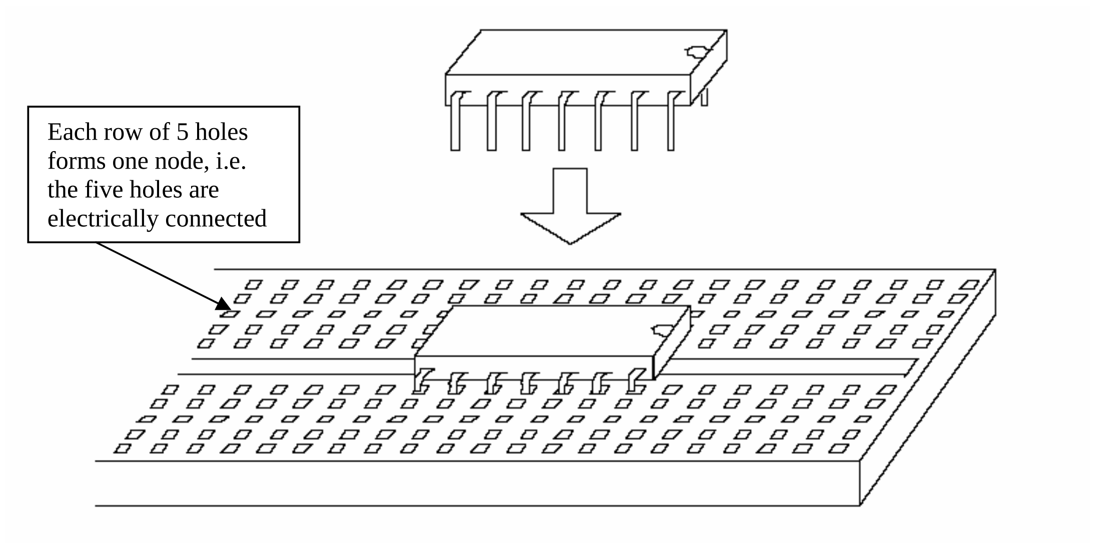

Each row of 5 holes forms one node, i.e. the five holes are electrically connected

Figure 4: Placing DIP devices on a breadboard

Design Specifications

You will construct a 1-bit full adder circuit. The full adder is a common circuit used in many designs both small and large (including processors). The function of the full adder is quite simple – add two, one-bit numbers. This may seem like a simple process, but the full adder is designed to be cascaded to compute addition on (arbitrarily) larger numbers.

Your circuit must have three inputs, **A**, **B**, and **Cin**. It will have two outputs, **Sum** and **Cout.** A block diagram is shown in Figure 5.

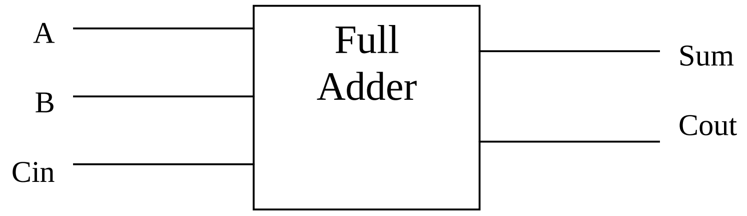


Figure 5: A block diagram of the circuit

Pre-Lab

1. Fill in the truth table given below.
2. Use any circuit minimization scheme (K-Maps, Boolean Algebra etc) and find out the equations for each output. Mention all your steps clearly.
3. Draw the circuit using gates

Show it to your instructor for approval prior to beginning to construct the circuit in the lab.

Truth Table:

A	B	Cin	Sum	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

In-Lab

1. Find out which chips do you need in order to construct your circuit
2. Check the datasheet in the lab and find out the number of the chips
3. Pick the chips from the shelf and test them using the IC tester.
4. Place the chips on the bread board carefully.
5. Connect the Vcc pins of the chips to the +5V on the board.
6. Connect the GND pins of the chips to the GND on the board.
7. Build your logic diagram by making appropriate connections.
8. Use 3 switches as your inputs to the 1-bit full adder.
9. Connect the two outputs to 2 LEDs.
10. Use the switches on the board to verify your circuit.
11. Print a copy of the grading sheet and get it signed by the instructor before you leave the lab.
12. Complete the lab report and submit it before the next lab

Lab Report

You are responsible for documenting your work and your report must include (at a minimum) the following:

1. Cover sheet (showing your name, ID and Title of the experiment)
2. Introduction - what you did
3. Description of the design - how you did it (You can show the Boolean Equations and write down the steps which you took to reach those equations. List any design aids (such as Logic Works) and how you used them
4. Features of final result – the final design is working properly or not. If not, where do you think the errors lie?
5. Problems Faced
6. Conclusion - Any comments on the experiment itself are most appreciated.
7. Answer the post-lab questions.

Submit it to the instructor **before** the next lab session. Attach the grading sheet with your lab report at the time of submission.

Post-Lab Questions

1. How many chips and wires will you need if you are asked to implement a 4-bit, 8-bit and 16-bit adder?
2. **Bonus**: If the probability of making a wiring mistake is p/wire, what is the probability of making a wiring mistake for the above adders?

Grading Sheet

There are 100 points awarded for this laboratory as follows:

Points Possible	Points Awarded	Topics Addressed
20	/20	Pre-Lab Truth Table(4) Logic Minimization (8) Boolean Equations (2) Circuit Diagram (5) Organization, Clarity (1)
60	/60	Lab Work Correct Wiring (30) Handling of Components (10) Verifying on the board (20)
20	/20	Lab Report Contents (10) Spelling, Grammar, and Style (10)
100	/100	Total

2. Prototyping of Logic Circuits using EEPROMs

Objectives

- Introduction to EEPROMs
- Learn how to use EEPROM Programmers
- Implementation of a simple sequential circuit using EEPROMs

Material

- Chips – 7474 or 74175, AT28C64
- Wire Stripper
- Prototyping board with power and ground connections
- EEPROM Programmer

Designing circuits using EEPROMs

A **Read-Only Memory (ROM)** is a combinational circuit with n inputs and b outputs. The inputs are called **address inputs** and the outputs are called **data outputs**. A 2^n x b ROM stores the truth table of an n-input, b-output logic function. ROM is a **non-volatile** memory. Its contents are preserved even if no power is applied

In a **Programmable ROM (PROM)**, the user may store data values (i.e. program the PROM) in just a few minutes using a PROM programmer. The ROM which we will be using in this lab is an **Electrically Erasable PROM (EEPROM).** In these ROMs, the contents can be erased electrically. We will need an EEPROM Programmer to alter the contents of the ROM.

There are certain distinct advantages of a ROM based circuit. A ROM-based circuit is usually faster than a circuit using multiple SSI/MSI devices and PLDs. The program that generates the ROM contents can easily be structured to handle unusual or undefined cases. A ROM function is easily modified just by changing the stored pattern, without changing any external connections. On the negative side, for small circuits, a ROM-based solution will consume more power as compared to implementing the circuit with MSI components.

Design Specifications

You are required to construct a sequence recognizer circuit for the sequence **0010.** The sequence recognizer is a common circuit used in many applications. The circuit should have a single-bit serial input and a single-bit output. As soon as the last four bits become 0010, the output should become 1; otherwise the output remains 0. You will need flip-flops to 'remember' the previous inputs. The circuit should be implemented using an EEPROM and flip-flops. A block diagram is given in Figure 1.

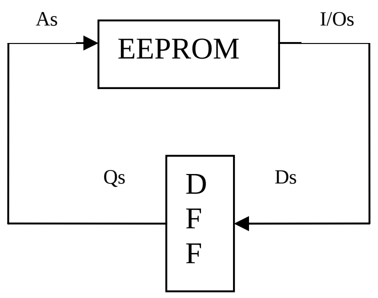


Figure 1: A block diagram of the circuit

Pre-Lab

1. Construct a state diagram having 4 states, for a circuit which recognizes the sequence 0010. Use the procedure mentioned in your textbook of COE 202
2. Suppose the input to your circuit is In and the output is Out. Fill in the state table given below.
3. Study the datasheet of the EEPROM and D-Flip Flop from the lab guide. Find out what should the control signals be connected to.

Show your work to your instructor for approval prior to beginning to construct the circuit in the lab.

