Department of Electrical and Computer Engineering

ECE 124 Lab Manual

Digital Circuits and Systems

Spring 2013

This document is under revision. Your feedback is valuable! Please send your comments to Rasoul Keshavarzi: rkeshava@uwaterloo.ca

Table of Contents

1 Introduction

The Lab Experiments are done in groups of two. Find a lab partner as soon as possible. Each laboratory experiment has several parts:

1. A prelab that must be done before coming to the lab. This will include material to read, and a circuit to design.
2. A three-hour laboratory session during which help is available, progress is demonstrated, and debugging is done.
3. A final report, due is 4:30pm, one day after your demo, regarding the guidelines in the lab manual. To minimize the VHDL and simulation print-outs use 2-up, double sided, and landscape orientation. Late lab reports will lose mark 10 % per day.
4. Absolutely no food or drink in the laboratories. Do not leave the doors or windows open. The room will be closed after hours if the rules cannot be followed.
5. You must make a reasonable attempt at the labs in order to pass the course. Failure to do so will result in a grade of INComplete.

Each workstation in the ECE 124 lab is equipped with:

1. Altera DE2 Board housing a Cyclone II Field Programmable Gate Array (FPGA) chip and a multitude of peripheral components
2. Altera Quartus-II FPGA Design Software

In this section you are going to be briefly introduced the on-board components.

