Texas A&M University
Department of Electrical Engineering

ELEN 248

Introduction to Digital Design
Laboratory Manual
# Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laboratory Policies and Report Format</td>
<td></td>
</tr>
<tr>
<td>Tips and Tricks of Bread-Boarding</td>
<td></td>
</tr>
<tr>
<td>Lab 1: Introduction to Combinational Design</td>
<td></td>
</tr>
<tr>
<td>Lab 2: Logic Minimization using K-Maps</td>
<td></td>
</tr>
<tr>
<td>Lab 3: Study and Implementation of Adders</td>
<td></td>
</tr>
<tr>
<td>Lab 4: Implementation of a 4 bit ALU</td>
<td></td>
</tr>
<tr>
<td>Lab 5: Verilog and Xilinx ISE</td>
<td></td>
</tr>
<tr>
<td>Lab 6: De-Bouncing, Counters &amp; LCD Display Control</td>
<td></td>
</tr>
<tr>
<td>Lab 7: Carry Look Ahead and Carry Save Adders</td>
<td></td>
</tr>
<tr>
<td>Lab 8: 16-bit ALU with eight operations</td>
<td></td>
</tr>
<tr>
<td>Lab 9: Design of a Traffic Light Controller</td>
<td></td>
</tr>
</tbody>
</table>
Laboratory Policies and Report Format

Each lab assignment has two deliverables— the pre-lab report and the post-lab report. The pre-lab reports are due at the beginning of the lab period and the post-lab reports are due after the completion of the lab. The pre-lab reports are intended to be a complete documentation of the work done in preparation for the lab. The post-lab reports are intended to be a complete documentation of the work done in the lab. These reports should be complete so that someone else familiar with digital design could use it to verify your work. Portions of the pre-lab needed during lab should be duplicated before you arrive since the TA will collect the original pre-lab at the beginning of each lab. The post-lab reports are due after the lab is completed. Check with your TA for the exact deadline to turn in your post-lab reports. The latest deadline the TA can allow for the post-lab submission is before the start of the next lab. The pre-lab and post-lab report format is as follows:

1. A neat thorough pre-lab must be presented to your lab instructor at (or before) the start of your scheduled lab period. Lab reports should be submitted on 8.5” x 11” paper, typed on one side only. Your report is a professional presentation of your work in the lab. Neatness, organization and completeness will be rewarded. Points will be deducted for any part that is not clear.

2. Each report (pre-lab & post-lab) will contain the following sections:
   a) **Cover Page:** Name, ELEN 248, Section No., Lab No., TA’s name and date.

   b) **Objectives:** Enumerate 3 or 4 of the topics that you think the lab is meant to teach you. DO NOT REPEAT the wording in the lab manual procedures. There should be one or two sentences per objective. Remember, you should write about what you will **learn**, not what you will **do**. These are not necessarily the same things.

   c) **Design:** This part contains all the steps required to arrive at your final circuit. This should include diagrams, tables, equations, K-maps, explanations, etc. Be sure to reproduce any tables you completed for the lab. This section should also include a clear written description of your design process. Simply including a circuit schematic is not sufficient.

   d) **Schematics:** As part of the design process for the first four labs, you will create gate-level schematics for each design and turn them in with your report. The schematics of the designs must be complete. You should be able to implement it directly using standard logic ICs and create a working circuit. Since the design process is completed before you come into lab, you may turn in a hand-drawn schematic, but it must be neatly drawn. Schematics that are difficult to read will receive no credit.
e) **Questions:** Specific questions asked for the lab (given in the lab report) should be answered here.

3. Late pre-labs will have 50% of the points deducted for being one day late. If a pre-lab is two days late, a grade of 0% will be assigned. The pre-lab for a two-week lab is due at the beginning of the first week’s lab period.

4. Your work must be original and prepared independently. However, if you need any guidance or have any questions or problems, please do not hesitate to call your Teaching Assistant (TA) during office hours. Copying any pre-lab will result in a grade of zero. The incident will be formally reported to the University.

5. Each laboratory exercise (circuit) must be completed and demonstrated to your TA before the pre-lab for the subsequent lab is due (1-2 weeks later) in order to receive working circuit credit.

This is the procedure to follow:

a) **Circuit works:** If the circuit works during the lab period (3 hours), then call your TA, and he/she will sign and date your pre-lab report. For Lab # 5 and higher, you should then save your Verilog Modules on your own Zip disk before leaving the lab. DO NOT SAVE ANY OF YOUR FILES ON THE LAB PC’S DEFAULT DRIVES, SINCE THE PC STORAGE IS REFRESHED EACH TIME IT BOOTS. This is the end of this lab, and you will get a complete grade (100) for this portion of the lab.

b) **Circuit does not work:** If the circuit does not work, you must make use of the lab open times for the lab room (Zachry 115C) to complete your circuit. When your circuit is ready, contact your TA to set up a time when the two of you can meet to check your circuit.

6. You are guaranteed a computer and workspace in 115C Zachry only during your lab period. If you need to work on your circuits at a time other than your regularly scheduled lab period, the lab in 115D either is open or can be opened any time the instrument room is open (8 am - 9 pm M-F and 9 am - 5 pm Saturday). However, if another lab section is in progress, ask the TA if he/she has any open lab stations.

7. Attendance at your regularly scheduled lab period is required. An unexpected absence will result in loss of credit for your lab. Your lab instructor may permit rescheduling if you arrange for the change ahead of time.
Detailed Grading Policies

A. Pre-lab report

1. A neat thorough pre-lab report must be presented to your lab TA at the beginning of your scheduled lab section.

2. If you need your pre-lab report during the lab, then i) you can make another copy of your pre-lab report before the lab; or ii) show your pre-lab report to your TA and get his/her initials on your report.

3. Each pre-lab report includes the following sections: a) cover page; b) Objectives (10 points); c) Design (40 points); d) Schematics (40 points); and e) Questions (10 credit for each question if any). Except for the k-maps, circuit diagrams and schematics, the other parts of the pre-lab report should be printed. Otherwise, up to 20 points will be deducted due to lack of adherence to the pre-lab rules.

4. Penalty for the late report: If you keep your pre-lab report during the lab but without getting the TA's initials on it, and you turn in your report at the end of the lab, your pre-lab report will receive only 80% credit. If your pre-lab report is 1 day late, it receives 40% credits. If your pre-lab report is 2 days late, it receives 10% credit.

B. Post-lab report

1. A neat thorough post-lab report must be submitted to your lab TA after the completion of the lab (the latest deadline the TA can allow for the post-lab submission is before the start of the next lab: due for Tuesday lab is this Thursday and for Thursday lab is next Tuesday.) You can show your design and experiments anytime during that particular lab week to your TA by taking an appointment to meet in the lab class.

2. Each post-lab report includes the following components: a) cover page and b) questions (10 points for each question); Except for the k-maps, circuit diagrams and schematics, the other parts of the post-lab report should be printed. Otherwise, up to 20 points will be deducted due to lack of adherence to post-lab rules.

3. Penalty for the late report: If your post-lab report is 1 day late, it receives 60% credits. If your post-lab report is 2 days late, it receives 10% credit. If you have part of the post-lab to be done in the lab (e.g. print waveforms, output response etc.) which you couldn’t complete during lab hours, then you can submit that part correctly labeled with your name, your section number and other details along with your experimental data.
Tips and Tricks of Bread-Boarding

The first four labs require extensive bread-boarding of digital circuits. Here are a few tricks which will help you in wiring up your circuits with ease:

1. Vertical lines in each partition are connected together i.e. they are short internally but the lines are not shorted across the partitions. Each vertical line in a different partition is independent.
2. Horizontal lines are not connected together except for the power and ground line.
3. Power and Ground lines shown horizontally in the figure below are connected together. The supply voltage of 5V needs to be connected to one of the pins in the power line for the entire row to be at 5V. Similarly, the Ground signal needs to be connected to one pin in the Ground line.

![Figure showing breadboard to be used in this class](image)
4. All wires connected to different Integrated Circuit (IC) pins in the design should traverse horizontally or vertically only. Don’t try to connect wires by taking them diagonally across the breadboard since it will result in a messy design and will be very hard to debug.

5. All the IC’s in the design should be placed across one of the splits as shown in the figure above. Do not place ICs in one partition because two pins of the IC on opposite sides will be shorted together which may burn the IC.

6. Use smaller wires when connecting nearby points on the breadboard. Please use a wire stripper to shorten wires before connecting if smaller sized wires are not available. This will help in keeping your design clean and easy to debug.

7. You may decide to always place the IC in such a way that the IC notch is located on the left hand side as shown in figure above. This helps you in identifying pin number easily and conveniently. Similarly, you may decide to use color codes while wiring up your design. You may assign black wires for the least significant bit, white for the next significant bit, green for the next and red for the most significant bit. The color code and order of assignment is entirely your choice and for your convenience in identifying wire connections easily while debugging circuits.

8. Before placing the ICs on the breadboard, plan the placement of your chips in such a way that it minimizes wiring distance on the breadboard. Chips connected to each other directly through wires should be placed adjacent to each other on the breadboard. For example, in your design if LED display inputs are connected to OR gate outputs then try to place the OR gate IC as close as possible to the LED display.

9. The figure above shows the pin-out of an inverter chip 74ALS04. If we keep the IC in the position above with the IC notch towards left then the pin numbers are counted in anti-clock wise direction as shown in the figure. The left most pin on
the bottom row is pin #1 and the left most pin in the top row is pin #14 in a 14-pin IC. Never connect the output of any inverter to the supply or ground. This will damage the IC.

10. **DIP switch**: A DIP switch is a set of manual electric switches that are packaged in a group in a standard dual in-line package (DIP) as shown in the Figure below (the whole package unit may also be referred to as a DIP switch in the singular). This type of switch is designed to be used on a printed circuit board or bread board along with other electronic components to feed inputs into a design from the user. DIP switches often come in packages of seven or eight. The eight switch package offers up to 256 total combinations. In this lab class, we will use these DIP switches to read in a ‘0’ (Off condition) or a ‘1’ (On condition) from the user.

![Figure showing a 16-pin DIP Switch](image)

11. **LED Bank**: This is a bank of 12 LEDs assembled in a line as shown in the Figure below. The corresponding Ground and Power legs protrude out from the bank. Each LED has two legs which are to be connected to power it on. The longer one is generally used for Power and the shorter one is used for Ground. If you need multiple LEDs in your design, then connect the corresponding legs to ground and power.

![Bank of 12 LEDs](image)
12. **Multimeter:** Multimeter is an instrument used to measure current, voltage, resistance etc. The multimeter to be used for this class is shown below. The multimeter has a LCD Display through which the measures values can be read. The knob in the center is used to select the mode of operation. The mode of operation sets up the multimeter to measure a particular quantity. The pins at the bottom are used to connect the multimeter to the circuit whose voltage or current needs to be measured. The pins at the bottom should be chosen appropriately, depending on the type of measurement.

![Front Panel of a Multimeter from B&K Precision](image)

13. **Oscilloscope:** An oscilloscope is a device with which we can see waveforms of signals with respect to time. The figure shown below is an oscilloscope from HP which has 2 channels and supports a bandwidth of 100MHz. The two channels enable us to see two different waveforms connected to the two different input probes of the oscilloscope. Both the waveforms can be displayed simultaneously on the LCD screen. Alternatively, you can also choose from the front panel settings to display just one of the two channels. Each of these channels, have a
vertical positioning knob, Volts per Division knob. The time per division knob is set for both the channels.

![Front panel of a HP 100 MHz Oscilloscope](image)

14. **Power Supply:** The front panel of the power supply we are going to use in this lab is shown below. In this lab, mostly we will be using the +6V supply pin which will be adjusted and fixed to +5V as most of our digital ICs operate at this supply voltage. If you need a second power supply, you can use the +20V supply adjusted at the required voltage level.

![Front Panel of a Power Supply from Hewlett Packard](image)

In case of any other queries regarding use of instruments, please contact your TA.
Lab 1: Introduction to Combinational Design

PART I
Study of Standard Gates

1.1 Introduction
The purpose of this experiment is to introduce you to the basics of circuit wiring, troubleshooting, logic interpretation, and gate behavior. In this lab, you will test the behavior of several of the basic logic gates and you will connect several logic gates together to create simple circuits.

1.2 Background
None.

1.3 Pre-lab
The T.A. will demonstrate the use of oscilloscopes, power supplies and multimeter in the beginning of this lab.

1.3.1 Pre-lab Deliverables
None.

1.4 Lab Procedure
We will look at the behavior of some basic logic gates, including inverters, AND gates, OR gates, XOR gates, NAND gates, and NOR gates. Each of these gates is embedded in an integrated circuit package. See Appendix A for the pin-outs, electrical and timing characteristics of these circuits. More detailed specifications may be found in the complete datasheets available on www.alldatasheet.com.

1.4.1 Experiment 1
We will start by setting up the DC power supply and multi-meter for our use. Be sure both are turned off. Then check to see that the multi-meter is set to measure DC, and be sure the red lead is connected to the red multi-meter input that is marked for voltage. Finally, set the scale to the range you need to measure (usually between 0V to 5V for digital circuits). Now, set the DC power supply voltage output to zero (turn the coarse adjustment counterclockwise until it stops). Connect the red lead of the power supply to the red lead of the multi-meter. Likewise, connect the black lead of the power supply to the black lead of the multi-meter.