State Table:

A	B	In	A+	B+	Out
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

In-Lab

1. Start the software for the EEPROM Programmer
2. Insert an EEPROM chip in the socket and verify that it is empty.
3. Press **Edit** and enter the values from your state table at successive addresses starting from address 0. (Suppose for input 000, your state table has the value 110; enter 06 for address 0).
4. Leave the unused addresses as it is.
5. Press **Program** and the values will be stored on your EEPROM.
6. Place the EEPROM and D-Flip Flop chips on the bread board carefully.
7. Refer to the datasheets of D-flip flop and EEPROM and complete your circuit by making appropriate connections.
8. Use a switch as your input **In**. This should be connected to A_0 pin of the EEPROM.
9. Use an LED as your output **Out**. This should be connected to I/O_0 pin of your EEPROM.
10. Connect the unused address pins of the EEPROM to GND.
11. Connect the Clock of the Flip flop chip to SW2
12. Use the switches on the board to verify your circuit.
13. Print a copy of the Grading sheet and get it signed by the instructor before you leave the lab.
14. Complete the lab report and submit it before the next lab

Lab Report

You are responsible for documenting your work and your report must include (at a minimum) the following:

1. Cover sheet (showing your name, ID, sec.# and Title of the experiment)

2. Introduction - what you did
3. Description of the design - how you did it. List any design aids (such as Logic Works) and how you used them
4. Features of final result – the final design is working properly or not. If not, where do you think the errors lie?
5. Problems Faced
6. Conclusion - Any comments on the experiment itself are most appreciated.
7. Answer the post-lab questions.

Submit it to the instructor **before** the next lab session. Attach the grading sheet with your lab report at the time of submission.

Post-Lab Questions

1. How many states are needed if you have to detect an 8-bit sequence?
2. What is the largest sequence which can be detected using this EEPROM

Grading Sheet

There are 100 points awarded for this laboratory as follows:

Points Possible	Points Awarded	Topics Addressed
10	/10	Pre-Lab State Diagram(5) State Table(4) Organization, Clarity (1)
70	/70	Lab Work Correct Wiring (30) Handling of Components (10) Verifying on the board (30)
20	/20	Lab Report Contents (10) Spelling, Grammar, and Style (10)
100	/100	Total

3. Introduction to FPGA Design Flow

Objectives

- Learn to use the Xilinx software to simulate the logic from Lab 1 using schematic capture.
- Learn how to program your Digital Logic Board.
- Verify your truth tables from Lab 1.

Materials

- Xilinx ISE 7.1 software (installed on the lab computers)
- Digital Logic Board – Digilent Inc.
- Programming cable

Overview

7400 series logic chips, used in Experiment 1, were the most common digital logic devices around for many years. In more recent time however, most digital logic implementations have moved towards either programmable logic devices (PLDs) or field programmable gate arrays (FPGAs). The Spartan-3 starter board is made by Digilent, and contains the Spartan-3 FPGA combined with some simple interface electronics like switches, LEDs and seven-segment displays etc. You can find more information about this board in the Lab Guide. To show a comparison of 7400 series logic implementations, and more modern systems, you will be implementing the logic of the full adder from Experiment 1 on your Digilent board.

You will construct a simple full adder using schematic editor. The full adder is a common circuit used in many designs both small and large (including processors). The function of the full adder is quite simple – add two, one-bit numbers. This may seem like an inane process, but the full adder is designed to be cascaded to compute addition on (arbitrarily) larger numbers.

Design Specifications

Your circuit must have three inputs, **A**, **B**, and **Cin**. It will have two outputs, **Sum** and **Cout.** The inputs **'A'** and **'B'** are two, 1-bit numbers that the addition will be preformed on. The output is located at the output labeled '**Sum'**. There are also two other I/O pins, '**Cin'** and '**Cout'**. These are called "Carry In" and "Carry Out", respectively. They are used for cascading adders together.

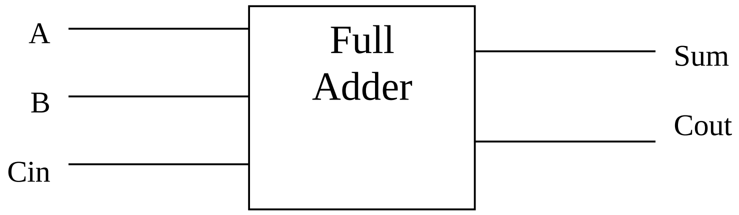


Figure 3: A Block Diagram of the circuit

Pre-Lab

1. Fill in the truth table given below.
2. Use any circuit minimization scheme (K-Maps, Boolean algebra, etc.) and find out the equations for each output. Mention all your steps clearly.

Show it to your instructor for approval prior to beginning to construct the circuit in the lab.

Truth Table:

A	B	Cin	Sum	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

In-Lab

1. Follow the tutorial given in the Lab Guide for designing circuits using schematic editor

2. The following three steps must be shown to the instructor during the lab:
   - Complete Schematic
   - Functional simulation of the circuit using ModelSim
   - Verification on the board

3. Print a copy of the Grading sheet and get it signed by the instructor before you leave the lab.

Lab Report

A Lab Report will be required for this and every laboratory assignment for the entire semester. You are responsible for documenting your work and your report must include (at a minimum) the following:

1. Cover sheet (showing your name, ID and Title of the experiment)
2. Introduction - what you did
3. Description of the design - how you did it (You can show the Boolean Equations and write down the steps which you took to reach those equations. List any design aids (such as Logic Works) and how you used them
4. Implementation of design – which pin numbers you assigned to the inputs and outputs
5. Features of final result – the final design is working properly or not. If not, where do you think the errors lie?
6. Problems Faced
7. Conclusion - Was it a good/bad design/ implementation? **Why?** What would you do differently next time? Any comments on the lab itself are most appreciated.
8. Include the following in the appendix
9. Schematics – print out the schematic diagram from your Xilinx software
10. Simulations - show functional simulation using ModelSim
11. Performance metrics - a screen shot of your timing info from the "Post-Layout Timing Report" for which you will have to implement as well as synthesize your design. Find out the maximum clock speed and the longest logic delay.

Lab reports **must** be typed and must include enough information to recreate your design. Submit it to the instructor **before** the next lab session. Attach the grading sheet with your lab report at the time of submission.

Grading Sheet

There are 100 points awarded for this laboratory as follows:

Points Possible	Points Awarded	Topics Addressed
15	/15	Pre-Lab Truth Table(4) Logic Minimization (8) Boolean Equations (2) Organization, Clarity (1)
60	/60	Lab Work Complete Schematic (10) Functional Simulation (10) Bit-file Generation (25) Verifying on the board (15)
25	/25	Lab Report Contents (10) Appendices (10) Spelling, Grammar, and Style (5)
100	/100	Total

4. Traffic Light Controller

Objectives

- Practice on the design of clocked sequential circuits.
- Introducing practical aspects of logic implementation (switch debouncing, timing constraints etc.).

Overview

In this lab you are going to develop a Finite State Machine (FSM) for a traffic light controller that will control the operation of traffic lights similar to the one at the KFUPM main gate. The crossing is shown in Figure 1.

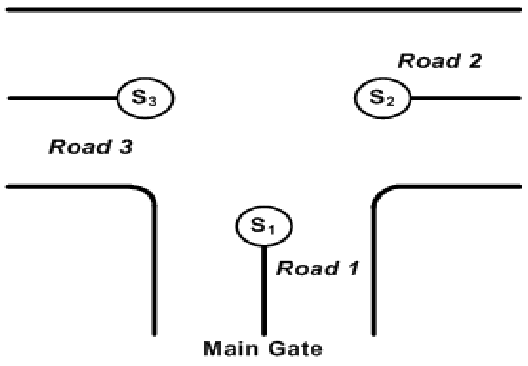


Figure 1: The Three Traffic Light Signals

Design Specifications

There are three traffic light signals (*S1*, *S2*, and *S3*), each alternating between two states, *RED* and *GREEN*. These signals control the traffic flow on the three roads, *road1*, *road2*, *road3* in four possible states as follows.

- In *STATE 0:* no traffic so give priority to the main gate's road. (*S1* = GREEN, *S2* = RED, *S3* = RED)
- In *STATE 1*, traffic coming through *road**1***. (*S1* = GREEN, *S2* = RED, *S3* = RED)
- In *STATE 2*, traffic coming through *road**2***. (*S1* = RED, *S2* = GREEN, *S3* = RED)
- In *STATE 3*, traffic coming through *road**3***. (*S1* = RED, *S2* = RED, *S3* = GREEN)

The last three states are shown in Figure 2 below.