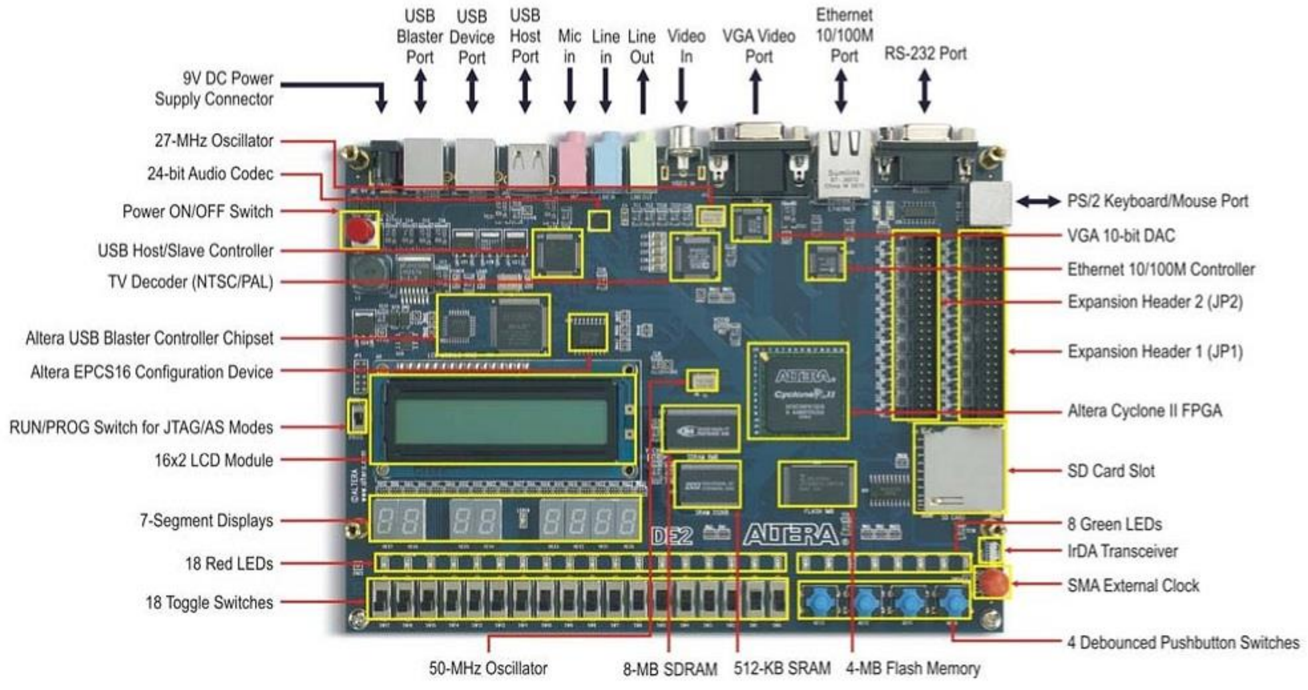


Figure 1 The DE2 board

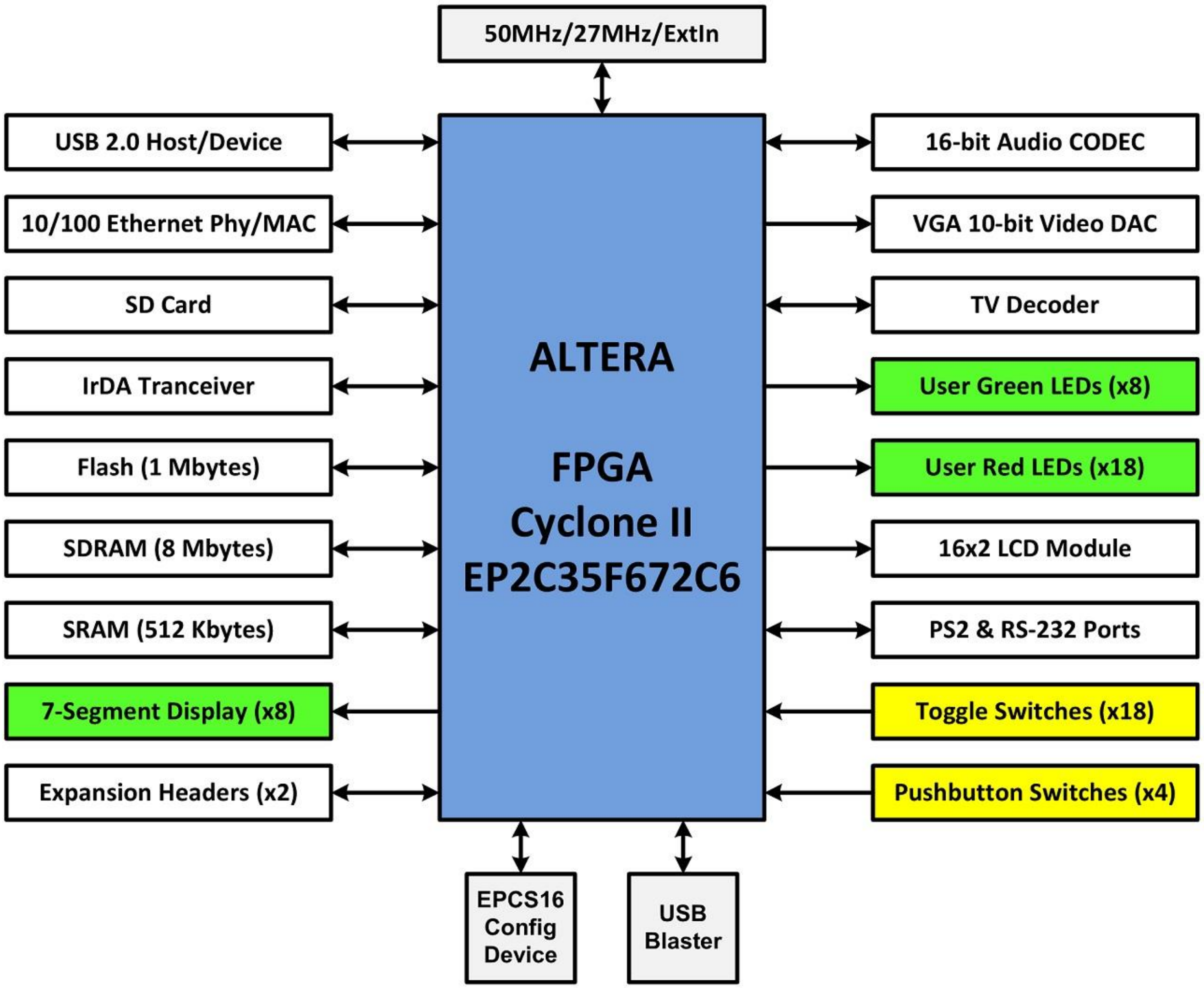


Figure 2 FPGA block diagram of DE2 board

1.1 Field Programmable Gate Arrays (FPGAs)

A FPGA is a Field Programmable Gate Array; basically an array of generic gates to perform any logic function. Many FPGAs simply use small blocks of memory, called CLBs (Combinational Logic Blocks), to look up the answer to equations of 4 or 5 variables. In the past, AND and OR gates would be interconnected to solve equations; but this has been replaced with CLB's as they are more flexible and can be used as memory blocks.

As not all equations have as little as 4 variables; typical designs will be spread over several CLBs; requiring signals to be routed between the CLBs. Just how much circuitry there is, and how fast it will run, in a particular FPGA, depends upon the speed of CLBs, the amount of resources for routing signals between CLBs, and how well a design can be "laid out" or optimized.

Newer FPGAs are tailored for specific circuits. The Cyclone II FPGA, which is in our labs, has hardware multipliers and adders, which run at 250MHz, allowing ultra-fast digital signal processing circuits to be built.

Configurable interconnects are provided between the chip resources (CLBs, hardware multipliers and memory blocks for the Cyclone II FPGA). The logic, circuitry, and interconnects in the architecture are configured by uploading a programming file to the FPGA chip. This property makes the FPGA chip very flexible since it is able to realize different digital circuits by uploading a different programming file. FPGAs are different that microprocessors or microcontrollers because the designer is able to change the hardware realized by the chip by programming it. Since hardware is always faster than software FPGAs allow hardware to be built with nearly the speed of software development.

1.2 Altera Quartus-II FPGA design software

Quartus is a full FPGA design software suite. It aids the designer through the different stages of describing the hardware design and targeting it for a certain Altera FPGA chip. The design process proceeds through the following stages:

- **Design Entry**: allows the designer to enter a hardware design specification using:
    - **Hardware Description Language**: such as VHDL or Verilog (we use VHDL)