NOTE - Do not connect Power (RED) and Ground (Black) together. This will cause a short.
Turn on both the multimeter and the power supply. The multimeter should read very close to zero. Turn the coarse adjustment clockwise until the multimeter reads 5V. If the multimeter display does not change significantly when you turn the coarse adjustment, turn the power supply off and recheck your connections. You may have a short. When the multimeter reads 5V, the adjustments are complete and you should turn off the power supply. You are ready to test your first gate. We will start by wiring a 74ALS04 (inverter) gate. Please refer to the pin configuration given in Appendix A. Insert the 74ALS04 chip onto the breadboard. Be sure you are not shorting pins together. **Identify the power (VCC) and ground (GND) pins for the 74ALS04 from the pin-out of the 74ALS04 in Appendix A.** Connect the VCC pin to the red lead of the power supply and connect the GND pin to the black lead of the power supply. This chip (7404) contains 6 different inverter gates. Each inverter gate has an input pin and a corresponding output pin. Choose one of the gates and connect the red lead of the multimeter to the gate output. The black lead of the multimeter should always be connected to the black lead of the power supply (at the GND pin). Then connect a wire from either the VCC pin to the input (for a logic High input) or from the GND to the input (for a logic Low input). Do not connect both at the same time, as this will cause a short. Turn on the power supply and observe the gate output. Assume A is the input to the inverter (either H or L) and that Y is the output. Fill in Table 1.1 in your post-lab report according to the logic behavior that you observe.

<table>
<thead>
<tr>
<th>A</th>
<th>Y (Volts)</th>
<th>Y (H/L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.1: Truth Table for Inverter**

**Note:** First fill in the second column of the table using the readings from the multimeter. Then determine the answers to the last column based upon these readings. If the output is high (H), the multimeter will read approximately 3.9V – 4.2V; when it is low (L), the multimeter will read about 91.9 mV. If you read a voltage between these values, you have likely wired your circuit incorrectly.

### 1.4.2 Experiment 2
We are going to repeat the same experiment with the gates 74LS00 (NAND), 74LS02 (NOR), 74LS08 (AND), 74LS32 (OR), and 74LS86 (XOR). Note that each of the gates has two inputs and one output. Fill in Table 1.2, 1.3 below in your post-lab report to indicate the observed responses of these gates. So far we have reviewed the voltage behavior (H/L) of various gates. This is an abstract way to interpret this voltage behavior which is called **logic interpretation**.
### Table 1.2: Truth Table for AND & OR Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>And2 (V)</th>
<th>And2 (H/L)</th>
<th>Or2 (V)</th>
<th>Or2 (H/L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 1.3: Truth Table for NAND, NOR & XOR Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Nand2 (V)</th>
<th>Nand2 (H/L)</th>
<th>Nor2 (V)</th>
<th>Nor2 (H/L)</th>
<th>Xor2 (V)</th>
<th>Xor2 (H/L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you still have questions, see your TA.

### 1.5 Post-Lab Deliverables

a) Fill in Table 1.1 with the truth table of an inverter and corresponding voltage levels.

b) Fill in Table 1.2 with the truth table of 2-input And & Or gates and corresponding voltage levels.

c) Fill in Table 1.3 with the truth table of 2-input Nand, Nor & Xor gates and corresponding voltage levels.

### PART II

**Inverter Characteristics and Ring Oscillator**

#### 1.6 Introduction

The purpose of this experiment is to introduce you to the concepts of Inverter voltage transfer characteristics, switching voltages, clock, and gate delays. In this lab, we will first study the behaviour of a single inverter. Then we will cascade an odd number of inverters in a ring to create a ring oscillator.

#### 1.7 Background

**1.7.1 Voltage Transfer Characteristics**

The voltage transfer characteristic of a device is a plot showing how the output voltage of a device varies with change in the input voltage. It is obtained by continuously varying...
the input voltage on the x-axis (independent) and output voltage on the y-axis (dependent). It is generally used to study the operation of a device. For example in Figure 1.1, the VTC can be used to determine that when the input voltage is 1V, then the output voltage is 5V as shown in Figure 1.1.

![Figure 1.1: Voltage transfer characteristic](image)

1.7.2 Ring Oscillator

A ring oscillator is a circuit composed of an odd number of inverter gates connected in a ring as shown in Figure 1.2. The NOT gates, or inverters, are arranged in a ring; the output of the last inverter is fed back into the first. Because a single inverter computes the logical NOT of its input, it can be shown that the last output of a chain of an odd number of inverters is the logical NOT of the first input. This final output is asserted a finite amount of time after the first input is asserted; the feedback of this last output to the input causes oscillation.

![Figure 1.2: Five Stage Ring Oscillator](image)

In Figure 1.2, let’s say the input value at ‘A’ is ‘1’ at time Φ. If the delay of an inverter is Dinv, then the time after which a ‘0’ is fed back into the same node ‘A’ is N.Dinv. Similarly, ‘1’ appears at node ‘A’ again after a further delay N.Dinv. Therefore, the time period (T) of the ring oscillator is 2.N.Dinv where N (odd) is the number of inverter
stages in the ring oscillator. The frequency of oscillation is 1/T. The ring oscillator is an important circuit since it can be used to find the delay of a single inverter, by measuring the time period of oscillation and dividing it by 2N. This is helpful if Dinv is too small to measure using the available equipment. Another reason why the ring oscillator is important is that it can be used to create a clock signal as shown in Figure 1.4.

1.8 Pre-lab
The TA will demonstrate the use of oscilloscope esp. channel selection and automatic triggering of channel on the oscilloscope which is required for observing waveforms on the display.

1.8.1 Pre-lab Deliverables
None.

1.9 Lab Procedure
Complete Experiments 1-2 listed below.

1.9.1 Experiment 1
In a circuit, logic variables (values ‘0’ and ‘1) can be represented as levels of voltage. The most obvious way of representing two logic values as voltage levels is to define switching voltage levels.

![Figure 1.3: Voltage switching levels](image_url)
To implement the switching – voltage concept, a range of low and high voltage levels is defined, as shown in Figure 1.3 for input as well as output of an inverter. This figure indicates that voltages in the range Ground to $V_{\text{OL}}$ represent logic value ‘0’ for the output of an inverter. Similarly, the range from $V_{\text{OH}}$ to $V_{\text{DD}}$ corresponds to logic value ‘1’ of the output stage of the inverter. For the input of an inverter, the voltage range from Ground to $V_{\text{IL}}$ represents logic value ‘0’ and voltage range from $V_{\text{IH}}$ to $V_{\text{DD}}$ represent logic value ‘1’. Logic signals do not normally assume voltages in undefined range except in transition from one logic value to the other. These voltage points are shown on the inverter VTC in Figure 1.4. For an inverter, $N_{\text{M}}$ and $N_{\text{L}}$ are defined as the high side and low side noise margins. The high output excursion shouldn’t be larger than the high input excursion. Same goes for the low excursions. If this is violated, one of the corresponding noise margins is negative. Noise margins provide a cushion for noise immunity. If the $V_{\text{DD}}$ or Ground of the driver glitches relative to the driven gate, the noise margins can provide noise immunity up to a certain level. The magnitude of glitch protection provided by a digital circuit is given by the Noise margins. In our case for this experiment, assume $V_{\text{OL}}$ and $V_{\text{OH}}$ are Ground and $V_{\text{DD}}$ respectively.

Use the continuous voltage output pin of your power supply and connect it to input of an inverter gate and measure its output voltage with the multimeter while the input voltage of the gate changes from 0 to 5 volt. Draw $V_o$–$V_i$ curve for the inverter gate. Plot the VTC. Demonstrate the results to your TA.
1.9.2 Experiment 2
This experiment will introduce you to the concept of clock signals and circuit delays. Clock is a control signal that allows the changes in the states of digital circuits to occur only at well-defined time intervals. Since such a signal orchestrates the timing of a digital circuit, it is referred to as a clock. Typically, rising clock edges or falling clock edges lead to changes in the circuit states. Actually, clock is a periodic signal with time period \( T \) (as shown in Figure 1.5) and a frequency \( F = 1/T \).

By connecting an odd number of inverter gates in a ring, you will make a clock signal. Fig 1.2 depicts the simple a ring oscillator circuit (Use \( N = 5 \)). You should connect the last gate output to first gate input. This feedback causes the circuit to oscillate at a particular frequency. To see this signal on the oscilloscope, connect the output of any inverter to one of the oscilloscope channel and auto trigger that channel by pressing the “auto trigger” button on the front panel (below the LCD) of the oscilloscope. Then look at the output signal of any inverter in the ring on the oscilloscope and measure its time period and oscillation frequency. Then calculate average delay of each gate and demonstrate your results to your TA.

![Figure 1.5: Sample Clock Signal](image)

1.10 Post-Lab Deliverables
a) Draw the voltage transfer characteristics on a piece of graph paper with \( V_{in} \) on x-axis and \( V_{out} \) on y-axis. Mark the range of voltages in output and input as Logic 0 and Logic 1. Calculate the range of input voltage for which the inverter shows Logic 1 as output. Also calculate the range of input voltage for which the inverter shows Logic 0 as output.

b) Derive the single-stage delay of the Ring Oscillator from the time period of oscillation that you see in Experiment 2. If the delay of one inverter is 10ns. What will be the frequency of the signal generated from a 21 stage ring oscillator?

c) Are the signals at P, Q, R, S in Figure 1.2 periodic? If so, what is their time period? How do these signals differ from the signal at node A?
Lab 2: Logic Minimization and Karnaugh Maps

2.1 Introduction
The purpose of this experiment is to introduce you to the application of digital electronics in real life environments to perform logical decisions and compute results. We will use logic minimization techniques (Karnaugh-Maps) which will aid the efficient implementation of these functions.

2.2 Background

2.2.1 Binary Coded Decimal (BCD)
Binary coded Decimal is an encoding method in which each decimal digit is represented by its own binary equivalent using 4 bits. This is generally done for the purpose of displaying results.

2.2.2 Seven-Segment Display
The seven-segment display has seven LED segments which are arranged in the shape of the digit ‘8’ as shown in the Figure 2.1 below. It takes in a set of four bits as inputs, and based on the inputs a unique combination of LEDs start glowing. In such a display, a specific set of input patterns result in a specific digit being shown on the display by means of glowing LEDs. The way BCD digits from ‘0’ to ‘9’ get displayed is shown in Figure 2.2.

![Figure 2.1: Seven-Segment Display](image-url)
2.2.3 BCD to Seven-Segment Driver Chip
A BCD to seven-segment driver takes a BCD digit as input (usually 4 bits long) and decodes it into seven outputs which serve as the inputs for each of the seven segment of a seven-segment display. In the case shown in Figure 2.3, input A is the least significant bit and D is the most significant bit. The outputs bits of this decoder (a, b, c, d, e, f, g) correspond to the different segments on the Seven-segment display as shown in Figure 2.1.

![Figure 2.3: Seven Segment Display Driver Input/Output](image)

2.3 Pre-Lab

2.3.1 Design 1
In this design, we will implement a digital circuit which will compute the total profit made by a person named John who runs an animal farm. Although John has a lot of experience in running the farm, he doesn’t know how to compute the profits and needs a device which can compute the profit based on the kind of animals that he is rearing. John has to follow a set of conditions according to which he goes about setting up the farm.
The conditions are listed below:

i) He can rear cows, sheep, hogs and chicken in his farm but cannot rear all at the same time.

ii) The farm doesn’t have enough space to rear more than 2 different types of animals. Therefore, his farm never has more than 2 types of animals at any particular time.

iii) Each of the animals earns a certain level of profit for the farm. Cows earn him a profit of 4 units, sheep earns him a profit of 2 units, hogs earn a profit of 3 units and chicken being small creatures earn him a profit of 1 unit.

iv) The total profit of the farm is the sum of profits made by each animal that he rears.

v) Hogs and chicken have very different food, so John needs to spend extra for the transportation of food for each separately. Therefore, if he plans to rear hogs and chicken together then his profits go down by 1 unit.

vi) Similarly, if he rears cows and chicken his profits go down by 1 unit because of extra transportation costs. Additionally, the vicious cows usually have a tendency to kill some chicken in the farm. This further reduces his profits by 1 unit.

vii) However, cows and sheep do very well if reared together. They have many similarities especially with respect to the kind of food they eat. Also due to the huge demand of sheep and cows, the profit they make goes up by 1 unit if cows and sheep are reared together.

Your objective is to design a digital circuit which takes in four inputs Chicken (I), Sheep (S), Cow (C) and Hog (H) and outputs of the profit using 3 bits P2, P1, P0 with P2 being the most significant bit and P0 being the least significant bit. Please put “don’t cares” for all those input combinations that are not possible due to the nature of the question. You can display the three bits of the profit computer output using the LED Bank given to you. Each LED in the LED Bank has two legs. The longer leg of the LED should be connected to your signal (P0, P1, and P2) and the shorter leg should be connected to the ground. You should use three separate LEDs on the LED bank to display P0, P1, and P2 respectively. Don’t forget to provide a common ground to all the devices in your circuit. The block diagram of the connections to be made is given in Figure 2.4 below.
2.3.2 Pre-Lab Deliverables

a) Draw K-Maps for each bit of the output of your Profit Computer (P0, P1, and P2) and find the minimized Boolean function to implement P0, P1 and P2 as a function of H, C, S, I i.e. P0=f(H, C, S, I); P1=g(H, C, S, I); P2=h(H, C, S, I).

b) Write down the minimized Boolean function for each output bit as function of I, S, C, H.

c) Draw the gate level schematics for each bit of the Profit Computer. Each bit should have a separate schematic showing implementation using standard gates.

2.4 Lab Procedure

Complete Experiment 1 listed below.

2.4.1 Experiment 1

Implement and test your Profit Computer design using the gates in the Integrated Circuits (ICs) provided. Use wires from the power rails (+5v or 0v) to read in all your inputs from the user into your design. For any input combination provided, the corresponding profit in binary should be displayed on the LED Bank given to you. When it is working demonstrate it to your TA.