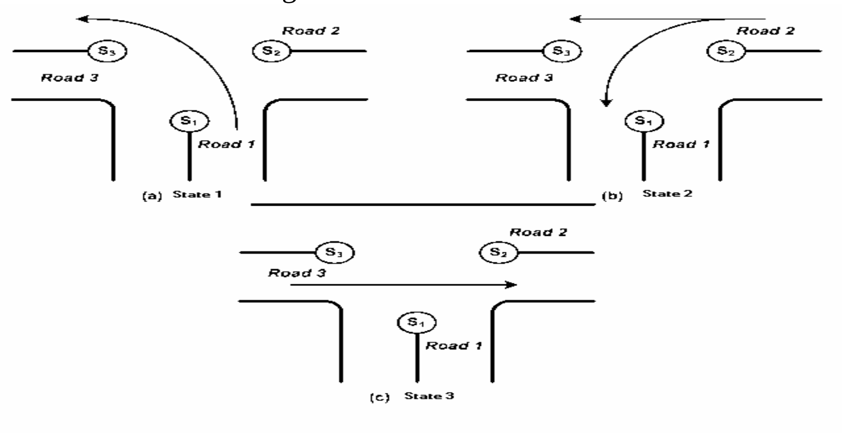


Figure 2: The Three States

The operation of the three light signals (S1, S2, and S3) is controlled through an arrangement of *traffic sensors* and *traffic light controller* circuit as shown in Figure 3**.** There are three traffic sensors *x1, x2, and x3,* which sense the presence of traffic on the three roads as illustrated in Table 1. The controller operation is determined by the output of these three sensors as enumerated in Table 2.

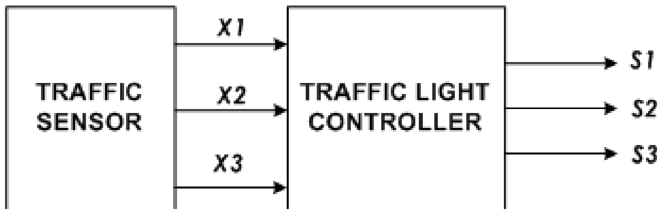


Figure 3: Traffic Sensor and Traffic Light Controller circuit

x_3	x_2	x_1	Indication
0	0	0	No traffic for all roads
0	0	1	Traffic for $Road_1$only
0	1	0	Traffic for $Road_2$only
0	1	1	No traffic for $Road_3$
1	0	0	Traffic for $Road_3$only
1	0	1	No traffic for $Road_2$
1	1	0	No traffic for $Road_1$
1	1	1	Traffic present on all roads

Table 1: Traffic Sensor Signals

x_3	x_2	x_1	Indication
0	0	0	Stay at STATE 0
0	0	1	Stay at STATE 1
0	1	0	Stay at STATE 2
0	1	1	Alternate between STATE 1 and STATE 2
1	0	0	Stay at STATE 3
1	0	1	Alternate between STATE 1 and STATE 3
1	1	0	Alternate between STATE 2 and STATE 3
1	1	1	Normal operation: STATE 1, STATE 2, STATE 3, STATE 1…

Table 2: Traffic Sensors and Controller operation

The design of the traffic controller module requires using D-Flip-Flops with asynchronous clear (FDC). Assume that the controller has three inputs: *x3*, *x2*, and *x1* coming from the traffic sensor, and three outputs: *S1*, *S2*, and *S3*, which control the operation of the three traffic light signals (logic 1 represents a GREEN signal and logic 0 represents a RED signal).

Pre-Lab

1. Read the section on switch debouncing from the lab guide.
2. Find a state diagram for this circuit.
3. Obtain a state table for the circuit (use the table below); whenever you have the choice to select between two states follow the order STATE1, STATE2, STATE3, and then STATE1 and so on.
4. Derive the Flip-Flop input equations from the state table.
5. Derive output equations for the traffic signals.

No	X_3	X_2	X_1	A	B	D_A	D_B	S_3	S_2	S_1
0	0	0	0	0	0					
1	0	0	0	0	1					
2	0	0	0	1	0					
3	0	0	0	1	1					
4	0	0	1	0	0					
5	0	0	1	0	1					
6	0	0	1	1	0					
7	0	0	1	1	1					
8	0	1	0	0	0					
9	0	1	0	0	1					
10	0	1	0	1	0					
11	0	1	0	1	1					
12	0	1	1	0	0					
13	0	1	1	0	1					
14	0	1	1	1	0					
15	0	1	1	1	1					
16	1	0	0	0	0					
17	1	0	0	0	1					
18	1	0	0	1	0					
19	1	0	0	1	1					
20	1	0	1	0	0					
21	1	0	1	0	1					
22	1	0	1	1	0					
23	1	0	1	1	1					
24	1	1	0	0	0					
25	1	1	0	0	1					
26	1	1	0	1	0					
27	1	1	0	1	1					
28	1	1	1	0	0					
29	1	1	1	0	1					
30	1	1	1	1	0					
31	1	1	1	1	1					

Show it to your instructor for approval prior to beginning to construct the circuit in the lab

In-Lab

1. Follow the tutorial given in the Lab Guide for designing circuits using schematic editor
2. Use D-Flip-Flops with Asynchronous Clear from the library.
3. If your design is too large to fit on one schematic, transfer the combinational logic to another schematic as a macro. Refer to the section on Creating Macros in the lab guide.
4. Constrain the inputs of your circuit (*x3*, *x2*, and *x1*) to 3 of the level switches (SW0, SW1, and SW2) respectively.
5. Constrain Flip-Flop outputs to LEDs (LD6 and LD7) where LD7 is the output of the least significant Flip-Flop. These will show the current state.
6. Constrain Flip-Flop inputs to LEDs (LD4 and LD5) where LD5 is the input of the least significant Flip-Flop. These will show the next state.

7. Constrain the three signal outputs (*S3*, *S2*, and *S1*) to three LEDs (LD0, LD1, and LD2) respectively.
8. Use two Buttons, one to provide a clock signal (BTN0) -Beware of switch debouncing problems, and the other as an asynchronous clear (BTN1). Connect these inputs to the respective inputs of the Flip-Flops.
9. Implement your design.
10. Perform the timing (post-place-and-route) simulation.
11. Download your design.
12. Verify its functionality by applying different input combinations and compare it with your state table.
13. The following three steps must be shown to the instructor during the lab:
    a. Complete Schematic
    b. Timing simulation of the circuit
    c. Verification on the board
14. Print a copy of the Grading sheet and get it signed by the instructor before you leave the lab.
15. Complete the lab report and submit it before the next lab

Post-Lab Questions

1. The circuit which you have designed is a Moore machine or a Mealy machine? **Why?**

Grading Sheet

There are 100 points awarded for this laboratory as follows:

Points Possible	Points Awarded	Topics Addressed
15	/15	Pre-Lab State Table(4) Logic Minimization (5) Boolean Equations (5) Organization, Clarity (1)
60	/60	Lab Work Complete Schematic (10) Timing Simulation (10) Bit-file Generation (25) Verifying on the board (15)
25	/25	Lab Report Contents (10) Appendices (10) Spelling, Grammar, and Style (5)
100	/100	Total

5. Verilog Based Design of Ripple Carry Adder

Objectives

- Familiarize with HDL based design entry, using Verilog.
- Learn to use Structural design method in Verilog, and understand its basics, by:
    - Constructing an 4-bit Ripple Carry Adder using the *'HalfAdd'* module given in the Lab Guide Chapter on Verilog, Section 3.2, and synthesizing it.
    - Modifying the Design by implementing the Boolean equations for Sum and Carry for a Full Adder, then re-synthesizing the 8-bit RCA, using the new Full Adder circuit.
- Familiarize with the basics of Top-down design methodology.

Overview

In this lab, you are going to be constructing a simple 8 bit Ripple Carry Adder using Verilog HDL design entry. The Diagram for which is given below:

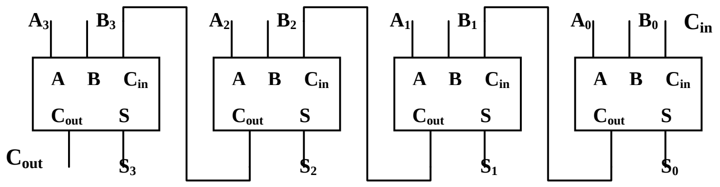


Figure 1: Block Diagram of a 4-bit Ripple Carry Adder

Pre-Lab

1. Read Section 3.2 on Structural Verilog Design entry from the Lab Guide.
2. Practice how to implement Boolean equations in Verilog, using the declared wires and the *'assign'* statement.
3. Using any simple Text editor, create the file *'HalfAdd.v'*. The contents of this file will be the code for the Verilog module *HalfAdd* as given in the lab guide Figure 3.6. Bring this file to the Lab.