    - **Schematic Entry**: by connecting blocks of ranging complexity. It can be used to interconnect simple components such as simple logic gates, or to interconnect previously created hardware modules
- **Design Compilation**: Once the design has been specified and entered into the tool, the designer must perform compilation which will take the design through various steps:
    - **Analysis and Synthesis**: A HDL or schematic file is analyzed and the hardware is broken down and mapped to the device resources (CLBs, flip flops, memory elements, .. etc) so that design logic is implemented via the available resources on the target chip
    - **Place and Route**: actual placement of design on certain device resources and routing it through the programmable interconnection take place in this step
    - **Assembly**: a programming file is produced so that it can be uploaded to the FPGA chip
- **Circuit Simulation**: In order for a designer to verify the functionality of their design simulation is required. The simulation step ensures that the circuit operates in the expected manner. A simulator is fed with the design description files and waveforms describing the input values against time to the circuit under test. The simulator then produces the logic values that will appear on the circuit outputs as waveforms also against time.
- **Timing Analysis**: It gives an accurate indication of how fast the circuit runs, and if speed and timing constraints can be met. Electronic circuits always have speed requirements to be met and being able to ballpark how fast a design works without having to build and measure it greatly speeds up design time.
- **Programming the FPGA**: In this step the programming file is uploaded to the FPGA chip to realize the design. The circuit can be physically tested afterwards by applying inputs and observing outputs

1.3 DE2 FPGA board peripherals

The DE2 board is equipped with peripherals that can be used to create various applications such as SDRAM, SRAM and flash memory chips, SD card socket, audio CODEC, VGA digital to output convertor and many others. For the purposes of the labs we are only using switches and push buttons for supplying inputs and the following peripherals for displaying outputs:

1.3.1 Light Emitting Diodes (LEDs)

LEDs are electronic components which can emit light with much greater efficiency than incandescent lamps. The Altera DE2 board has many outputs but you will only use the LEDs.