2.5 Post-Lab Deliverables

a) Make a 4-input (I, S, C, and H), single output (P) table and list your profits (in decimal) as displayed on the seven-segment. For all the combinations that are not possible, mark an ‘X’ in the output column. Verify your result against the truth tables drawn during your pre-lab.
b) Using K-maps derive the minimized Boolean functions for each output of the BCD to Seven Segment Driver Outputs. Your input should be the driver inputs (D, C, B, A) with A being the least significant input and D being the most significant input (as shown in Figure 2.4). Similarly, the outputs should be the driver outputs (a, b, c, d, e, f, g, h). Your design should take in BCD inputs and decode them for the seven-segment display. Hint: Remember that a 4-bit BCD number cannot take on a value greater than 9.

c) Derive the Product of Sum (POS) expression for a 3-input (A, B, C) XNOR Gate. Assume that output of the gate is called ‘Y’.
Lab 3: Study and Implementation of Adders

3.1 Introduction
The purpose of this experiment is to introduce you to the design of simple combinational circuits, in particular half adders and full adders. You should also be able to design an $n$-bit ripple carry adder using $n$ full adders.

3.2 Background

3.2.1 Logic Minimization
Logic minimization is a method to reduce the number of logic gates required to implement a Boolean function. Among the various ways used to minimize Boolean functions, the three most popular ones are Karnaugh-Maps, Quine-McCluskey Algorithm and Espresso heuristic logic minimize.

- Espresso Heuristic Methods: These methods are very fast, and can achieve results that are almost as good as the Q-M approach with respect to the number of product terms.
- Quine-McClusky Method: This is an ‘exact’ method (which means that it returns a solution with the fewest product terms) but it is very slow requiring lot of computation.
- Karnaugh Maps: This is a visual method which implements the Q-M algorithm. It works only on function with 5 or 6 variable at most.

In this lab, we will use Karnaugh-Maps to reduce our Boolean expressions.

3.2.2 Karnaugh-Maps
The Karnaugh-map, also known as a K-map, is a tool to facilitate the simplification of Boolean algebra expressions. In K-maps, the Boolean variables are transferred generally from a truth table and ordered according to the principles of Gray Code in which only one variable changes in between squares. Once the table is generated and the output possibilities marked, the data is arranged into the largest group possible. The number of min-terms covered in a group should be a power of two. The minimized min-terms are generated through the axiom laws of Boolean algebra.

3.2.3 Adders
We will be using two types of adders for this lab and throughout the semester. A half adder adds two bits together and therefore has possible sums of 0, 1, or 2. A full adder adds a carry-in bit to two bits and has possible sums of 0, 1, 2, or 3. See Figure 3.1 and 3.2.
To create an \( n \) bit ripple carry adder, we connect \( n \) full adders together in the fashion of Figure 3.3. The carry-out from a full adder becomes the carry-in for the next significant bit in the next full adder.

In Figure 3.3 you can see the interaction of the input and output signals in a 2 bit adder circuit.
3.3 Pre-lab

3.3.1 Design 1
Construct a truth table for a half adder circuit, including the sum and carry-out outputs. Write equations for the outputs of the half adder that are minimal in terms of the number of gates required for implementation. **Note that XOR and NOT gates count as one gate each.** Draw a gate-level circuit diagram also called a schematic for the half-adder.

3.3.2 Design 2
Construct a truth table for a full adder circuit, again including the sum and carry-out outputs. Write the equation for each output that is minimal in terms of the number of gates required for implementation. Draw a gate-level schematic.

3.3.3 Design 3
Design a 2-bit ripple carry adder using the full adder equations derived previously include the carry-in signal to the LSB. There should be two 2-bit inputs: (A1 A0) and (B1 B0) and one single bit input Cin which is the carry-in to the LSB of the adder. There should be 3 outputs: (S1 S0), which is the sum, and Cout, which is the carry-out of the MSB. Provide the truth table for the 2 bit adder.

3.3.4 Pre-lab Deliverables
a) Draw truth table, K-maps and gate level schematic for the Half Adder in Design 1.
b) Draw truth table, K-maps and gate level schematic for the Full Adder in Design 2.
c) Draw the truth table and gate level schematic for the 2-bit adder in Design 3.

3.4 Lab Procedure
Complete Experiments 1-3 listed below.

3.4.1 Experiment 1
Implement and test your half adder design using the gates in the Integrated Circuits (ICs) provided. Read inputs (A, B) from the user into your design and display the output on two consecutive LEDs (Cout, S) from the LED Bank provided to you with each bit displayed in accordance to its positional significance. When it is working demonstrate it to your TA.

3.4.2 Experiment 2
Implement and test your full adder design using the gates in the ICs provided. Read in inputs (A, B, Cin) from the user into your design. The output bits of the full adder should be displayed on two consecutive LEDs (Cout, S) from the LED Bank provided to you with each bit displayed according to its significance. When it is working, demonstrate it to your TA.
3.4.3 Experiment 3
Implement your design for the 2-bit ripple carry adder from pre-lab Design 3. Read in the inputs (A1, B1, A0, B0, and Cin) from the user into your design and the output bits of the 2-bit adder should be displayed on three consecutive LEDs (Cout, S1, and S0) from the LED Bank provided to you with each bit displayed in accordance to its significance. Test it using the truth table drawn during Design 3 and demonstrate it to your TA.

3.5 Post-Lab Deliverables
a) Draw the truth table, K-Map and schematics for Design 1.
b) Draw the truth table, K-Map and schematics for Design 2.
c) Draw the truth table and schematics for Design 3 using full adder blocks.
d) Solve the following:
   i) Using your design for the full adder, determine the worst case propagation delay from the inputs to the carry-out output bit. Assume each gate has the same delay of 1 unit. Show the maximum delay path in your schematic. The maximum delay path is known as the critical path for that particular combinational block.
   ii) How would you make an 8-bit ripple carry adder using only 2-bit adder blocks? (Do not use individual full adders as building blocks.) Draw the schematics of the 8-bit ripple carry adder using 2-bit adder as building blocks.
   iii) How would you make a 4-bit ripple carry adder using only 3-bit adder blocks? Hint: Use one 3-bit adder for the least significant 2 bits and use another 3-bit adder for the most significant 2 bits. Again, do not use individual full adders as building blocks. Draw the schematics using 3-bit adders as building blocks for the design. Show clearly all the input signals to both the 3-bit adder blocks.
Lab 4: Multiplexer Based Arithmetic Logic Unit

4.1 Introduction
The purpose of this experiment is to design a basic four bit arithmetic logic unit which does elementary computation like addition, subtraction and bit-wise AND. A multiplexer based control will be used to choose between the operations.

4.2 Background

4.2.1 Two’s Complement Numbers
There are many ways to represent numbers in binary. Here, we will discuss one possible way called the Two’s Complement representation. We will find that this is a convenient and efficient manner of representation for signed numbers. To begin, let’s consider 3-bit numbers. With 3 bits, we can represent $2^3 = 8$ numbers. Table 4.1 shows the straightforward unsigned integer representation using three bits. In reality, the whole process of assigning a numerical value to a particular binary representation is arbitrary and up to the designer’s discretion. It is the convenience and efficiency of an assignment that makes a designer prefer one assignment over the other. For instance, in Table 4.1 we could have assigned “011” to represent 0, “111” to represent 1, etc. However, on inspecting Table 4.1, you can see that:
1. The normal ordering of the positive integers is maintained by our assignment (i.e. subsequent binary assignments are greater than the preceding assignment by 1). This leads to an efficient assignment, as we can use the normal adders of Lab 2 to add any numbers.
2. We have an unified representation for 0, regardless of the number of bits (e.g. for 3-bits we have 0 being represented by “000”, for 7-bits we would have 0 being represented by “0000000”, and so on). This makes our assignment convenient. If we had to extend our design by scaling (i.e. increasing or decreasing) the numbers of bits, we would not need to worry about re-assigning the zero.

<table>
<thead>
<tr>
<th>WORD</th>
<th>UNSIGNED MAGNITUDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.1: Unsigned number representation scheme
So far discussion was restricted to unsigned numbers. Now, we can now go ahead and try to represent both positive and negative (i.e. signed) integers; Table 4.2 shows one possible representation.

<table>
<thead>
<tr>
<th>WORD</th>
<th>SIGNED MAGNITUDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Signed Number Representation scheme

In this representation,

\[-x = (2^n - 1) + x\]  \hspace{1cm} (1)

On inspecting this representation we notice some characteristics:
1) The ordering of integers is maintained.
2) The representation is symmetrical about zero, as with integers.
3) The zero maintains its representation regardless of the number of bits.
4) Using a regular adder (e.g. the one from Lab 2), we can add complementary pairs (e.g. \(n\) and \(-n\)) to get zero.
5) Using only inverters and a regular adder, we can obtain the negative of any represented number.

The first three points refer to the convenience of the representation; it maintains the properties of the mathematical set of integers. The last two points attest to the efficiency of the representation; no new hardware elements are required to manipulate these numbers. For these reasons and more, the representation of Table 4.2 is widely-used and
is called the Two’s Complement Representation. We will study more about Signed number arithmetic later in the lab class.

### 4.2.2 Multiplexer

A Multiplexer (Mux in short) is a device which selects a specific input from all the inputs and passes its value to the output. The selection is done based on the data that’s fed to the control lines of the Multiplexer. The figure below shows a 2:1 Multiplexer and a 4:1 Multiplexer. Depending on whether S is zero or one, a 2:1 Multiplexer chooses A as the output or B as the output respectively. Similarly, depending on the combined value on S1 and S0, the 4:1 multiplexer chooses which among the inputs A, B, C, and D is chosen to be passed to the output.

![Figure 4.1: (L-R) 2:1 Multiplexer, 4:1 Multiplexer](image)

### 4.2.3 Arithmetic Logic Unit (ALU)

In computing, an Arithmetic Logic Unit (ALU) is a digital circuit that performs arithmetic and logical operations. The ALU is a fundamental building block of the Central Processing Unit (CPU) of a computer, and even the simplest microprocessors contain one. The processors found inside modern CPUs and Graphics Processing Units (GPUs) have inside them very powerful ALUs. Sometimes these units can house multiple ALUs for parallel and faster operation.

![Figure 4.2: Typical ALU Schematic Symbol](image)
In the above schematic symbol, A and B are operands which feed in data to the ALU; F is the control or operation selector bus; R is the output which gives the result of the operation performed over A & B; D outputs the status of the ALU.

### 4.3 Pre-Lab

#### 4.3.1 Design 1

The objective of this experiment is to design an elementary ALU which has the following inputs:

1) A [3:0] : First operand of four bits
2) B [3:0] : Second operand of four bits
3) C0, C1: Control Bits to decide operation

The ALU has the following outputs:

1) OUT[3:0]: Showing the computed result
2) Cout: This indicates whether the operation has resulted in overflow or not.

![Figure 4.3: Block Diagram of ALU with Input/Output](image)

The operations which the ALU should be able to perform are the following:

1) **BITWISE AND**: When C0=0, this operation is selected. The data bus A and B should be bitwise ANDed when this operation is selected.
2) **ADDITION**: When C0=1 and C1=0, the addition operation is selected. In this case, the result bus OUT is the sum of data buses A and B. The Signal ‘COUT’ should be driven to logic ‘1’ if the addition overflows 4 bits else it should be driven to logic ‘0’.
3) **SUBTRACTION**: When $C_0=1$ and $C_1=1$, the subtraction operation is selected. In this case, the signal OUT is the result obtained when B is subtracted from A. Note that $C_0$ is 1 when the operation is addition or subtraction. The logic value of $C_1$ decides which operation out of these two is chosen.

Hint: Your design should use a single four bit adder block. Remember subtraction can be implemented simply by adding the first operand with the 2’s complement of the second operand. Assume there is no overflow in your subtraction.

**4.3.1 Pre-Lab Deliverables**

a) Draw the block diagram of your design using the following sub-blocks: an Adder/Subtractor block, 2-input AND gates and 2:1 multiplexers. Your diagram should clearly show the flow of signals from input to output.

b) Draw the gate level schematics of the all the sub-blocks used in part (a). This includes Adder/Subtractor block, multiplexer block etc. For the adder/Subtractor block, you can use four bit full adders as a sub blocks.

**4.4 Lab Procedure**

Complete Experiments 1 listed below.

**4.4.1 Experiment 1**

Implement and test your ALU design using the gates in the Integrated Circuits (ICs) provided. Read in all your inputs from the user into your design. There should be 8 input wires for data bus A and B and 2 input wires for control lines $C_0$ and $C_1$. Connect your ALU outputs $COUT$ and $OUT$ to 5 consecutive LEDs in the LED bank exactly in order of their positional significance. $COUT$ should be placed leftmost followed by $OUT [3]$ and $OUT [0]$ should be placed rightmost. When the ALU is working demonstrate it to your TA.

**4.5 Post-Lab Deliverables**

a) Draw the schematic of a difference calculation unit. This unit operates as follows:

   If $(A \geq B)$ then it computes $(A-B)$ else it computes $(B-A)$.

   Assume you have a comparator block in your design which gives a ‘0’ when A is greater than or equal to B and gives a ‘1’ when B is greater than A. Your design should use a single four bit adder.

b) Design an 8:1 multiplexer using:

   I) Both 4:1 multiplexers and 2:1 multiplexers
   II) Only 2:1 multiplexers
Draw diagrams showing the Input/Output (I/O) and internal signals of your design.

c) Design a circuit which will implement a shift operation on a 4-bit operand \( A[3:0] \) where the magnitude and direction of shift depends on another 3-bit operand \( B[2:0] \). The circuit should use multiplexers as sub-blocks rather than gates. Here is the specification of your design:

I) If \( B[2] = '0' \) then shift left else shift right.

II) Together \( B[1] \) and \( B[0] \) decide the magnitude of the shift operation.