In-Lab Part 1:

1. Create a New Project in your Xilinx Project manager, Titled Experiment #.
2. Select Verilog HDL as the Type for the Top Level Module Type field.
3. Using a simple text editor, create a file named *'RCA_4bit.v'* as the top level file in the project.
4. Add the *'HalfAdd.v'* Verilog file to your project.
5. In the *RCA_4bit.v* file, declare the top level module of the 4-bit RCA, listing and declaring properly all the input and output wires. There should be 9 input wires (4 inputs + 1 Carry-in) and 5 output wires (4 sums and 1 Carry-out). This module will implement the functionality described by Figure 1.
6. After the Signal Declarations are out of the way, make four dummy instantiations of a module named *'FullAdd'*, with the instance names being Adder1 to Adder4. This module, although not yet defined, will be used to implement a single Full adder; thus it will have 3 inputs and 2 outputs. This module will implement the functionality of

any one of the 4 blocks shown in Figure 1. (For help with this, refer to Lab Guide, Verilog Chapter, Section 3.2)

7. Make sure to connect the input and output wires of all the *FullAdd* module instantiations properly, as shown in Figure 1.
8. Once the *'RCA_4bit.v'* module is completely implemented, close this and open a new file *'FullAdd.v'* that will contain the HDL code for the as yet unimplemented *FullAdd* module.
9. Using the *HalfAdd* module given in the Lab Guide, implement the *FullAdd* module by instantiating 2 *HalfAdd* modules, and connecting them as shown in Figure 2. Use the *assign* statement to generate the C_{out} signal by performing a bitwise OR of the two half-adder carry signals. At this point the design is complete.

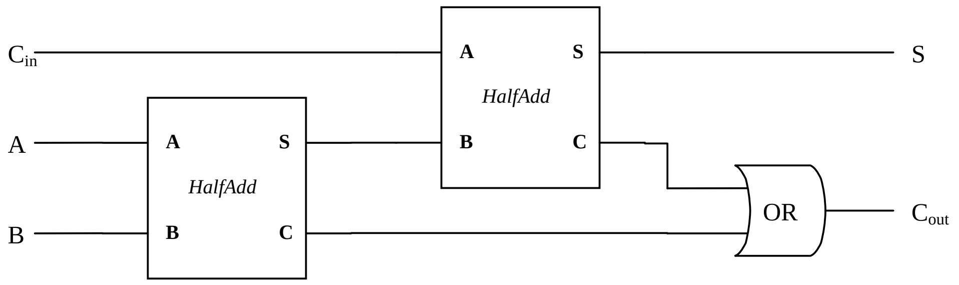


Figure 2: A Full Adder Circuit using 2 Half Adders

10. Implement your design.
11. Perform the timing (post-place-and-route) simulation.
12. Download your design.
13. Verify its functionality by applying different input combinations, and checking the result.
14. The following three steps must be shown to the instructor during the lab
    a. Complete Verilog Structural Code.
    b. Timing simulation of the circuit
    c. Verification on the board, using the given truth table.

In-lab Part 2:

15. Now open the *'FullAdd.v'* file.
16. Remove the instantiations of the *'HalfAdd'* module in the *'FullAdd.v'* file,
17. Using the Boolean equations for a Full Adder, and *assign* statements, generate the output Sum and Carry signals for the *FullAdd* module (for help, refer to Lab Guide, Verilog Chapter, Section 3.2). The logic diagram for a Full Adder is given in Figure 3 for reference. Save and Close the file.
18. Remove the *'HalfAdd.v'* file from the project.
19. Implement your design.
20. Perform the timing (post-place-and-route) simulation.
21. Download your design.
22. Verify its functionality by applying different input combinations, and checking the result.
23. The following three steps must be shown to the instructor during the lab
    a. Complete Verilog Structural Code.
    b. Timing simulation of the circuit.

c. Verification on the board.

24. Complete the lab report and submit it before the next lab.

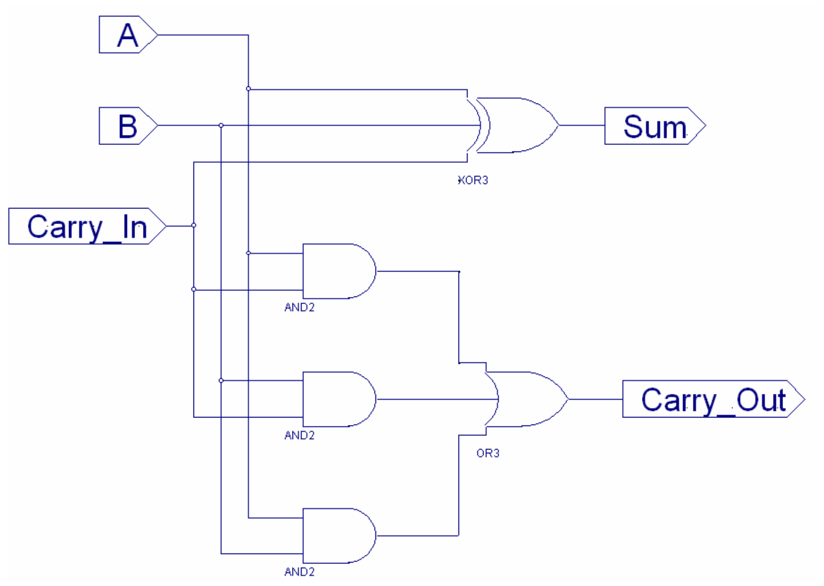


Figure 3: Logic Diagram of a Half Adder

Post Lab Questions

1. Explain what you understand from Top-down design methodology. Describe how it was used in this experiment.
2. Why was it possible to completely change the *HalfAdd* module, and still get a functioning circuit without changing anything else?

Grading Scheme

Points Possible	**Points Awarded**	**Topics Addressed**
55	/55	In-lab Part 1 *HalfAdd.v* (5) *FullAdd.v*(20) *RCA_4bit.v* (30)
20	/20	In-Lab Part 2 Boolean equations of Full Adder (5) New version of *FullAdd.v* (15)
25	/25	Lab Report Contents (10) Post-Lab Questions (10) Spelling, Grammar and Style (5)
100	/100	Total (100)

6. RTL Verilog Based Design of an Arithmetic Logic Unit

Objectives

- Familiarize with the RTL level design methodology in Verilog, by:
  - Under taking procedural design of an Arithmetic Logic Unit using *'case'* statements.
  - Using select line(s) to multiplex between two input busses at one of the inputs of both units, using an *'if'* statement.
- Attain an introductory understanding of how to combine different design methodologies, such as structural and RTL Verilog.

Overview

In this lab, you are going to be constructing a basic 4-bit Arithmetic-Logic Unit (ALU) using Verilog HDL design entry. The truth table for this ALU is given in Figure 1. Y, A, and B are all unsigned 4-bit values.

S_3	**S_2**	**S_1**	**S_0**	**Output**	**Comments**
				Arithmetic Operations	
0	0	0	0	Y <= A	Transfer A
0	0	0	1	Y <= A + 1	Increment A
0	0	1	0	Y <= A + B	Add A and B
0	0	1	1	Y <= A + B + 1	Add with Carry
0	1	0	0	Y <= A + ~B	A plus 1's complement of B
0	1	0	1	Y <= A + ~B + 1	A – B
0	1	1	0	Y <= A – 1	Decrement A
0	1	1	1	Y <= B	Transfer B
				Logic and Shift Operations	
1	0	0	0	Y <= A & B	Bitwise AND of A and B
1	0	0	1	Y <= A | B	Bitwise OR of A and B
1	0	1	0	Y <= A ^ B	Bitwise XOR of A and B
1	0	1	1	Y <= ~A	Bitwise NOT of A (1's complement)
1	1	0	0	Y <= A ~& B	Bitwise NAND of A and B
1	1	0	1	Y <= A ~| B	Bitwise NOR of A and B
1	1	1	0	Y <= A << 1	Left Shift A once
1	1	1	1	Y <= A >> 1	Right Shift A once

Figure 1: Truth Table for a 4-bit Arithmetic-Logic-Shift Unit

The diagram for the circuit to be implemented is given below in Figure 2. The design has three 4-bit input busses and one 4-bit output bus. Two of the input busses pass through a 4-bit 2-1 multiplexer, before being connected to the B input bus of the ALU. One of these busses will be selected based on the value of a fifth select bit – S_4. The ALU module will be implemented by expanding upon the code provided in the Lab guide Verilog chapter, Figure 3.13b.