1.3.2 7-Segment display

A 7-segment decoder takes an input, typically a 4 bit binary number, and correctly drives a 7-segment LED display so that a person can see a number or letter as opposed to trying to interpret the original 4 bit binary number.

The 7-segment display is so called because it is 7 bar shaped LEDs arranged in such a way that the numbers 0 to 9, and letters A to F, can be displayed. Seven segments is the minimum number which can uniquely display numbers and that is why they are used for many calculator or electronic displays.

1.4 VHDL basics

VHDL is a language used by a hardware designer to describe the behavior of hardware. A synthesis tool then converts this into a circuit to be built. VHDL is helpful for the design of digital circuits, and is one of two main languages in use; the other being Verilog. An array of other languages is becoming popular such as SystemC and recently SystemVerilog. Basic circuit constructs such as AND, OR, NOT (gates) and look-up tables (SELECT statement), counters and flip-flops are straight forward. The VHDL language is very complex and was originally designed for the simulation of most anything. We encourage you to use the provided examples and stick with what you know will work. The most basic rule of VHDL is that if you can't understand how the CAD tool will create the circuit within the FPGA; then it's unlikely that the design will do what you expect. Because of this, it is vital that you have an understanding of basic logic circuitry and design.

2 Lab 1 – Design entry using Altera Quartus-II

The goal of this lab session is to gain experience with the Altera Quartus FPGA design software with both circuit entry and simulation and then modifying the provided circuits. The student will be programming the FPGA to verify that the circuit works in hardware. No prior experience with digital design, Altera Quartus or FPGAs is necessary; although experience with basic digital logic will enable one to understand the circuit.

2.1 Prelab

No prelab work is necessary for the first part, the introduction, of this lab experiment. A simple multiplexer circuit has been provided. After the self-guided introduction the student will modify this circuit to perform a different task. The lab starts with a brief introduction to the laboratory room, the equipment and the lab experiments. Lab 1 is composed of four parts in which you will:

1. Design entry of a VHDL code for a small digital circuit
2. Simulate the circuits to check that they operate correctly
3. Program the FPGA and check that the hardware implementation functions correctly
4. Modify the provided circuit to provide new functionality, simulate & test it and submit a report

2.2 VHDL design entry using Altera Quartus-II

To start the software click Start then select the program group Altera. Then click on the Altera Quartus II icon. To create a project select **File** -> **New Project Wizard** and then enter as much information as you wish. Create a new project called **Lab1.** In project wizard page 1 you must set a working directory for your project. You should use "**N:\ECE124\Lab1"** or something similar; creating a unique, logical, directory name which describes your project. Next you must select the name of your project that must be the same name as your top level design entity (**Lab1**). Then click on **Next** and say yes for the creation of the directory. Page 2 then allows you to add VHDL files to your project. Skip this now by clicking on **Next**. In page 3 you must select the Altera FPGA family **Cyclone II**, and **EP2C35F672C6** from available devices. In particular the DE2 board uses a **FBGA** package with **672 pins**, and **speed grade 6**. Once you've selected the correct part you should select **Finish** in order to skip the next page.

2.2.1 Pin assignment

Pin assignment is the process which maps the input and output signals of your design to physical pins of the FPGA chip available on the hardware board. As mentioned earlier the chip has a vast amount of resources and the design can be built anywhere on the chip in the place and route step. The inputs and outputs can be exported to any pins on the chip. Pin assignments act as constraints to specify where exactly your pins are going to be exported and are necessary as the peripherals are already pre-wired. To use different DE2 board peripherals such as LEDs, buttons and switches we must go through pin assignment. Since each peripheral is connected physically to a certain pin in the FPGA on the board, we should make sure the tool exports our inputs and outputs to the peripherals we target for use. Pin names are complex to use so the pins assignment file gives a logical name to each pin to be used throughout the design and maps it to a certain physical pin name. You can open the supplied file to see how this mapping is done.