III) If \( B[1] = '0' \) and \( B[0] = '0' \) means don’t shift at all.

IV) If \( B[1] = '0' \) and \( B[0] = '1' \) then shift by 1 position.

V) If \( B[1] = '1' \) and \( B[0] = '0' \) then shift by 2 positions.

VI) If \( B[1] = '1' \) and \( B[0] = '1' \) then shift by 3 positions.

Hint: Implement your design using multiplexers. Connect wires to different inputs of multiplexers depending on the magnitude and direction of shift.
Lab 5: Verilog and Xilinx ISE

5.1 Introduction
The purpose of this experiment is to introduce you to a hardware description language (HDL) in particular, Verilog. An HDL is a method to describe hardware, by using software. An HDL representation of any hardware block is a software file, which adheres to a specific syntactical format. We will also use a tool called Xilinx Integrated Software Environment (Xilinx ISE) which will help us to convert the code in Verilog to a fully functional design on the Xilinx series of Field Programmable Gate Arrays (FPGAs). In this lab, we will design a simple 8-bit ripple carry order using Verilog, on a Spartan 3E FPGA. We will also use most of the I/Os on the Xilinx Spartan 3E starter Board (Figure 5.1) to read in our input patterns and display the output of our design.

5.2 Background

5.2.1 Verilog
Verilog is a hardware description language (HDL) used to model digital systems. The language supports the design, verification, and implementation of digital circuits at various levels of abstraction. The language differs from a conventional programming language in that the execution of statements is not strictly sequential. A Verilog design can consist of a hierarchy of modules. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and Sequential statements define the behaviour of the module by defining the relationships between the ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all Concurrent statements and all begin/end blocks in the design are executed in parallel. This is the key difference between Verilog and any software programming language. A module can also contain one or more instances of another module to define hierarchy.

Only a subset of statements in the language is synthesizable. If the modules in a design contain only synthesizable statements, software like Xilinx ISE can be used to transform or synthesize the design into a gate level net-list that describes the basic components and connections to be implemented in hardware. The synthesized net-list may then be transformed into a bit-stream for any programmable logic devices like Field Programmable Gate Arrays. Note that this enables a significant improvement in designer productivity- A designer writes their hardware behaviour in synthesizable Verilog and the ISE (or similar) tool realizes this hardware on a hardware platform such as an FPGA. Verilog designs can be written in two forms:

1) Structural Verilog: This is a Verilog coding style in which an exact gate level net-list is used to describe explicit connections between various components, which are explicitly declared (instantiated) in the Verilog code.

2) Behavioral Verilog: In this format, Verilog is written to describe the function of the hardware. The Verilog code in this case describes the behavior of the hardware
without making explicit references to connections and components. A logic synthesis tool is required in this case to convert this Verilog code into gate-level net-lists. Usually, a combined coding style is used where part of the hardware is described in structural format and part of the hardware is described in behavioral format according to convenience.

5.2.2 Field Programmable Gate Arrays (FPGA)
A field-programmable gate array is a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions. In most FPGAs, the logic blocks also include memory elements like flip-flops. A hierarchy of programmable interconnects allows logic blocks to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. Logic blocks and interconnects can be programmed in the field by the customer or designer (after the FPGA is manufactured) to implement any logical function as and when required hence the name "field-programmable logic arrays".
Realizing a design on an FPGA consists of five steps, which are implemented in a software tool like the ISE:

1) Synthesis: This is the process of converting a Verilog description into a gate level net-list.
2) Mapping: This maps the above gate level net-list to the technology specific components on the FPGA.
3) Placing: This step places the mapped components in a manner that minimizes wiring, delay etc.
4) Routing: This step configures the programmable interconnects (wires) so as to wire the components in the design.
5) Programming the FPGA: In this step, the placed and routed design is converted to a bit-stream using the Xilinx ISE tool. The bit-stream generated by the tool is loaded (as a .bit file) on to the FPGA. This bit-stream file programs the logic and interconnects of the FPGA in such a way that the design gets implemented.

5.2.3 Xilinx Spartan 3E starter Board
The Spartan 3E starter board is built around the Xilinx Spartan 3E series of FPGA. The board has the facility to program the FPGA using a USB connection to your PC. The board provides programmable interfaces to a global reset, four push buttons, a rotational knob, four on/off switches, eight LEDs, clock, memories and the LCD Display, as shown in Figure 5.1.

5.2.4 Xilinx Integrated Software Environment (ISE)
Xilinx Integrated Software Environment is a tool from Xilinx which integrates various stages of the FPGA design cycle into one software tool. Xilinx ISE provides processes
for logic synthesis, mapping, placing and routing. It also integrates various simulation tools which are necessary to validate our design at various stages in the design cycle.

Figure 5.1: Components on Xilinx Spartan 3E starter board

5.3 Pre-Lab

5.3.1 Design 1
In this lab, you will use Verilog to implement an 8 bit adder on the Xilinx Spartan 3E FPGA. The Verilog used to describe the adder will be in structural format, containing the exact gate level net-list of the design. Please take a look at the Verilog files provided to know how to write structural Verilog. For using the I/O’s on the board we need some other Verilog files which will interface our adder hardware with the I/Os on the board. The functionality of each Verilog file is described in the section 1.3.2. The complete block diagram of this design is provided in Figure 5.2 below.
In the design above, CLK port is connected to the system clock net. RESET is connected to one of the push buttons provided on the board. The design provided to you uses asynchronous reset with the push button to the east of the rotational switch. ROT_A and ROT_C are two internal switches in the rotational knob. We will learn more about them in the next lab. ROT_C is port connected to the push-button of the rotational switch. SF_D [3:0] is the four-bit data bus which communicated with the LCD Display. The other three ports are control ports for the display. We will learn more about the display in the next lab as well.

In this lab, you will be provided all the Verilog code that interprets the rotational motion of the rotational knob (via ROT_A and ROT_B), as well as the LCD Display signals.

5.3.2 Using the Integrated Software Environment
The aim of this section is to get you familiar with the FGPA Design Flow using Xilinx’s Integrated Software Environment (ISE). A complete step by step procedure to design the simple 8-bit adder is described below:

a) Launch the ISE Project Navigator and create a new design project.
Select Applications → Accessories → Terminal and then run the following commands:

> /softwares/setup/xilinx/ise-10.1
The former sets up the environment in order to run ISE, while the latter actually starts the ISE toolset.

In the Project Navigator, select File → New Project.
The ‘New Project’ Wizard opens as shown in Figure 5.3.
Open a Project Location, say ECEN248Lab in your home directory.

If there is no folder by that name in your home directory, then create a new folder by that name. All projects done in this lab should be saved to this folder.
Type a Project Name, say lab5 as shown in Figure 5.3.
Chose HDL as the Top-Level Source Type and click Next >

![Image of New Project Wizard]

**Figure 5.3: New Project Wizard**

b) The ‘Device and Design Flow’ Dialog appears as shown in Figure 5.4.
Select the following options and click Next >

- **Product Category:** All
- **Device Family:** Spartan 3E
- **Device:** XC3S500E
- **Package:** FG320
- **Speed Grade:** –5
- **Synthesis Tool:** XST (VHDL/Verilog)
- **Simulator:** ISE Simulator
- **Preferred Language:** Verilog

In this step, we are instructing ISE to implement our design on the FPGA that is present on our Spartan 3E started board. Please check the device code on the
FPGA for your board and match it to the device name given above. The package type is also available on the FPGA chip. Check your package type for the FPGA on your board.

c) In this lab, you are provided with all the Verilog files that you only required to use. So, we need not create any new Verilog source files. So, you may skip this step for this lab. This step (step c) is only necessary in subsequent labs when you need to type in your own Verilog files to describe your design. Therefore, when ‘Create New Source’ Dialog appears as shown in Figure 5.5, click on ‘Next’ to skip this step.

In subsequent labs you may need to perform this step, so when the ‘Create New Source’ Dialog appears as shown in Figure 5.5. Click on ‘New Source’ to add a new source (Verilog) file associated with this project. This will open the ‘New Source Wizard – Select Source Type’ window as shown in Figure 5.6. Now, the ‘New Source Wizard – Define Module’ window will open, as shown in Figure 5.7. This wizard is used to define your I/Os prior to describing your hardware. You have to specify I/O names, bus widths and the type of bus for the particular design module. The next window will show a summary of the design created, click ‘Finish’.

You can add new source files anytime in your design cycle by right clicking on the project name in the ‘Sources’ tab and by clicking on ‘New Source File’.

![Figure 5.4: Device Properties](image)
Creating a new source to add to the project is optional. Only one new source can be created with the New Project Wizard. Additional sources can be created and added to the project by using the ‘Project->New Source’ command.

Existing sources can be added on the next page.

Figure 5.5: Create New Source Wizard

Figure 5.6: Select Source Type Window
This will open ‘New Project Wizard – add existing sources’. For this lab, we need to skip this step too. So, click ‘Next’ and ‘Finish’.

e) Now you need to add the Verilog files given to you for this lab to the project. First download the zipped folder for Lab 5 from your lab website. Contact your TA or your course instructor for the exact URL of your lab. Unzip the folder on your machine and ‘copy’ (CTRL+C) all files inside it. Now go to your local ‘ECEN248Lab’ and find the folder named ‘lab5’. ‘Paste’ (CTRL+V) all your Verilog files into this folder. This folder serves as the local folder for current project. It is usually convenient to place all the Verilog files required in the project in a particular project-specific folder.

Now you are ready to add the ISE Verilog files to the current project. Go to the ‘Sources’ window to the left, right click on the project name (say lab5) and click on ‘Add Source’. Press the CTRL key and select the Verilog files that you need to add to the project and click on ‘Open’. The ‘Adding Files’ pop-up window appears click on ‘Ok’. These files will now show up on the ‘Sources’ window.

You can add source files at any time in the design cycle by right clicking on the project name in the ‘Sources’ tab and by clicking on ‘Add Source’.
f) From the Sources window, open any Verilog file by double clicking on it. This will make the Verilog code appear on the top right hand side of the ISE project navigator window. If required, you can edit any file after opening it. After editing, click on **File > Save** to save all the changes that you made to your design. In the ‘Sources’ window, all the Verilog files get displayed in hierarchical order. Here is a brief summary of the functionality of these Verilog files:

1) **rot_intfc.v**: This module interfaces the rotational switch, reset switch etc with the adder module.

2) **rot_switch1.v**: This is the Verilog module which reads in data from the rotational knob and provides it to the rot_intfc module. You can re-use this file in later labs for reading in data from the rotational knob.

3) **FA.V**: This is the structural Verilog net-list of the full adder block which is going to be used in building the 8-bit ripple carry adder block.

```verilog
//Verilog Module for Full Adder
`timescale 1ns / 1ps
module FA(C,S,A,B,Cin);
input A,B,Cin;
output C,S;
wire ab,ca,bc;

xor(S,A,B,Cin);
and(ab,A,B);
and(ca,A,Cin);
and(bc,B,Cin);
or(C,ab,ca,bc);
endmodule
```

4) **adder_8bit.v**: This is the adder module which takes in two operands as input from the rot_intfc module, adds them using a ripple carry adder and outputs the sum of the operands which is provided to the adder_lcd module below.

5) **hex_dec.v**: This module is used by the adder_lcd module. This module decodes hexadecimal nibbles to an eight bit output which corresponds to the bit combination that is required to display that particular digit on the display. For example if you want to display hexadecimal value ‘A’ on the display (corresponding to the nibble 1010), the decoded value for the LCD display is ‘01000001’. This eight bit binary value causes ‘A’ to appear on the LCD
Display. A part of this Verilog code is given below. The bold part of this code shows how a nibble having hexadecimal value ‘A’ (‘1010’) is decoded to hexadecimal (‘41’) or binary (‘01000001’).

```verilog
// Structural Verilog Module
// for 8-bit Ripple Carry Adder Block

`timescale 1ns / 1ps
module adder_8bit(A,B,Sum);
input [7:0] A,B;
output [8:0] Sum;

wire C1,C2,C3,C4,C5,C6,C7;

FA U1 (C1,Sum[0],A[0],B[0],1'b0);
FA U2 (C2,Sum[1],A[1],B[1],C1);
FA U3 (C3,Sum[2],A[2],B[2],C2);
FA U4 (C4,Sum[3],A[3],B[3],C3);
FA U5 (C5,Sum[4],A[4],B[4],C4);
FA U6 (C6,Sum[5],A[5],B[5],C5);
FA U7 (C7,Sum[6],A[6],B[6],C6);
FA U8 (Sum[8],Sum[7],A[7],B[7],C7);
endmodule

always @ (*)
begin
  case (in)
    ...
    // Decoded value for hexadecimal ‘9’
    4'b1001: out=8'h39;
    // Decoded Value for hexadecimal ‘A’
    4'b1010: out=8'h41;
    ...
  endcase
end
```

Parts of the Verilog file: hex_dec.v

6) adder_lcd.v: This module is specific to the adder design of Lab 5. It outputs the 32 character string to be displayed on the LCD display. For other labs and designs, you have to write such an interface file which provides data to the lcd_disp module.
//Module lcd_display is instantiated here
lcd_disp U_lcd_disp (clk, reset, display, SF_D, LCD_E, LCD_RS, LCD_RW);

/*All empty spaces on the LCD display not required for our adder is filled with the code (h’20) which print blank spaces. */
assign temp_line1=96’h202020202020202020202020;
assign temp_line2=104’h202020202020202020202020;

/*Hexadecimal data is encoded to bits using hex_dec module which the LCD will understand. */
hex_dec u_hex1 ({3'b000,data[24]},temp1);
...
...
/*Line 1 and Line2 of LCD Display is assigned here with all the required bits to set the display for our adder. */
assign line1={temp4,temp5,temp_line1,temp6,temp7};
assign line2={temp1,temp2,temp3,temp_line2};
assign display={line1,line2};