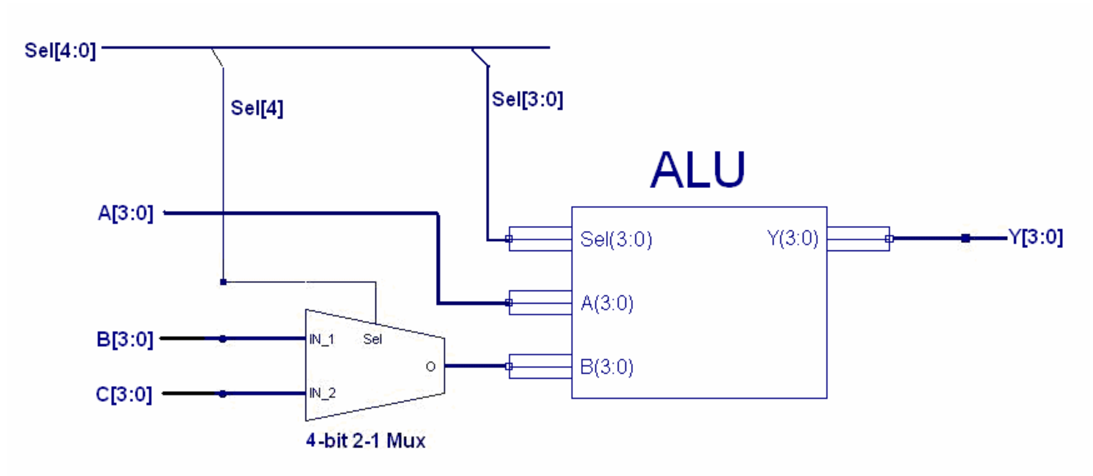


Figure 2: Block Diagram of a 4-bit Arithmetic-Logic Unit with a multiplexed input.

Pre-Lab

1. Read Section 3.3 on RTL level Verilog Design entry from the Lab Guide.
2. Practice how to implement Boolean equations in Verilog, but instead of using continuous assignment (*assign*) statements, use *'always'* blocks and procedural assignment statements.
3. Using any simple Text editor, create the file *'ALU.v'*. The contents of this file will be the code for the Verilog module *Logic*, as given in the lab guide Figure 3.13b. Bring this file to the Lab.

In-Lab Part 1:

1. Create a New Project in your Xilinx Project manager, Titled Experiment 6.
2. Select Verilog HDL as the Type for the Top Level Module Type field.
3. Using a simple text editor, create a file named *'ALU_4bit.v'* as the top level file in the project.
4. Add the *'ALU.v'* Verilog file to your project.
5. Since the code given in Figure 3.13b of the lab guide does not include Arithmetic and Shift functionality, modify the code by expanding the case variable 'Sel' to 4-bits instead of 3-bits, and add more *'case'* statements to incorporate the un-implemented rows of the truth-table above. The 'ALU' module should now have two 4-bit input ports, one 4-bit Select input port, and one 4-bit output port.
6. Refer to Section 3.1 of the lab guide Verilog chapter for help with the shift and arithmetic operators.
7. In the *ALU_4bit.v* file, declare the top level module of the 4-bit ALU: *ALU_4bit*, listing and declaring properly all the input and output ports. There should be three 4-bit wide input ports of type *'wire'* (for A, B and C), one 4-bit wide output port of type *'reg'*, (for Y), and one 5-bit input port (for the select lines S_4 through S_0). This module will implement the functionality described by Figure 2.
8. Declare an additional *reg* type 4-bit variable in the *ALU_4bit* module: *Secondary_Input*. This will serve as an intermediate bus that connects the output of the 4-bit 2-1 multiplexer to the B-input of the ALU module that will produce the final 4-bit output Y.
9. Instantiate the 'ALU' module in the *ALU_4bit* module. The 4-bit output port of the *ALU* module should be connected to the *Y* output bus, and the B input port should be connected to the *Secondary Input* bus.

10. Using an '*if*' statement and procedural assignments (all within an *always* block, of course), assign the *B* and *C* inputs of the 'ALU_4bit' module to the 4-bit *Secondary Input* bus in the top-level block, based on the value of the S_4 select line.
11. Implement your design.
12. Perform the timing (post-place-and-route) simulation.
13. Download your design.
    a. Set the inputs B and C to fixed 4-bit values by creating a new top level module that instantiates your design, as given below.

```
module lab6top(A,Sel,Y);
      input [3:0] A;
      input [4:0] Sel;
      output [3:0] Y;
      wire [3:0] G,H;

      assign G = 4'b 0010;
      assign H = 4'b 1001;

      ALU_4BIT alu1(Sel,A,G,H,Y);
endmodule
```

    b. Constrain the input 4-bit A input to the first 4 switches.
    c. Constrain the select inputs S_3 through S_0 to the remaining switches, and S_4 to the first push button.
14. Verify its functionality by applying different input combinations, and checking the result.
15. The following three steps must be shown to the instructor during the lab
    a. Complete Verilog RTL Code for both modules.
    b. Timing simulation of the circuit.
    c. Verification on the board, using the given truth table.

Post Lab Questions

1. Explain what you understand from RTL design methodology.
2. Describe in detail how the *always* statement is used in Verilog based Design.

Grading Scheme

Points Possible	**Points Awarded**	**Topics Addressed**
5	/5	Pre-Lab *ALU.v* (5)
70	/70	In-Lab *ALU.v* modification to include shift functionality (15) *ALU.v* modification to include arithmetic functionality (20) *ALU_4bit.v* implementation (35)
25	/25	Lab Report Contents (10) Post-Lab Questions (10) Spelling, Grammar and Style (5)
100	/100	Total (100)

7. Finite State Machine Design in Verilog

Objectives

- Familiarize with the design and implementation of Finite State Machines using Verilog.
- Acquiring an understanding of best practices for safe implementation of State Machines, including the use of Resets.

Overview

In this lab, you are going to be designing the Finite State Machine for the Traffic Light Controller that was developed in Experiment 4, this time using Verilog HDL instead of Schematic Design Entry.

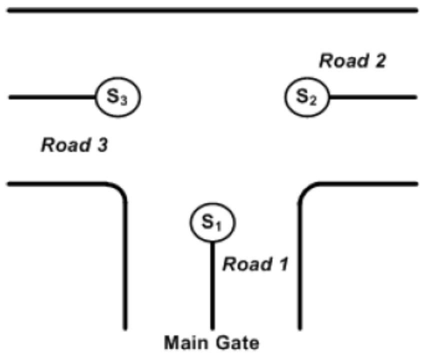


Figure 1: The Three Traffic Light Signals

Design Specifications

There are three traffic light signals (*S1*, *S2*, and *S3*), each alternating between two states, *RED* and *GREEN*. These signals control the traffic flow on the three roads, *road1*, *road2*, *road3* in four possible states as follows.

- In *STATE 0:* no traffic so give priority to the main gate's road. (*S1* = GREEN, *S2* = RED, *S3* = RED)
- In *STATE 1*, traffic coming through *road1*. (*S1* = GREEN, *S2* = RED, *S3* = RED)
- In *STATE 2*, traffic coming through *road2*. (*S1* = RED, *S2* = GREEN, *S3* = RED)
- In *STATE 3,* traffic coming through *road3*. (*S1* = RED, *S2* = RED, *S3* = GREEN)

The operation of the three light signals (S1, S2, and S3) is controlled through an arrangement of *traffic sensors* and *traffic light controller* circuit as shown in Figure 2**.** There are three traffic sensors *x1, x2, and x3,* which sense the presence of traffic on the three roads as illustrated in Table 1. The controller operation is determined by the output of these three sensors as enumerated in Table 2.

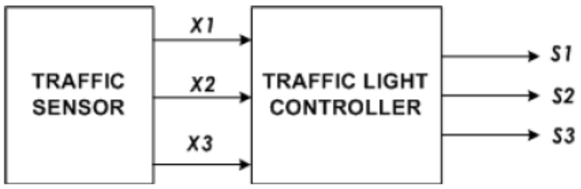


Figure 2: Traffic Sensor and Traffic Light Controller Circuit

x_3	x_2	x_1	Indication
0	0	0	No traffic for all roads
0	0	1	Traffic for Road$_1$ only
0	1	0	Traffic for Road$_2$ only
0	1	1	No traffic for Road$_3$
1	0	0	Traffic for Road$_3$ only
1	0	1	No traffic for Road$_2$
1	1	0	No traffic for Road$_1$
1	1	1	No traffic for all roads

Table 1: Traffic Sensor Signals

x_3	x_2	x_1	Indication
0	0	0	Stay at STATE 0
0	0	1	Stay at STATE 1
0	1	0	Stay at STATE 2
0	1	1	Alternate between STATE 1 and STATE 2
1	0	0	Stay at STATE 3
1	0	1	Alternate between STATE 1 and STATE 3
1	1	0	Alternate between STATE 2 and STATE 3
1	1	1	Normal operation: STATE 1, STATE 2, STATE 3, STATE 1…

Table 2: Traffic Sensor and Controller Operation

Based on the above specifications, the state diagram of this Finite State Machine has been provided in Figure 3 below. Your objective in this experiment is to implement the functionality described by this state diagram using Verilog HDL.