### Parts of the Verilog file: adder_lcd.v

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7)</td>
<td>lcd_disp.v: This is a generic Verilog module which generates the required timing and control signals to display the 32 characters on the LCD screen. You may re-use this module along with a design specific interface file (such as adder_lcd.v above) to configure your LCD for a particular design.</td>
</tr>
<tr>
<td>8)</td>
<td>counter.v: This module counts up by one on the positive edge of its own clock input port. It is used by the rotational knob interface to count up with every turn of the knob.</td>
</tr>
<tr>
<td>9)</td>
<td>push_butt.v: This module de-bounces the signal from a mechanical switch to generate a clean pulse which might be used as inputs to some parts of the design. In this lab, it is used to clean the bounces of the rotational knob push button, by sampling the output of the mechanical switch.</td>
</tr>
<tr>
<td>10)</td>
<td>sr.v: This module describes a set-reset flip-flop which is used to clean up the bounces from ROT_A and ROT_B of the rotational knob. You will study these techniques in further detail in the next lab.</td>
</tr>
<tr>
<td>11)</td>
<td>adder_test.v: This is the test bench file (also called testfixture file) for this design. A test bench is a Verilog file which contains input patterns to test the functionality of the hardware at various stages of the design cycle. For</td>
</tr>
</tbody>
</table>
simulation purposes, this test is generally performed on the top module in the circuit. Remember the test bench is not synthesized into any hardware; it is used for simulation purposes only. This file only shows up on the ‘Sources’ window when Simulation options are selected as we will see subsequently in this lab. Parts of the Verilog test bench for the adder is given below showing the input patterns that are being used to test the adder design.

```
initial
begin
  // Initialize Inputs
  //Begin of Simulation Period
  A = 0; B = 0;
  // Add Input Patterns here
  #2 A = 8'h12; B = 8'h13;
  #2 A = 8'h20; B = 8'h30;
  ...
  #2 A = 8'h11; B = 8'h10;
  #2 A = 8'h08; B = 8'h09;
  //End of Simulation Period
end
```

**Parts of the Verilog test bench file: adder_test.v**

g) Now we need instruct the ISE about the top module of our design. The top-most module in our design is in rot_intfc.v. To do this, go to sources window and find the file named rot_intfc.v. Now, right click on this file and click on ‘Set as Top Module’ as shown in Figure 5.8.

h) Now we need to check the syntax of our Verilog files. In the ‘Sources’ window, click on any Verilog file in the design. Now the ‘Process’ window appears showing various operations that need to be performed to download the design into the Xilinx FPGA. First of all, check the syntax of your code by clicking on ‘Check Syntax’ under ‘Synthesize’ process. An ✔ icon should appear next to Check Syntax once syntax checking is complete. If you get any compilation errors, please correct them before continuing. Check all your Verilog files for correct syntax before proceeding to the next step.
Figure 5.8: Setting Top Module of the Design

i) The design should be simulated by using the test patterns written in the test bench to validate the functionality of the Verilog Code. To perform simulation before synthesis, go to the ‘Source for’ tab on top of the ‘Sources’ window and select ‘Behavioral Simulation’ from the drop-down menu. You will see the test bench file ‘adder_test.v’ in the ‘Sources’ window. Click on the ‘adder_test.v’ file and go to the ‘Process’ window and click on ‘Xilinx ISE Simulator’ > ‘Simulate Behavioral Model’ as shown in Figure 5.9. This will launch the integrated Xilinx ISE Simulator, and show the simulation results in a black waveform window as shown in Figure 5.10. This window will show all the tested input patterns and output of our design at various instances in time. Notice that all signals in the module being tested shows up in the waveform window. For continuing on Synthesis and Implementation, go to the ‘Source for’ tab in the Sources window and select ‘Synthesis and Implementation’ from the drop down menu.
Figure 5.9: Selecting Behavioral Simulation

Figure 5.10: Waveform window of Xilinx ISE Simulator

j) Now we need to create and add the User Constraints File (.ucf) to the project. This contains the location of FPGA I/Os on the board. This file will be used to connect the design I/O signals (ports in the top level Verilog file) physically to the FPGA I/O pins through which we feed our data from the Spartan 3E board and display results to the LCD in the Spartan 3E board. To create a user constraints file, go to ‘User Constraints’ in the ‘Process’ window and click on ‘Edit Constraints (Text)’. Click ‘Yes’ on the pop up which appears asking for permission to create and add a constraint file automatically. For this lab, the constraints file necessary (adder.ucf) is given in the zipped folder along with the
Verilog files. You can copy and paste the content of this constraint file in the blank .ucf automatically generated in the above step and ‘Save’ it.

Alternately, you can also enter user constraints by using the Xilinx PACE editor. For designs to be done in other labs, a standard user constraints file for the Xilinx Spartan 3E starter board is given in Appendix B. You need to modify the entries in this file depending on your own requirement of I/O ports in the design.

![Figure 5.11: Choose Configuration Options](image)

k) Now you need to implement the complete design and program the FPGA. In the Process window, double click on ‘Generate Programming File’. This will run all the steps necessary to create the bit-stream that can be downloaded on to the board to program the FPGA. Running these processes may take several minutes; progress is indicated by the spinning icon and output to the console. When a process completes, an icon appears next to the Generate Programming File text. The steps that are run before the Programming File is created are: synthesis of the Verilog, mapping of the result to the FPGA hardware, placement of the mapped hardware, and routing of the placed hardware and generation of the .bit bit-stream file required to program the FPGA.
For designs in other labs, you may want to do these steps separately. All these steps can be done separately one at a time by clicking individually on the ‘Synthesize’, ‘Map’ and ‘Place and Route’ buttons in the ‘Process’ window. You can also simulate your design after each of these operations to check the functionality of your design at each step of the design cycle. Xilinx ISE provides facility for post-synthesis, post-map and post-place and route simulations. Each of these processes also have several properties that you can set by right clicking on it and clicking on ‘Properties’. These properties are generally used to optimize the design in a specific way i.e. to optimize timing or area or power.

![Figure 5.12: Device Programming Properties Window](image)

1) We now need to download the bit-stream to the FPGA on the Spartan 3E board. Turn on the power to the Spartan 3E starter Board. In Processes window, under the ‘Configure Target Device’ tab, double click on “Manage Configuration Project (iMPACT)”. This will open iMPACT (the tool which performs the bit-stream download to the FPGA, also referred to as device configuration). The above command will cause a pop-up to appear which asks you for a method of configuration. Choose ‘Configuration device using Boundary-Scan (JTAG)’ as shown in Figure 5.11 and click on ‘Finish’. After scanning the board, iMPACT finds three devices. The first, "XC3s500E", is the FPGA that we need to program.
Double Click on the appropriate bit-stream file (say ‘rot_intfc.bit’ in our case) with which you need to program the FPGA and click on ‘Bypass’ for the other two devices.

m) Now the “Device Programming Properties” window appears on the screen. Click on ‘Ok’.

![Figure 5.13: Programming the FPGA through Boundary Scan (JTAG)](image)

n) Select XC3S500E from the device list shown in Figure 5.13. Click on ‘Boundary Scan’ > ‘rot_intfc.bit’ (Assuming the .bit file of your design was named rot_intfc.bit) from the Sources window on the top left corner of your screen. Then click on ‘Program’ in the ‘Configuration Operations’ window to program the FPGA as shown in Figure 5.12. When the FPGA is programmed successfully, you should see a ‘Program Succeeded’ sign on the programming window else you will see a ‘Program Failed’ sign. If you encounter ‘Program Failed’ sign, then check whether you have done all the steps mentioned in the manual correctly.

If you have successfully programmed your design on to the FPGA, then you can now test whether the design functions correctly by proceeding on to the next step.

o) Use the rotational knob to enter the first operand starting from least significant nibble (4 bits) to the most significant nibble and press the knob to freeze the value. As you are entering data, you will see it on the LCD display. Once the first operand is set, the cursor automatically moves to the second operand. Now, enter the second operand nibble by nibble. On entering the last nibble of the second operand, the FPGA based adder adds the two operands and displays the result on
the second row of the LCD display. All the data displayed on the LCD is in hexadecimal format. An example, which adds ‘A5’ and ‘E6’ to produce ‘18B’ is shown in Figure 5.13 below.

If you have any more queries regarding the use of ISE, contact your TA. You should understand all the steps described above in detail. Please ask the TA if you don’t understand any specific terminology or steps. Please clarify all your doubts before you proceed. Alternatively, you can also use the ‘Help’ option on the Xilinx ISE Project navigator and click on ‘Tutorials’ for more detailed descriptions of various options on the Xilinx ISE.

5.3.3 Pre-Lab Deliverables
None

5.4 Lab Procedure
Complete Experiments listed below.

5.4.1 Experiment 1
Apart from the test patterns already in the test bench file (adder_test.v), add at least 5 more patterns to your test bench. Perform behavioral simulations of the adder block using the test bench provided as described in Section 5.3.2. Demonstrate the results to your TA.

5.4.2 Experiment 2
Now set rot_intfc.v as the top module of your design and implement the complete design (synthesize, map and Place & Route). Program the FPGA using the bit-stream file which is generated in the process. Demonstrate at least one addition operation to your TA from the patterns you have used in the test bench. Show that your simulation results above matches the output on the LCD Display.

5.5 Post-Lab Deliverables
a) Submit the waveforms (similar to Figure 5.10) for the Experiment 1. Attach the waveforms (in separate sheets) in your lab report.

b) Now instead of the structural Verilog net-list provided to you for the adder. Write a behavioral Verilog module for the 8 bit adder. Do behavioral simulation to validate the code. Use the test bench provided to you above for the same. Submit your Verilog file for the adder in your lab report.

c) Based on the standard .ucf file for the Spartan 3E starter Board (given in Appendix B), modify the design 1 specific .ucf file provided to you such that the reset push button is to the ‘West’ of the rotational knob (instead of the original ‘East’ position). Provide the modified design specific .ucf file in your lab report.
Lab 6: De-Bouncing, Counters & LCD Display Control

6.1 Introduction
The objective of this lab is to design an eight bit counter which takes a clock input signal from mechanical switches and counts up with every positive edge of the clock signal (i.e. On every push of a mechanical switch). In this lab, you will learn how to ‘clean’ a ‘noisy’ signal generated from a mechanical switch before using it in a digital circuit. You will also learn about configuring the LCD display for a particular design.

6.2 Background

6.2.1 Counter
Counter is a digital circuit which uses sequential logic to count clock events (either rising clock edges or falling clock edges). An up-counter counts up with every clock event where as a down-counter counts down with every clock event. A counter uses memory elements like flip-flops to keep track of the current count state and increments/decrements it with every clock pulse. Also, it has a ‘reset’ input which resets the current state of the counter to an initial state (Usually ‘zero’) so that it can again start counting from the initial state. The behavioral block of a counter is given in Verilog below.

In this lab, we will design an eight bit counter as shown in Figure 6.1. This counter can count to a maximum value of 255 and uses 8 flip-flops to store its current state.

```
//Behavioral block of a counter
always @ (posedge clk or posedge reset)
begin
    if (reset)
        count <= 0;
    else
        count <= count + 4'd1;
end
```

Behavioral block of a counter in Verilog

![Figure 6.1: Block Diagram of an eight bit counter](image)
6.2.2 Push Button
The Spartan 3E starter board has four push button switches (BTN_EAST, BTN_WEST, BTN_SOUTH, BTN_NORTH) surrounding the rotational knob on each side and one push button on the rotational knob (BTN_CENTER) as shown in Figure 6.2.

![Rotational Knob and Push Button location on the Spartan 3E Board](image_url)

**Figure 6.2: Rotational Knob and Push Button location on the Spartan 3E Board**

![Non-Debounced and De-Bounced Signals](image_url)

**Figure 6.3: Non-Debounced and De-Bounced Signals**
Push Button switches are mechanical and rely on mechanical contact to generate an electrical pulse. Every time a Push Button is pushed, an electrical chatter is generated as shown in Figure 6.3. This chatter consists of many short duration pulses. These pulses have a duration which is considerably shorter than the time for which the button was pressed. Hence, the pulses appear as a bouncing signal in that interval when the button is pressed. Generally, some de-bouncing circuitry needs to be used to get rid of these bounces and produce an electrical pulse waveform with the pulse width equal to the time period for which the signal was bouncing. This is called de-bouncing and is shown in Figure 6.3.

### 6.2.3 Rotational Knob

The rotational knob encoding circuitry is given in Figure 6.4. When the shaft is stationary, ROT_A and ROT_B stay connected to ground. Any rotational motion to the shaft opens the two switches (ROT_A and ROT_B) which cause them to be pulled to a ‘1’. Due to mechanical chatter on the switches, with each movement of the shaft ROT_A and ROT_B bounces as shown in Figure 6.5. When the shaft turns right, ROT_A bounces first and then ROT_B bounces whereas when the shaft turns left, ROT_B bounces first followed by ROT_A.

![Figure 6.4: Basic rotary shaft circuitry](image)

A pull up resistor in each input pin generates a ‘1’ for an open switch. See the generic UCF file given in Appendix B of this manual for specifying the pull up resistor.

Figure 6.5 shows the ROT_A and ROT_B output electrical signals generated by two right turn steps of the rotary shaft encoder. These bouncing signals cause erroneous interpretation of the shaft movement, so we need to de-bounce those two signals to generate precise pulses for each shaft movement as shown in the de-bounced signal (bottom-most waveform in Figure 6.5).
6.2.5 De-Bouncing Signals

Mechanical chatter in switches results in bouncing of electrical signals as shown in Figure 6.3 and Figure 6.5. De-Bouncing is a technique to generate steady pulses from these signals which can be used in digital circuits. Many techniques need to be used to de-bounce a signal, like Latch based de-bouncing, Sampling based de-bouncing, Low Pass filter based de-bouncing and software based de-bouncing. In this section, we will discuss two of these de-bouncing techniques.

6.2.5.1 S-R Latch Based De-Bouncing

This method of de-bouncing signal is generally used when there are two switches one for each edge of the clock pulse. The rotational knob is one such switch which employs two internal switches to generate pulses with each turn of the knob. So, the SR-Latch based de-bouncing technique can be employed to de-bounce the signal from the rotational knob to generate a digital pulse for each turn of the knob. In order to create a de-bounced clock signal in this case, you will use a very simple sequential circuit called the S-R (set-reset) latch.

Sequential circuits are characterized by the fact that their outputs depend not only on the current inputs, but on past inputs. This is different from combinational circuits whose outputs depend only on the current inputs. You will study sequential circuits in detail later in the semester, but for the purposes of this and future labs, you need to understand the function of the S-R latch. The S-R latch has two inputs and two outputs. One output is the complement of the other. The S input “sets” the output (to logic ‘1’) while the R “resets” the output to logic ‘0’. Note that, this is the kind of behavior we require for a de-bounced clock signal. See Figure 6.6 for a schematic and truth table for the S-R latch. For the rotational knob, connect the ROT_A signal to the ‘S’ input of the latch and ROT_B...
signal to the ‘R’ input of the latch. Therefore, the latch sets to ‘1’ when ROT_A bounces and holds on to that value because ROT_B is steady at ‘0’ during that time. This is followed by ROT_B bouncing which resets the latch to ‘0’. Again, the latch holds on to this value as ROT_A is steady at ‘0’ during this phase. Figure 6.5 shows the ROT_A, ROT_B signal along with the de-bounced signal from the output of the latch.

![SR Latch Schematic](image)

**Figure 6.6: SR Latch – Schematic and Truth Table**

### 6.2.5.2 Sampling Based De-Bouncing

This technique is generally employed when we have just one mechanical switch to read in our input pulse. A single push button switch is an example where this kind of technique is generally applied. In the Spartan 3E Starter board, if we want to use the push-buttons (BTN_EAST, BTN_WEST, BTN_NORTH, BTN_SOUTH, ROT_CENTER) independently to read in input pulses from the user, then we may use the sampling based technique to de-bounce the inputs of these signals.

In this technique, the bouncing signal from the mechanical switch is sampled using a high frequency clock. The default clock on the board which has a frequency of 50 MHz can be used for this purpose. Sampling is a method in which we read the bouncy signal at every positive edge of the clock pulse. The sampled data is monitored by the De-Bouncing unit. If the data changes frequently, the output of the De-Bouncing unit is set to ‘1’ where as if the data is steady for a time window ‘T’ then the output of the De-Bouncing unit is reset to ‘0’. Generally, for practical purposes a time window T=100ms is used. Figure 6.7 demonstrates the sampling technique described above. The behavioral description of the Verilog code which implements this technique is also provided below.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q(n)</th>
<th>Q(n+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q(n)</td>
<td>Q(n)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
6.2.6 Configuring the LCD Display

The LCD Display can be configured using the two Verilog modules ('lcd_disp', 'lcd_int') given in Figure 6.8. The 'lcd Disp' Verilog module written in the file 'lcd_disp.v' was provided to you in Lab #5. You can use it for configuring the display in other labs as well.
For every design, in addition to the ‘lcd Disp’ module you will need a design specific ‘lcd int’ module. You have to write this module separately for each design, depending on the format in which you need to configure your display (because data to be displayed at different locations varies from design to design). The ‘lcd int’ module takes in data from the other hardware in the design and outputs the entire data sequence required for configuring all the 32 characters in the display. The LCD display needs eight bit of data for each character to be displayed. The character that gets displayed is picked from the

**Figure 6.9: LCD Display Decoder Chart**

<table>
<thead>
<tr>
<th>Lower Data Nibble</th>
<th>Upper Data Nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x 0 0 0 0</td>
<td>0 0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>x x x x 0 0 0 1</td>
<td>! 1 A Q a e</td>
</tr>
<tr>
<td>x x x x 0 0 1 0</td>
<td>&quot; 2 B R br</td>
</tr>
<tr>
<td>x x x x 0 0 1 1</td>
<td># 3 C S s</td>
</tr>
<tr>
<td>x x x 0 1 0 0</td>
<td>$ 4 D T dt</td>
</tr>
<tr>
<td>x x x 0 1 0 1</td>
<td>% 5 E U eu</td>
</tr>
<tr>
<td>x x x 0 1 1 0</td>
<td>&amp; 6 F V v</td>
</tr>
<tr>
<td>x x x 0 1 1 1</td>
<td>' 7 G W w</td>
</tr>
<tr>
<td>x x x 1 0 0 0</td>
<td>( 8 H X h</td>
</tr>
<tr>
<td>x x x 1 0 0 1</td>
<td>) 9 I Y y</td>
</tr>
<tr>
<td>x x x 1 0 1 0</td>
<td>* : J Z j</td>
</tr>
<tr>
<td>x x x 1 0 1 1</td>
<td>+ : K [k</td>
</tr>
<tr>
<td>x x x 1 1 0 0</td>
<td>, &lt; L 半1</td>
</tr>
<tr>
<td>x x x 1 1 0 1</td>
<td>- = M ] m</td>
</tr>
<tr>
<td>x x x 1 1 1 0</td>
<td>. &gt; N ^ n</td>
</tr>
<tr>
<td>x x x 1 1 1 1</td>
<td>/ ? O _ o</td>
</tr>
</tbody>
</table>
chart given in Figure 6.9 below. The character to be displayed as a result of the eight bit into the LCD is chosen as described below. The vertical column is specified by the upper data nibble of the 8-bit data and the horizontal row of the table is specified by the lower nibble of the 8-bit data. The character corresponding to this particular column and row is picked to be displayed. Let’s say if you need to display the letter ‘A’, then the eight bit sequence required is ‘01000001’ in binary or ‘41’ in hexadecimal as seen from Figure 6.9. You can write your own Verilog module to decode hexadecimal values to this 8-bit sequence in a ‘Case’ statement as shown in the sample Verilog code given below.

/* Verilog Code for Decoding Hexa-Decimal to the display 8-bit sequence */
always @ (*)
begin
  case (in)
    4'b0000: out=8'h30;
    4'b0001: out=8'h31;
    .
    .
    .
    4'b1110: out=8'h45;
    4'b1111: out=8'h46;
  endcase
end

Therefore, the ‘lcd_int’ module outputs thirty-two eight-bit sequences for the thirty-two characters in the display (8 bits each) through the ‘data’ output pin. The ‘data’ output bus requires a width of 32 x 8 = 256 through which the entire bit sequence is passed on to the ‘lcdDisp’ module. The ‘data’ output of the ‘lcd_int’ module is generated by concatenating the 32 eight bit sequences starting from the most significant bit to the least significant bit. The ‘lcdDisp’ module generates all the necessary timing signals and appropriately multiplexes the input from the ‘data’ bus to configure the LCD display.

6.3 Pre-Lab

6.3.1 Design 1
Using the LCD control technique provided in section 6.2.6 above, print ‘ALPHA123’ and ‘321AHPLA’ on the LCD screen as shown in Figure 6.11. Write your LCD interface module in Verilog and connect it to the ‘lcdDisp’ module given to you. Implement and program your design on the Spartan 3E Board using Xilinx ISE.
6.3.2 Design 2
Design an eight bit counter with a non-debounced clock from the rotational knob Push-Button. Configure the LCD to display the output of the counter.

```
data = {char1, char2, char3 ... char31, char32};
```

In the example above,
char1= '00110001'
char2= '01000001'
char3, char4 ... char30= '00100000'
char31= '00110001'
char32= '01000001'

Figure 6.10: Example showing the ‘data’ output from the ‘lcd_int’ module

6.3.3 Design 3
Design two eight bit counters with the non-debounced signal from the rotational knob on the board. One counter should take the clock input from ROT_A and the other counter should take the clock input from ROT_B. Configure the display to show both the counter outputs. Counter 1 output should be on the first line of the LCD display and Counter 2 output should be displayed on the second line of the LCD display.

Figure 6.11: Design 1 LCD display configuration

6.3.4 Design 4
Design an eight bit up-counter (only one counter) with de-bounced clock from the rotational knob. Use the SR latch based method of de-bouncing. Configure the LCD to display the output of the counter.
6.3.5 Design 5
Design an eight bit up-counter which has a de-bounced clock from the Push-Button on the rotational knob. Use the sampling based method of de-bouncing. Configure the LCD to display the output of the counter.

6.3.6 Pre-Lab Deliverables

a) Submit the Verilog Interface module for Design 1 which would connect to the ‘lcd_disp’ module to configure the LCD as shown in Figure 6.11.
b) Submit the Verilog files for all the modules to be used in Design 2 and Design 3. Submit the User Constrained File for each design as well.
c) Submit the Verilog files which implements the de-bouncing in Design 4 and Design 5. Also provide the User Constrained File in your pre-lab report.

6.4 Lab Procedure
Perform experiments from 1-5 described below.

6.4.1 Experiment 1
Implement design 1 on the Spartan 3E Starter Board. Demonstrate the results to your TA.

6.4.2 Experiment 2
Implement design 2 on the Spartan 3E Starter Board. Demonstrate the results to your TA. Now press the push button 20 times and read the incremented value on the display. Calculate the average number of bounces with every ‘push’ event.

6.4.3 Experiment 3
Implement design 3 on the Spartan 3E Starter Board. Demonstrate the results to your TA. Now turn the rotational knob 20 times and read the incremented value of both the counters on the display. Calculate the average number of bounces with every ‘rotate’ event for Counter 1 and Counter 2.

6.4.4 Experiment 4
Implement design 4 on the Spartan 3E Starter Board. Demonstrate the results to your TA.

6.4.5 Experiment 5
Implement design 5 on the Spartan 3E Starter Board. Demonstrate the results to your TA.

6.5 Post-Lab Deliverables

a) Report the average number of Bounces you calculated in Experiments 2 and 3.
b) Draw the schematic of a D-latch using only a 2:1 Multiplexer? Design a master-slave D-Flip flop using D-Latch as a functional block?
c) Design a 2-bit synchronous counter using D-flip-flops. Draw the gate level schematic of the design. You may use flip-flops as functional blocks in your schematic.

d) If the clock frequency of the system in Design 4 was:
   i) 100 Hz ii) 1Khz iii) 5 KHz
Which among these clock frequencies would result in design failure? Why? (Hint: Use the information which was given in Section 6.2.5.2 (Sampling Based De-Bouncing Technique)).
Lab 7: Carry Look Ahead and Carry Save Adders

7.1 Introduction
In this lab, you will design two types of adders called ‘Carry look-ahead Adder’ and ‘Carry Save Adder’ respectively. We will also configure the rotational knob to read in the operand of each adder from the user, one of the push buttons for design ‘reset’ and also the LCD to display the input data as well as the results in a specific format.

7.2 Background

7.2.1 Carry Look-Ahead Adder
Earlier in Lab #5 we used ripple carry adders to add two eight bit numbers. The main disadvantage of the ripple-carry adder is that when adding numbers with large word lengths, it can get very slow which brings down the maximum speed of operation. For fast applications, ripple carry adders cannot be used. In these cases, faster variety of adders is used. These adders calculate the carry signal in advance, depending on the values of the input signals. They are based on the fact that a carry signal $C_{i+1}$ will be generated from the $i^{th}$ bit position in either of two cases:

1) When both bits being added $A_i$ and $B_i$ are equal to ‘1’.
2) When one of the two bits is 1 (Either $A_i=1$ or $B_i=1$) and the carry-in (carry of the previous stage) is 1.

Thus, one can write the logic expression for the carry-out from a stage as:

$$C_{OUT} = C_{i+1} = A_i.B_i + (A_i \oplus B_i).C_i$$

Where $C_{i+1}$ is the carry out from the current stage to the next stage, $(A_i, B_i)$ are inputs of the $i^{th}$ stage, $C_i$ is the carry output of the $i^{th}$ stage and $\oplus$ can be an ‘Or’ operation or ‘Xor’ operation. One can write this expression also, as:

$$C_{i+1} = G_i + P_i.C_i$$

In which

$$G_i = A_i.B_i$$

is called the Generate term and

$$P_i = (A_i \oplus B_i)$$

is called the Propagate term.

Let us assume that the delay through an AND gate is ‘D’ and the delay through an XOR gate is 2D. Notice that the Propagate and Generate terms expressions only depend on the input bits and thus will be valid after 2D and D delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its final value. Let us apply this idea to a 4-bit adder to make it clearer.

$$C_1 = G_0 + P_0.C_0$$

$$C_2 = G_1 + P_1.C_1 = G_1 + P_1.G_0 + P_1.P_0.C_0$$
\[ C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0 \]  \hspace{1cm} (7)

Notice that the carry-out bit, \( C_4 \), of the last stage will be available after 4D delays (2D delays to calculate the Propagate signal and 2D delays as a result of the ‘AND’ and ‘OR’ gate outputs. The Sum signal can be calculated as follows:
\[ S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \]  \hspace{1cm} (9)
The Sum bit will thus be available after two additional gate delays (due to the XOR gate) (i.e. a total of 6D delays after the input signals \( A_i \) and \( B_i \) have been applied). The advantage is that these delays will be the same independent of the number of bits one needs to add, in contrast to the ripple carry adder.