Pre-Lab

1. Read Section 3.5 on Modeling Finite State Machines using Verilog from the Lab Guide.
2. Verify the functionality of the State Diagram Provided Above.

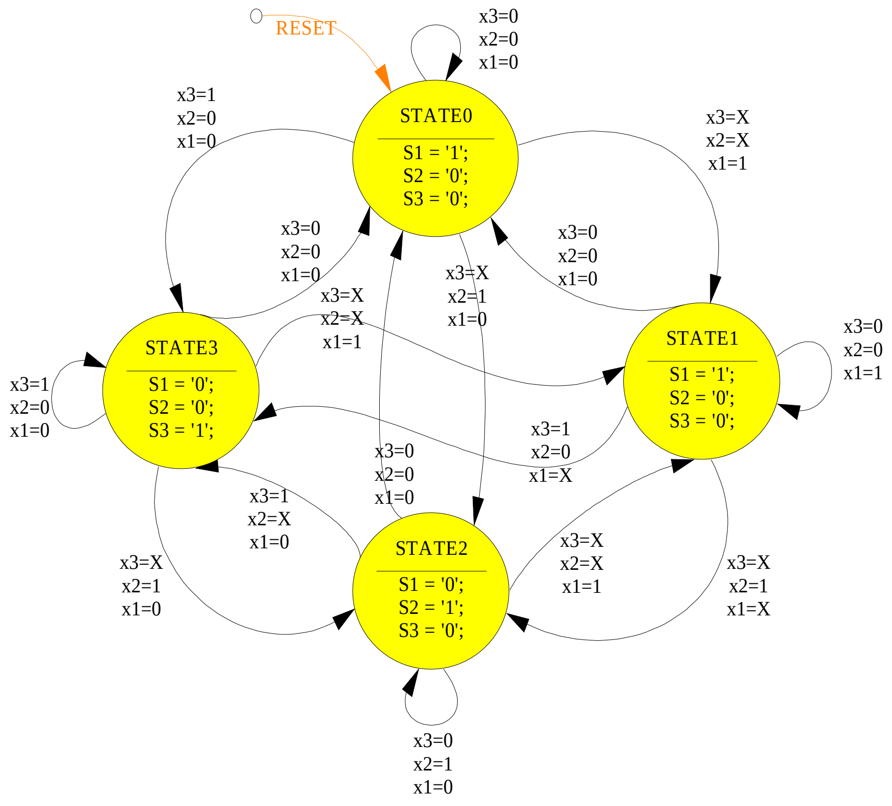


Figure 3: State Diagram for Traffic Light FSM
(X represents Don't Care conditions)

In-Lab:

1. Create a New Project in your Xilinx Project manager, Titled Experiment #.
2. Select Verilog HDL as the Type for the Top Level Module Type field.
3. Using a simple text editor, create a file named '*Traffic_Controller.v*' as the top level file in the project.
4. In the *Traffic_Controller.v* file, implement a module named *Traffic_Controller* which will implement the functionality specified by the state diagram of Figure 3.
5. Declare the *Traffic_Controller* module, with all its inputs and outputs – it should have:
    a. Three 1-bit inputs from the Traffic Sensors: *x3*, *x2*, and *x1*.
    b. Three 1-bit outputs, the signals *S1*, *S2*, and *S3* (a '1' output should represent a GREEN signal, while a '0' output should represent a RED signal).
    c. One 2-bit output identifying the Current State of the FSM,
    d. One 2-bit output indicating the Next State of the FSM.
6. The Verilog code for this module should contain 3 *always* blocks:
    a. One for the Next State Logic, which generates the Next state value based on the Current State and the Traffic Sensor inputs
    b. Another for the Output Logic, which generates the signals *S1*, *S2*, and *S3* based on the Current State only (see Figure 3 above).
    c. The Third for the Current State Registers, which implements the Current State Registers, as well as the Asynchronous Reset.

For help with the concepts, refer to Section 3.5 of the Verilog Chapter in the Lab guide, in particular to the example given in Figure 3.14.

7. Constrain the inputs of your circuit (*x3*, *x2*, and *x1*) to 3 of the level switches (SW0, SW1, and SW2) respectively.
8. Constrain the 2-bit Current state output to LEDs LD6 and LD7 where LD7 is the output of the LSB. These will show the current state.
9. Constrain the 2-bit Next-state output to LEDs LD4 and LD5 where LD5 is the input of the LSB. These will show the next state.
10. Constrain the three signal outputs *S3*, *S2*, and *S1* to three LEDs LD0, LD1, and LD2 respectively.
11. Use two Buttons, one to provide a clock signal (BTN0) and the other for the asynchronous reset signal (BTN1). - Beware of switch debouncing problems.
12. Implement your design.
13. Perform the timing (post-place-and-route) simulation.
14. Download your design.
15. Verify its functionality by applying different input combinations, and checking the result.
16. The following three steps must be shown to the instructor during the lab
    a. Complete Verilog Code.
    b. Timing simulation of the circuit
    c. Verification on the board, using the given state diagram.
17. Complete the lab report and submit it before the next lab.

Post Lab Questions

1. What is the difference between a Mealy machine and a Moore machine? The State machine designed in this lab is of which kind?
2. Attempt to merge the two combinatorial *always* blocks – the one for the output logic and the next state logic. Test and verify your new code. Present your code to the instructor as part of the post lab report.

Grading Scheme

Points Possible	**Points Awarded**	**Topics Addressed**
80	/80	In-lab *Traffic_Controller* module declaration (10) *always* block for next state logic (40) *always* block for output logic (15) *always* block for Current State and asynchronous reset (15)
20	/20	Lab Report Contents (5) Post-Lab Questions (15)
100	/100	Total (100)

8. Designing a Programmable Digital Lock

Objectives

- Learning how to partition a system into data-path and control unit.
- Integrating Schematics and Verilog code together

Overview

In this lab you will design a Digital Lock. We will begin by partitioning the system into two blocks: a data-path unit and a control unit. The data-path contains blocks that only deal with data. They do not provide control to any other blocks and they themselves need to be controlled by the Control Unit. These blocks can be viewed as workers that can perform certain tasks (on the data) but need to be managed by someone. The manager in this case is the Control Unit that tells every worker what to do. Examples of data-path blocks include registers, counters, comparators, multiplexers, adders etc. Each of these blocks have control signals like load, count, enable, clear etc. which have to be controlled through the control unit. The general structure of a digital system is shown in Figure 1.

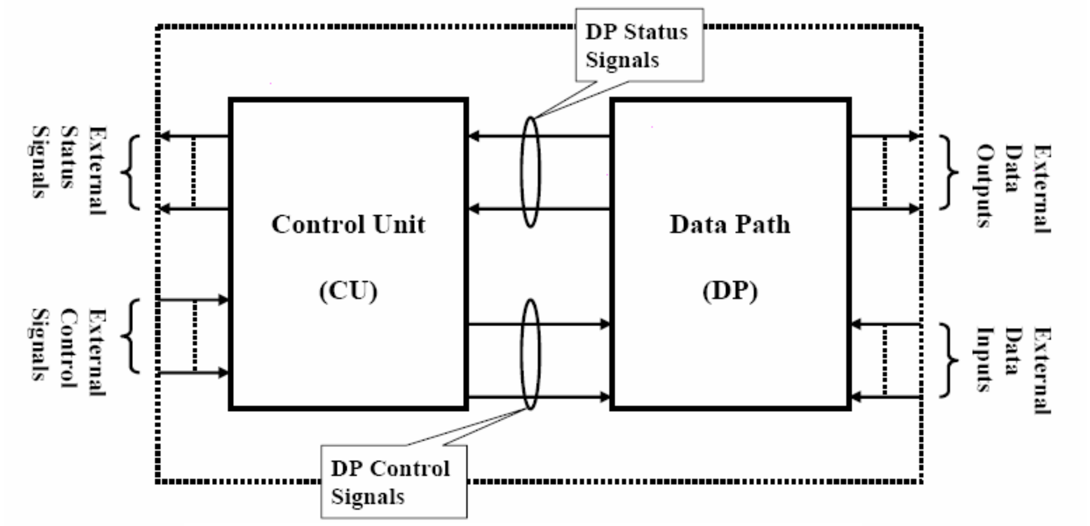


Figure 1: General Structure of a Digital System

External Control Signals specify the task required from the whole circuit (e.g. calculate the average of some integers).
External Status Signals indicate the status of the whole circuit (finished processing, overflow, error etc.)
External Data Inputs/Outputs contain the data going into or coming out of the circuit (for e.g. integers o be averaged and the result of the average)
DP Control Signals are generated by the Control Unit to control different blocks in the data-path (for e.g. enable inputs in counters, shift inputs in registers)
DP Status Signals indicate the status of some blocks in the data-path (e.g. when a counter reaches its maximum value or adder generates an overflow)