The carry-lookahead adder can be broken up in two modules: (1) the Partial Full Adder, PFA, which generates \( S_i, P_i \) and \( G_i \) (2) the Carry Look-ahead Logic, which generates the carry-out bits. The 4-bit adder can then be built by using 4 PFAs and the Carry Look-ahead logic block as shown in Figure 7.1.

**Figure 7.1: Four-Bit Carry Look-Ahead Block**

The disadvantage of the carry look-ahead adder is that the carry logic gets complicated for more than 4 bits. For that reason, larger carry-look-ahead adders (for adders greater than 4 bits) are usually implemented as 4-bit modules which are used in a hierarchical structure to realize adders that have multiples of 4 bits. Figure 7.2 shows the block diagram for a 16-bit CLA adder which you will design in this lab. The circuit makes use...
of the same Carry Look-Ahead Logic block as the one used in the 4-bit adder of Figure 7.1. Notice that each 4-bit adder produces a group Propagate and Generate Signal, which is used by the CLA Logic block. The group Propagate $P_G$ of a 4-bit adder will have the following expressions,

$$P_G = P_3P_2P_1P_0$$  \hspace{1cm} (10)

$$G_G = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$  \hspace{1cm} (11)

The group Propagate $P_G$ and Generate $G_G$ one or two additional delays than the $P_i$ and $G_i$ signals, respectively.

![Figure 7.2: 16-bit CLA Adder using 4-bit CLA modules](image)

7.2.2 Carry-Save Adder

There are many cases where it is desired to add more than two numbers together. The straightforward way of adding together $m$ numbers (all $n$ bits wide) is to add the first two, then add that sum to the next, and so on. This requires a total of $m - 1$ additions, for a total gate delay of the order of $m \log n$ or $O(m \log n)$, assuming look-ahead carry adders.

Instead, a tree of adders can be formed, taking only $O(\log m \cdot \log n)$ gate delays. Using carry save addition, the delay can be reduced further still. The idea is to take 3 numbers that we want to add together, $X + Y + Z$, and convert it into 2 numbers $C + S$ such that $X + Y + Z = C + S$, and do this in $O(1)$ time. The reason why addition can’t be performed in $O(1)$ time is because the carry information must be propagated. In carry save addition, we refrain from directly passing on the carry information until the very last step. We will first illustrate the general concept with a binary two 5-bit operands as shown in Figure 7.3.

To add three numbers by hand, we typically align the three operands, and then proceed column by column in the same fashion that we perform addition with two numbers. The three digits in a row are added, and any overflow goes into the next column. In this case,
when there is some non-zero carry, we are really adding four digits (the digits of x, y and z, plus the carry). The carry save approach breaks this process down into two steps. The first is to compute the sum ignoring any carries and separately, we can compute the carry on a column by column basis as shown in the top-row of Figure 7.3 below. Now, the same ‘Sum’ can be computed by adding the ‘S’ and ‘C’ in the final stage of the addition. The important point is that c and s can be computed independently, and furthermore, each ci (and si) can be computed independently from all of the other c’s (and s’s). This achieves our original goal of converting three numbers that we wish to add into two numbers that add up to the same sum, and in O(1) time.

\[
\begin{array}{c}
X: 10011 \\
Y: +11001 \\
Z: +01011 \\
S: 00001
\end{array}
\quad
\begin{array}{c}
X: 10011 \\
Y: +11001 \\
Z: +01011 \\
C: 11011 \\
S: 00001 \\
C: 11011 \\
Sum: 110111
\end{array}
\]

**Figure 7.3: Carry-Save Addition Process of three 5-bit operands**

![Diagram of carry save adder block]  

**Figure 7.4: The carry save adder block is the same circuit as the full adder**
The carry-save adder structure can be implemented using Full Adders. A full adder, but with some of the signals renamed as shown in Figure 7.4 can be used to compute ‘C’ and ‘S’ in a carry-save adder. A carry save adder simply is a full adder with the ‘C_{in}’ input renamed to ‘Z_i’, the ‘C_{out}’ output renamed to ‘C_i’. Figure 7.5 shows how sixteen carry save adders can be arranged to add three 16 bit numbers X, Y and Z into two numbers ‘C’ and ‘S’. Note that the CSA block in bit position zero generates ‘C_0’ which gets added to ‘S_1’. Also, the first carry save adders output ‘S_0’ is the least significant bit of the ‘Sum’. Note that all of the CSA blocks are independent, thus the entire circuit takes only O(1) time. To get the final sum, we still need a Carry Look-Ahead Adder in the last stage, which will cost us O (lg n) delay. The gate delay to add three n-bit numbers is thus the nearly the same as adding only two n-bit numbers especially when ‘n’ is larger.

Figure 7.5: Three operand 16-Bit Carry-Save Adder Block Diagram

7.3 Pre-Lab

7.3.1 Design 1
Implement a 4-bit carry look-ahead adder as shown in Figure 7.1. Test and verify your design of this four-bit Carry Look-Ahead Adder. Now use four of these four bit carry look-ahead adders to implement a 16-bit adder as shown in Figure 7.2. The adder should take two 16-bit inputs (A, B) and generate a 17 bit output (SUM) as shown in the adder module of the design given in Figure 7.3. The inputs should be provided to the adder in
the same way as the ripple carry adder of Lab # 5. ‘A’ and ‘B’ should be entered nibble by nibble using the rotational knob. The LCD display should be configured to show the inputs ‘A’, ‘B’ and also the output ‘SUM’ as shown in Figure 7.4.

![Figure 7.6: Adder Module of the 16-bit CLA](image)

![Figure 7.7: LCD display configuration for Design 1](image)

### 7.3.2 Design 2

Implement a carry save adder which has three 16-bit operands as its input (shown in Figure 7.5). The adder should take three 16-bit inputs (A, B, C) and generate the sum of the three numbers as output called ‘SUM’ which is of 18 bits as shown in Figure 7.8. The inputs should be provided to the adder in the same way as the ripple carry adder of Lab # 5. The inputs ‘A’, ‘B’ and ‘C’ should be entered nibble by nibble using the rotational knob. The LCD display should be configured to show the inputs ‘A’, ‘B’, ‘C’ and also the output ‘SUM’ as shown in Figure 7.9.

![Figure 7.8: Adder Module of the 16-bit CSA Design](image)
7.3.3 Pre-Lab Deliverables
a) Draw a block diagram of the two operand 16-bit carry look-ahead adder and three operand 16-bit carry-save adders. You should name the blocks and sub-blocks with exactly the same name as you have used in your Verilog Files in the design.

b) Submit your Verilog files for all the modules used in Design 1 and Design 2. Also submit the User Constrained File for Design 1 and Design 2. Label all your Verilog files cleanly with comments placed on Top of the file.

7.4 Lab Procedure
Perform experiments from 1 and 2 described below.

7.4.1 Experiment 1
Implement design 1 above and demonstrate the results to your TA.

7.4.2 Experiment 2
Implement design 2 above and demonstrate the results to your TA.

7.5 Post-Lab Deliverables
a) Attach the “Synthesis Report” from Xilinx ISE Software for the Carry-Look-Ahead Adder Design. Show the critical path from input to output of your Carry-Look-Ahead Adder. For this, set the adder-module as the top CLA-module and then run synthesis on Xilinx ISE.

b) Similarly, Attach the “Synthesis Report” from Xilinx ISE Software for Carry-Save Adder Design. Show the critical path from input to output of your Carry-Save Adder. For this, set the CSA-module as the top module and then run synthesis.
Lab 8: 16-bit ALU with eight operations

8.1 Introduction
In this lab, you will design a 16-bit Arithmetic Logic Unit (ALU) using Verilog and implement your design on the Xilinx Spartan 3E starter Board. The ALU will have eight operations viz. Bitwise AND, Multiplexer-Based Shift, Subtract, Add, Multiply, Compare-Equal, Compare-Greater, Multiply and Accumulate (MAC).

8.2 Background

8.2.1 Signed Integer Arithmetic
In the section 4.2.1, we mentioned that a convenient property of the 2’s complement representation is that we can add complementary pairs of numbers to obtain zero. What about any two pairs of numbers in general? The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers. If ‘n’ bits are used to represent signed numbers in 2’s complement scheme, then the result must be in the range $-2^{n-1}$ to $2^{n-1}-1$. If the result does not fit in this range, then we say that arithmetic overflow has occurred. To ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow. The key to determining whether overflow occurs is to detect the values at the carry-out from the MSB position, called $C_{n-1}$ and from the sign-bit position, called $C_n$. It can be proved that:

$$V = C_{n-1} \oplus C_n$$

Where V is overflow and $C_{n-1}$ is the carry-out from the MSB position and $C_n$ is the carry-out from the sign-bit position (Fig. 4-1). When the numbers have opposite signs, there is no overflow ($V=0$). But if both numbers have the same sign, based on the magnitude of added numbers maybe overflow occurs which can be detected from the magnitude of V.

In Verilog, signed integer Arithmetic in inferred automatically in the synthesis process by using the ‘Signed’ keyword while declaring ‘reg’ and ‘wire’ data-types. So when writing the code in Verilog for this ALU, use the ‘signed’ keyword while declaring the data-types needed for this lab.

8.2.2 Details of Operations on the ALU
The ALU in this lab should be able to perform eight operations. The ALU will have 3 control signals (C2, C1, and C0) which will decide the exact operation that needs to be performed. The table below shows which operation will be performed for a particular value of C2, C1 and C0. {C2, C1, and C0} shown in the table means the 3-bit binary word formed by concatenating C2, C1, and C0.
The ALU should be able to read in two operands (A [15:0], B [15:0]) and the control signals (C2, C1, and C0) from the user, perform one of the above operations based on the values of C2, C1, C0 and output a 32 bit result Y [31:0]. Here, the operands ‘A’ and ‘B’ are signed numbers which can take values from $-2^{15}$ to $2^{15}-1$.

### 8.2.2.1 Bit-Wise AND
In this operation, operands ‘A’ and ‘B’ are bit-wise ANDed together to give output ‘Y’. For example, Y [0] is the bit-wise ‘AND’ result of A [0] and B [0], Y [1] is the bit-wise ‘AND’ result of A [1] and B [1] etc.

![Figure 8.1: Bit-wise AND Operation](image)

### 8.2.2.2 Addition
In this operation, the output ‘Y’ is the sum of the operands ‘A’, ‘B’. Figure 8.2 shows the block diagram of the adder with its input/output configuration.

![Figure 8.2: Block diagram of ADD Operation](image)

### 8.2.2.3 Subtraction
In this operation, ‘Y’ is the output on subtracting the signed integer ‘B’ from another signed integer ‘A’. Figure 83 shows the block diagram of the subtractor with its input/output configuration.
8.2.2.4 Compare-Equal

In this operation, output ‘Y’ is ‘0’ if the operand ‘A’ is not equal to the operand ‘B’ but output ‘Y’ is ‘1’ if ‘A’ is equal to ‘B’. Figure 8.4 shows the block diagram of the Compare-Equal to block with its input/output configuration.

![Figure 8.3: Block Diagram of Subtract Operation](image)

8.2.2.5 Compare-Greater

In this operation, output ‘Y’ is ‘0’ if the operand ‘A’ is less than or equal to the operand ‘B’ but output ‘Y’ is ‘1’ if ‘A’ is greater than ‘B’. Figure 8.5 shows the block diagram of the greater than comparison block with its input/output configuration.

![Figure 8.4: Block Diagram of Compare-Equal Operation](image)

8.2.2.6 Multiplication

When this operation is selected, the output ‘Y’ is equal to the product of ‘A’ and ‘B’. Figure 8.6 shows the block diagram of the 16-bit multiplier with its input/output configuration.

![Figure 8.5: Block Diagram of Compare-Greater Operation](image)

![Figure 8.6: Block Diagram of Multiplication Operation](image)
8.2.2.7 Multiply and Accumulate
When this operation is selected, operands ‘A’ and ‘B’ are multiplied together and added to the existing contents of ‘Y’ shifted by one place to the right.

\[ Y(n+1) = Y(n) + A(n) \times B(n) \]

Here, \( Y(n) \) is the current value in the output register ‘Y’ after ‘n’ MAC cycles and \( Y(n+1) \) is the value stored in the output register after ‘n+1’ MAC cycles. \( A(n) \) and \( B(n) \) are the operand values input to the ALU in the ‘n’th cycle.

Figure 8.7: Block Diagram of Multiply and Accumulate Operation

8.2.2.8 Shift Operation
When the ‘Shift’ operation is selected, operand ‘A’ should be shifted based on the value in ‘B’. The binary value in ‘B’ decides the direction, magnitude and type of shift of ‘A’. The details of the shift operation are given in Table 8.2. The top-level block diagram of the shift operation can be seen in Figure 8.9.

**Logical Shift:** A logical shift is a shift operator that shifts all the bits of its operand. A logical shift does not preserve a number's sign bit. Every bit in the operand is simply moved a given number of bit positions, and the vacant bit-positions are filled in, generally with zeros. A logical shift is often used when its operand is being treated as a sequence of bits rather than as a number. Logical shifts can be useful as efficient ways of performing multiplication or division of unsigned integers by powers of two. Shifting left by \( n \) bits on a signed or unsigned binary number has the effect of multiplying it by \( 2^n \). Shifting right by \( n \) bits on an unsigned binary number has the effect of dividing it by \( 2^n \) (rounding towards 0). The logical shift operation by one bit position on an eight bit operand is pictorially shown in Figure 8.8.
<table>
<thead>
<tr>
<th>Range of Bits of ‘B’</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>B [7:0]</td>
<td>Magnitude of Shift of ‘A’</td>
</tr>
<tr>
<td></td>
<td>(Maximum Magnitude of Shift = 32)</td>
</tr>
<tr>
<td>B [11:8]</td>
<td>Left Shift or Right Shift</td>
</tr>
<tr>
<td></td>
<td>If (B[11:8]=0) then Left Shift</td>
</tr>
<tr>
<td></td>
<td>Else Right Shift</td>
</tr>
<tr>
<td>B [15:12]</td>
<td>Arithmetic or Logical Shift</td>
</tr>
<tr>
<td></td>
<td>If (B[15:12]=0) then Arithmetic Shift</td>
</tr>
<tr>
<td></td>
<td>Else Logical Shift</td>
</tr>
</tbody>
</table>

Table 8.2: Significance of bits in ‘B’ operand

**Arithmetic Shift:** An arithmetic shift is a shift operator, sometimes known as a signed shift. For binary numbers it is a bitwise operation that shifts all of the bits of its operand; every bit in the operand is simply moved a given number of bit positions, and the vacant bit positions are filled in. Instead of being filled with all 0s, as in logical shift, when shifting to the right, the leftmost bit (usually the sign bit in signed integer representations) is replicated to fill in all the vacant positions. Arithmetic shifts can be useful as efficient ways performing multiplication or division of signed integers by powers of two. Shifting left by $n$ bits on a signed or unsigned binary number has the effect of multiplying it by $2^n$. Shifting right by $n$ bits on a two's complement signed binary number has the effect of dividing it by $2^n$, but it always rounds down (towards negative infinity). The arithmetic shift operation by one bit position on an eight bit operand is pictorially shown in Figure 8.8.

![Logical Right Shift](image1)

![Logical Left Shift](image2)

![Arithmetic Right Shift](image3)

![Arithmetic Left Shift](image4)

Figure 8.8: Pictorial Representation of Eight Bit Shift Operations
8.3 Pre-Lab

8.3.1 Design 1
Implement the ALU, given in section 8.2.2. It should have the ability to perform all the operations that is given in Sections 8.2.2.1 to 8.2.2.8. The block diagram of the ALU is given in Figure 8.10 with all the required input and output signals.

The control signals can be provided to the ALU from the On/Off switches provided on the Spartan 3E starter Board. The operands ‘A’ and ‘B’ can be read in using the rotational knob exactly in the same manner as in earlier labs. The push button on the rotational knob can be used to freeze the exact values of the operands. Another push button from the board should be used for ‘Reset’ so that you can reset all the values begin fed to the ALU before performing a new operation. You are also required to write a Verilog module which would interface the output of your ALU to the LCD display in the manner shown in Figure 8.11. You may use the ‘lcd_disp’ module provided to you in Lab # 5 along with your interface module to display your data in the format given in Figure 8.11.
The ‘Selected Operation’ section of the LCD should display in accordance to the following table:

<table>
<thead>
<tr>
<th>{C2,C1,C0}</th>
<th>‘Selected Operation Display’</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND</td>
</tr>
<tr>
<td>001</td>
<td>ADD</td>
</tr>
<tr>
<td>010</td>
<td>SUB</td>
</tr>
<tr>
<td>011</td>
<td>EQL</td>
</tr>
<tr>
<td>100</td>
<td>GRT</td>
</tr>
<tr>
<td>101</td>
<td>MUL</td>
</tr>
<tr>
<td>110</td>
<td>MAC</td>
</tr>
<tr>
<td>111</td>
<td>SHT</td>
</tr>
</tbody>
</table>

8.3.3 Pre-Lab Deliverables

a) Submit your Verilog files for all the modules used in Design 1. Also submit the User Constrained File for Design 1. Label all your Verilog files cleanly with comments placed on Top of the file.

8.4 Lab Procedure

Perform experiments 1 described below.

8.4.1 Experiment 1

Implement design 1 given above on the Xilinx Spartan 3E starter Board. Demonstrate the results to your TA.
8.5 Post-Lab Deliverables

a) In the circuit given in Figure 8.13 which has two inverters and one Ex-Or gate, the inverters have delay of T1 and T2 respectively. ‘IN’ is a clock signal with 50% duty cycle and period T. It is given that T1+T2 is less than T/2.

(i) What is the functionality of the circuit shown above?
(ii) Derive the duty cycle of output waveform?
(iii) What is the condition to get 50% duty cycle at the output also?

![Figure 8.13: Circuit for Question No 8.5 (a)](image)

b) Give the circuit that adds two BCD numbers and gives out a BCD number? The input BCD numbers should be of two digits and the output BCD number should be of 3 digits. Explain the algorithm for the addition of BCD numbers.

c) Draw the schematic of an Adder/Subtractor unit using 4-bit binary adder and some external gates, which gives out A+B if C=0 and A-B if C=1 as in the ALU design of Lab # 4. Also provide an indicator for checking the overflow? Use the above designed circuit as block box and give a scheme for finding the absolute value of a 4-bit number? Draw all the input/output signals clearly in your design schematic.
Lab 9: Design of a Traffic Light Controller

9.1 Introduction
In this lab you will learn about state machines. You will use state machines to design a Traffic Light Controller which takes care of traffic on the main highway and a small farm road going across it. The implementation should mimic the specifications precisely.

9.2 Background

9.2.1 Generating Timing Delays
In digital circuits, mostly counters are used to generate precise timing delays. Counters being sequential machines, use a clock signal which oscillates with a definite frequency. Counters count all the clock events (positive edge or negative edge) that occur. So, if the counter counts up to a value ‘n’ then ‘n’ clock events should have occurred at the input of the counter. Hence, the timing delay of ‘n’ clock events with the clock frequency of ‘f’ is \( \frac{n}{f} \).

The system clock of the Spartan 3E starter Board is 50 MHz. To generate a time delay of ‘t’ with this clock fed to the counter, it needs to count up to a value which is equal to ‘t’ multiplied by ‘f’.

9.2.2 Notion of States in Sequential Machines
A state machine is a type of sequential circuit structure which can allow you to create more elaborate types of digital systems.

A state machine consists of:
1) memory elements (e.g. D flip-flops) to store the current state
2) combinational logic to compute the next state
3) combinational logic to compute the output

There are two classes of state machines: Mealy State Machines and Moore State Machines.
1) The Mealy machine uses both the current state and the current inputs to form the output.
2) The Moore machine only uses the current state to form the output.

Figure 9.1 and Figure 9.2 illustrates this difference. These figures show the basic elements of a state machine:
1) the inputs (e.g. switches)
2) the memory elements (“D flip-flops”) that store the current state (MEM)
3) the combinational logic (“Next State Logic”) that computes the next state (CL-2)
4) the combinational logic (“Output Logic”) that computes the output (CL-1)
With a Mealy machine, a connection exists between the “Inputs” and both the “Next State Logic” and the “Output Logic”. With a Moore machine, a connection only exists between the “Inputs” and the “Next State Logic”.

**Figure 9.1: Block Diagram of Mealy’s Finite State Machine**

**Figure 9.2: Block Diagram of Moore’s Finite State Machine**

In order to design a state machine, you must:

1) determine the “states” of the machine (if there are \( n \) states, then you will need \( \log_2(n) \) memory elements for a binary encoding scheme)
2) design the “Next State Logic” block (CL-2 in the Figures 9.1 and 9.2)
3) design the “Output Logic” block (CL-1 in the Figures 9.1 and 9.2)

### 9.2.3 Traffic Light Controller

In this lab, you will design a traffic light controller (TLC) for the highway-farm road crossing that’s given in Figure 9.3. The TLC should control the traffic lights for the highway as well as for the farm-road. The traffic lights at the opposite end of highways are the same. Therefore, you don’t need to display both of them. Similarly, the farm road traffic lights at opposite ends will be the same for which we can just display one of the
two. In this crossing, the highway has greater priority than the farm road owing to the much greater traffic it caters to. The TLC should take care to allow greater amount of the

![Figure 9.3: Highway and Farm Road Crossing](image)

<table>
<thead>
<tr>
<th>PHASE 1</th>
<th>PHASE 2</th>
<th>PHASE 3</th>
<th>PHASE 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20s</td>
<td>3s</td>
<td>10s</td>
<td>3s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time-Line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHASE 1</td>
</tr>
<tr>
<td>PHASE 2</td>
</tr>
<tr>
<td>PHASE 3</td>
</tr>
<tr>
<td>PHASE 4</td>
</tr>
<tr>
<td>25s</td>
</tr>
<tr>
<td>3s</td>
</tr>
<tr>
<td>5s</td>
</tr>
<tr>
<td>3s</td>
</tr>
</tbody>
</table>

Time-Line for Normal Mode Operation

**Figure 9.4: Figure showing the timeline of the traffic signals in Normal Mode**
traffic to flow through via the Highway so that congestion at the crossing is avoided. The time-line of the traffic signals at the highway and the farm road is given in Figure 9.4. The meaning of each PHASE in the timeline is described in the Table 9.1. Part1 in the table refers to the part of the LCD showing the status of the highway traffic light. Part2 refers to the part of the LCD showing the status of the Farm-Road traffic light. Part3 refers to the part showing the status of the Highway Pedestrian Crossing and Part4 refers to the part showing the status of the Farm-Road Pedestrian Crossing. There are two modes of operation of the Traffic Light: Normal mode of operation and the Night mode of operation. In the normal mode of operation, if there are pedestrians waiting to cross the highway or if there are cars waiting in the farm road to cross the highway then the traffic lights should operate according to Time-Line1 otherwise the traffic lights should operate according to Time-Line2. When there are no cars or pedestrians waiting to cross the highway, the TLC stays in Phase1 for a longer time as seen in Figure 9.4. In contrast to this, in the night mode of operation TLC is always in Phase1. TLC goes through the cycle (Phase1-Phase2-Phase3-Phase4-Phase1) either when there is a pedestrian waiting to cross the highway or when there is a car on the farm road waiting to cross the highway. Figure 9.5 shows the time line for the night-mode. The cross-event in Figure 9.5 shows the time point at which either a pedestrian requests to cross the highway or the sensor on the farm road detects a car waiting to cross the highway.

**Table 9.1: Relationship between PHASE and Traffic Light Signals**

<table>
<thead>
<tr>
<th>PHASE NO.</th>
<th>PART1</th>
<th>PART2</th>
<th>PART3</th>
<th>PART4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHASE 1</td>
<td>GREEN</td>
<td>RED</td>
<td>STOP</td>
<td>Countdown Mode</td>
</tr>
<tr>
<td>PHASE 2</td>
<td>YELLOW</td>
<td>YELLOW</td>
<td>STOP</td>
<td>STOP</td>
</tr>
<tr>
<td>PHASE 3</td>
<td>RED</td>
<td>GREEN</td>
<td>Countdown Mode</td>
<td>STOP</td>
</tr>
<tr>
<td>PHASE 4</td>
<td>YELLOW</td>
<td>YELLOW</td>
<td>STOP</td>
<td>STOP</td>
</tr>
</tbody>
</table>

**Figure 9.5: Figure showing the timeline of the traffic signals in Night Mode**

Cross Event

TLC always stays in this Phase

Time Line for Night Mode of Operation
9.3 Pre-Lab

9.3.1 Design 1
Implement the traffic light controller discussed in Section 9.2.3 above. The top-level block diagram of the TLC is given in Figure 9.6. Your design should have five input pins and the required number of output pins to connect it to the LCD. You can use the ‘lcd_disp’ Verilog module provided to you in Lab# 5 to configure your display for the design. The LCD display configuration for this design is given in Figure 9.7. In order to print your signals in this specific format, you have to write a LCD interface module which connects to the ‘lcd_disp’ module as you have already done in earlier labs.
Here is a brief description of the input pins of the TLC:

1) Clk: This is the global clock input to the traffic light controller.
2) Reset: This pin is generally taken from one of the push buttons and is used to reset to initialize all the sequential parts of the design.
3) Ped_H: This is the pedestrian switch (push-button) at the highway crossing. Pressing this push-button signifies a crossing request by the pedestrian to cross the highway.
4) Ped_F: This is the pedestrian switch (push-button) at the farm-road crossing. Pressing this push-button signifies a crossing request by the pedestrian to cross the farm-road. This has no considerable impact on the design as farm has lesser priority has hence, the TLC by default provides for ample crossing times for the pedestrian who wants to cross the farm road.
5) No_Car: This can be taken from an on/off switch (such as SW0) on the Spartan 3E started Board. The switch being ‘On’ will mean that the sensor detects a car on the farm road waiting to cross the highway. When the switch is ‘Off’, it means that the sensor doesn’t detect anything waiting on the farm road.

9.3.3 Pre-Lab Deliverables

a) Which machine (Mealy’s or Moore’s) are you using to design the TLC? Draw the bubble diagram showing all the states of the design? You can choose to a draw a single state diagram for all the three modes or give three different state diagrams one for each mode.

b) Give the state table for the design based on your state diagram given in part (a). Encode the states using binary encoding scheme. The state table should show the transition of states and output for all the combination possible in the TLC.

c) Submit your Verilog files for all the modules used in Design 1. Also submit the User Constrained File for Design 1. Label all your Verilog files cleanly with comments placed on Top of the file.

9.4 Lab Procedure
Perform experiment 1 described below.

9.4.1 Experiment 1
Implement design 1 above and demonstrate the results to your TA.

9.5 Post-Lab Deliverables

a) Suppose a state machine has k states. Find an exact expression for the number of D flip-flops (memory elements) required to hold the state in a:

1) Binary Encoding Scheme
2) One hot Encoding Scheme
3) Johnson’s Encoding Scheme
4) Gray Encoding Scheme
b) Draw two state diagrams (using both Mealy’s and Moore’s machine) to output a "1" for one cycle if the sequence "0110" shows up and the leading 0s cannot be used in more than one sequence. Draw the state tables for both the state diagram. Implement your state table using D flip flops and minimum number of logic gates.

c) Among Mealy and Moore state machines, which one do you think is better and why?