Design Specifications

You are asked to design a programmable digital lock circuit. The circuit receives serial input combinations. If the input combination matches a 4-BCD digit stored pattern, the lock is opened. The user should be able to store a new pattern that is, he should be able to re-program the lock. If 3 wrong 4-BCD digit combinations are entered, the lock jams and needs to be reset. The Lock has 3 inputs; Program, Open and 4-bit serial input. It has two outputs Red and Green. If the lock is opened Green is ON. If it is closed Red is ON. If it is locked both Red and Green are ON.

Design Steps:

The block diagram is shown in Figure 2. The 4-bit SERIAL input provided by the user is given to the Data-path unit. The data-path generates three status signals, MATCH, LOCK and COUNTDONE. The inputs PROG and OPEN are given to the Control Unit. Based on these inputs, it generates six control signals for the data-path. These are SHIFT-A, SHIFT-B, ENABLE3, CLR3, ENABLE2 and CLR2. The primary outputs, RED and GREEN are also generated by the Control Unit.

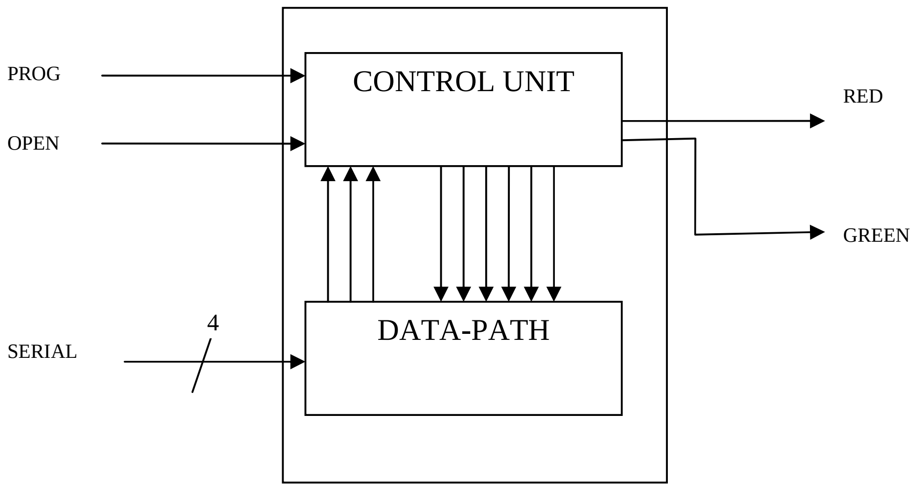


Figure 2: A Block Diagram of the Circuit

The data path consists of several registers, a counter and a magnitude comparator. The required components include:

A-Register: A set of four 4-bit registers which hold the serial input given by the user

B-Register: Another set of four 4-bit registers which hold the password. A schematic for the shift registers is shown in Figure 3.

Comparator: This can be designed using XOR gates or the standard magnitude comparator given in the library. Its function is to compare the magnitudes of A-Register and B-Register. The XOR tree should be constructed as follows: The contents of each 4-bit register in Register-A should be compared with the contents of the corresponding 4-bit register in Register-B by using XOR gates. The outputs of the four XOR gates should be connected by a fifth XOR gate and the resultant output should be named as MATCH.

Counter3: A 3-bit counter used to count the number of BCD digits entered. It generates an output COUNTDONE which is equal to 1, when the count reaches 4.

Counter2: A 2-bit counter used to count the number of times a wrong combination is entered. It generates an output LOCK, which becomes equal to 1 when three wrong combinations are entered.

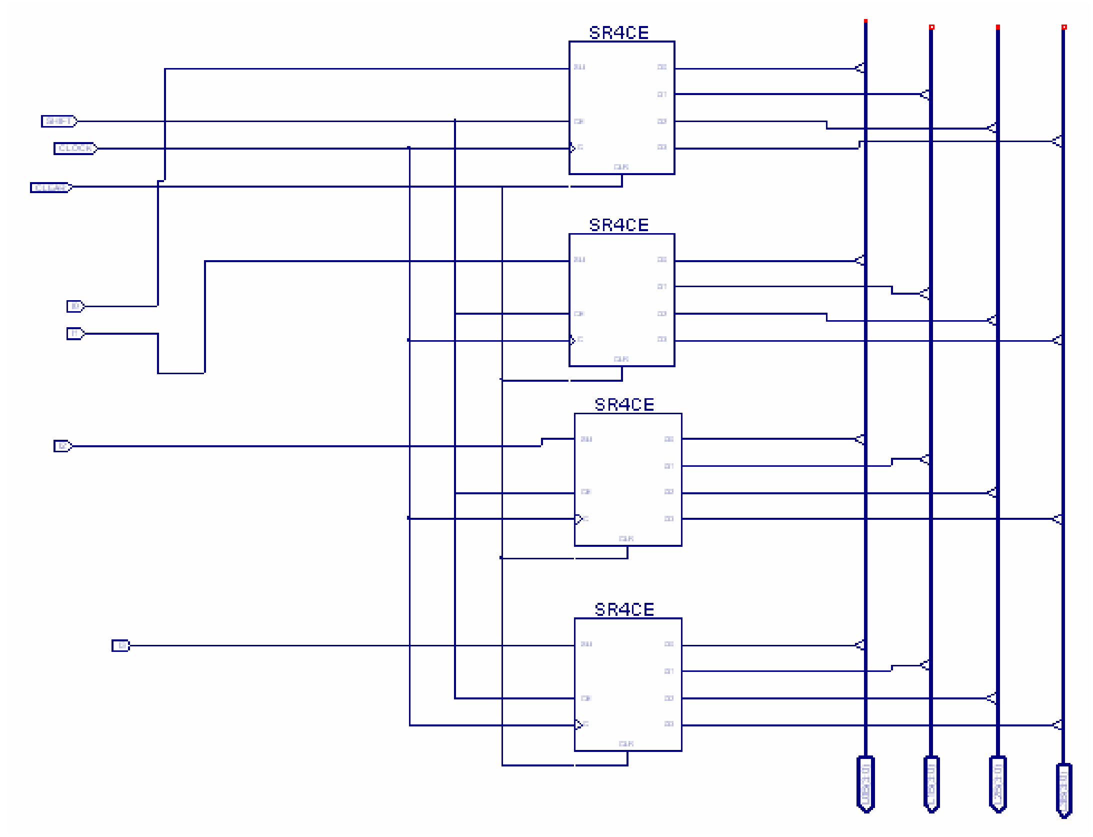


Figure 3: The Shift Registers

A block diagram of the data-path is shown in Figure 4.

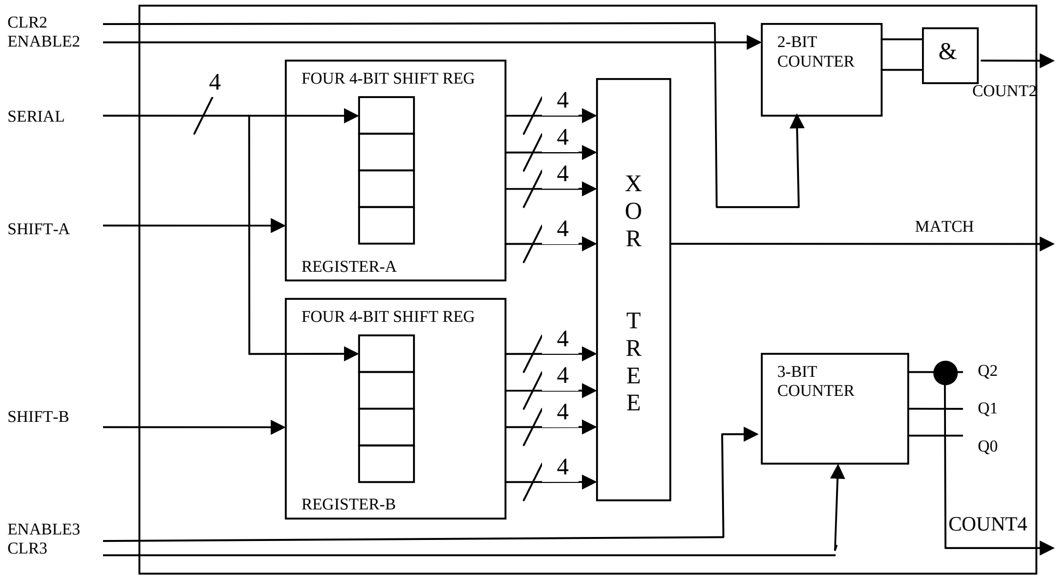


Figure 4: A Block Diagram of the Data-path

Controllers in general need status information to decide on the next proper actions. In this circuit, the controller uses the signal *COUNT4* to determine when to do the magnitude comparison. A Block diagram of the Control Unit is given in Figure 5.

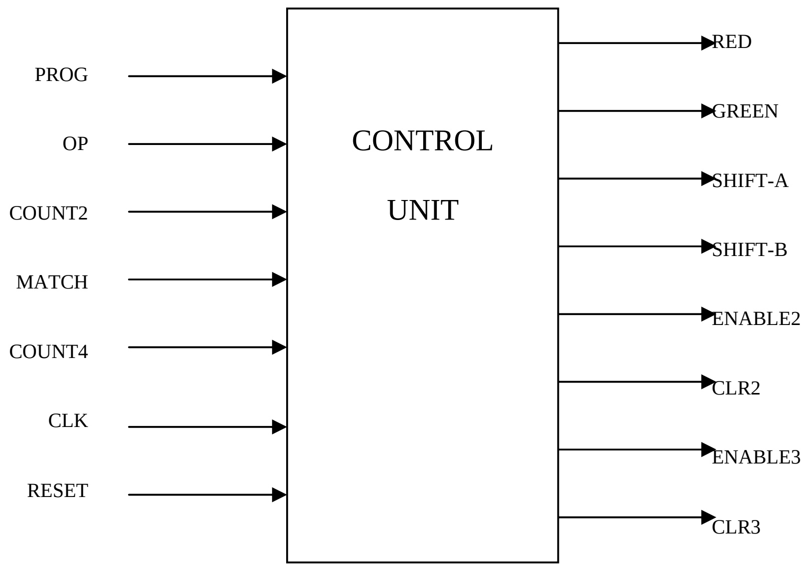


Figure 5: A Block Diagram of the Control Unit

The state diagram of the controller is shown in Figure 6. The controller FSM has 5 states (STATE0, STATE1, STATE2, STATE3, STATE4 and STATE5). The *RESET* signal resets the system to the first state S0 and clears all the components.
If the *Prog* input is 1, the FSM moves to the second state S1, and the B-Register is loaded with the Serial input provided by the user.
If the *Open* input is 1, the FSM moves to the third state S2, and the A-Register is loaded with the Serial input provided by the user. Counter3 is incremented by 1.

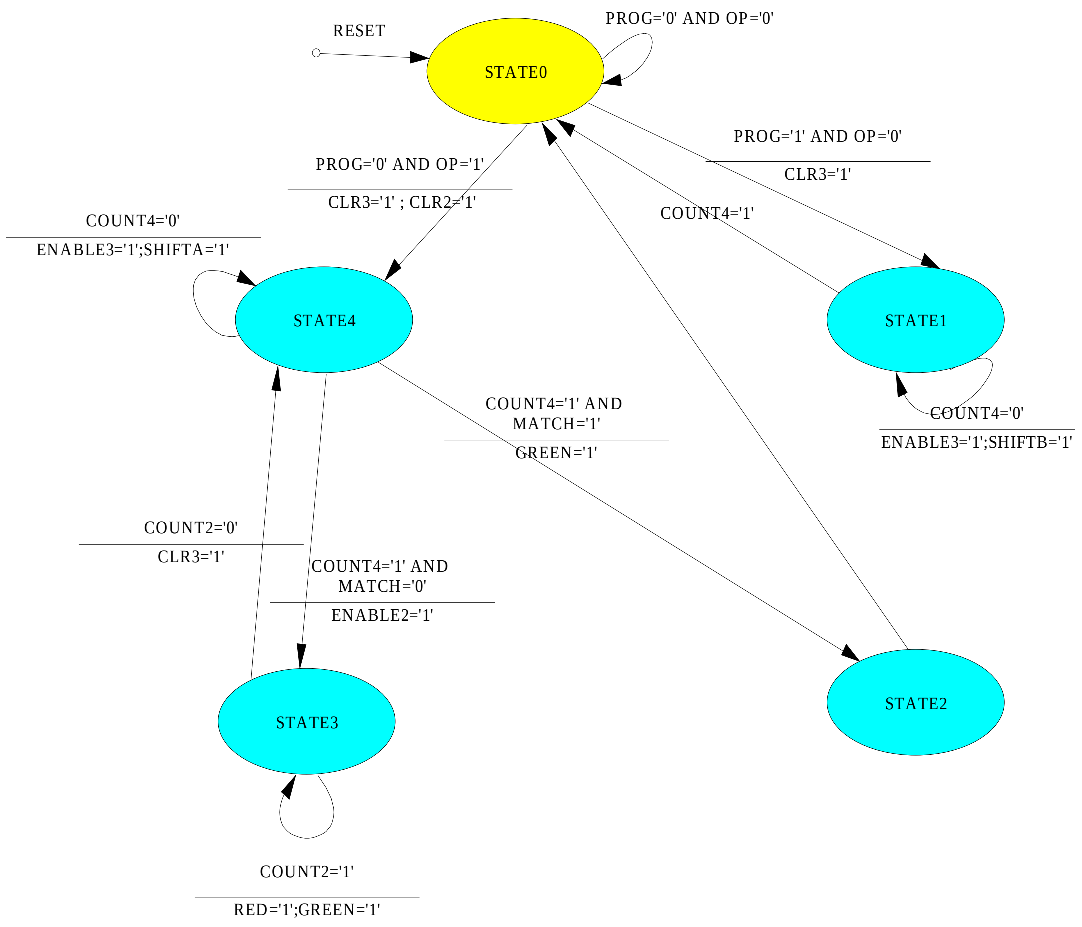


Figure 6: The State Diagram for Control Unit

Pre-Lab

1. Identify the following signals:
2. External Control Signals
3. External Status Signals
4. Data Input/Output Signals
5. Data-path Control Signals
6. Data-path Status Signals
7. Use the State Diagram Editor to create a CONTROL.DIA file. Generate a Verilog file using STATECAD.
8. Design the data-path using either the schematic editor or Verilog.

Show the design to your instructor for approval prior to beginning to construct the circuit in the lab

In-Lab

1. Follow the tutorial given in the Lab Guide for using the State Diagram Editor
2. Create a Schematic Symbol of the Control Unit from the Project Manager.
3. Import it as a symbol in your top-level schematic.

4. Perform functional simulation of the Control Unit.
5. Import your data-path design in the top-level schematic as well.
6. Constrain the Prog and Op signals to SW0 and SW1.
7. The outputs RED and GREEN should be constrained to LD0 and LD1.
8. The clock should be constrained to a push button.
9. Constrain the Reset signal to a push button as well.
10. Implement your design.
11. Perform the timing (post-place-and-route) simulation.
12. Download your design.
13. Verify its functionality by applying different input combinations and compare it with your state table.
14. The following three steps must be shown to the instructor during the lab
15. Complete the design
16. Timing simulation of the circuit
17. Verification on the board
18. Print a copy of the Grading sheet and get it signed by the instructor before you leave the lab.
19. Complete the lab report and submit it before the next lab

Post-Lab Questions

1. Instead of using the state diagram editor, design your control unit using Verilog. Refer to the Verilog Cookbook. Do you need to change the top-level module if you re-design the control unit using Verilog?

Grading Sheet

There are 100 points awarded for this laboratory as follows:

Points Possible	Points Awarded	Topics Addressed
15	/15	Pre-Lab Identifying the signals(5) State Diagram (5) Data-path design (5)
60	/60	Lab Work Integrating the design (20) Timing Simulation (10) Bit-file Generation (15) Verifying on the board (15)
25	/25	Lab Report Contents (10) Appendices (10) Spelling, Grammar, and Style (5)
100	/100	Total