Download the supplied pin assignment file (DE2_pins.csv) from course web site and save it in your project directory. Click on **Assignments->Import Assignments**, then click on the **[...]** to select the saved pin assignment file (DE2_pins.csv), which you've saved to your working directory and click **OK**.

For the Altera DE2 FPGA board, unused pins should all be left as inputs tri-stated. Unfortunately the default is to ground unused pins (this reduces noise and is what one should do for a product to be shipped). To change this setting click on **Assignments->Settings** then select the **Device** category and click on **Device & Pin Options** then select the **Unused Pins** tab and change the value to **Input tri-stated**. Click **OK** twice to close the windows.

2.2.2 Adding VHDL codes

You may use a VHDL or Schematic (Block Diagram) design in your project. For this part of lab, download the "lab1.vhd" file from web site and add this file to your project directory. To add it to the project, click on **Project->Add/Remove Files in Project**, find and select the VHDL file on your local directory and click **Add** and the **OK**.

2.2.2.1 Understanding VHDL structure

VHDL is case insensitive. For this reason many coders use all lower case. VHDL language uses two main structures to describe a design unit (hardware block):

1. **Entity**: it declares the design unit name and the ports (which are inputs and outputs of the entity or design unit) associated with it. Each port name, type (input or output) and width (number of bits) is declared in the entity.
2. **Architecture**: the architecture specifies the actual functionality of the entity. Notice that the entity has no information about how the hardware block uses the inputs to produce the outputs - that is the role of the architecture associated with the entity.

There are two ways to describe the functionality of a certain block:

1. **Behavioral**: where the relation between input and output is declared using logical equations.
2. **Structural**: where you can use previously created entities in your design unit as components. For example if you built an adder unit you can use it, as a component, in designing a microprocessor.

2.2.2.2 Understanding the VHDL code

Understanding the given VHDL code will help you in the upcoming labs. The given code consists of one design unit (single entity and architecture). "Lab1" is the top level entity for the design. It has 2 input ports (key and sw) and 2 output ports (ledr and ledg). Notice that the names of the ports are case insensitive and identical to those ones in the pins assignment files and are similar to the names on the DE2 board. We are using those logical names in order to map our inputs to the keys, switches, red and green LEDs respectively. Analyze the code and try to familiarize yourself with VHDL, it is fully commented. If you still cannot figure out a certain line ask for help.

2.2.3 Compiling the design for the FPGA

To compile the design and make a programming file for the FPGA click on **Processing->Start Compilation** or use “Ctrl+L” or the arrow button on the toolbar. You will see around 100 messages; mostly due to pins which have been defined but are not being used. Near the bottom of the Quartus II window you will see several tabs. You can click on **Warning** or **Critical Warning** or **Error** or **Info** messages to see the details. You should always check the **Error** and **Critical Warning** messages and resolve them. Prudent users should also check the **Warning** messages if the design isn't working as expected. If there are any error(s), compilation will be stopped and one can read the error message(s), divine the problem and fix it. Expect to see many warnings because of the pin assignment file as it defines almost every pin on the FPGA and you will be warned, at least once, for each one that is not connected.

The following steps are done by the software to convert the schematic circuit and/or HDL (e.g. VHDL, Verilog) circuit into a file which is used to program the FPGA:

- **Analysis & Synthesis:** This stage converts the design into parts which are available within the selected FPGA. Parts which are available are typically flip flops, memory blocks, look-up tables and adders; and sometimes multipliers and other complex support parts.
- **Fitter:** This places the parts within an FPGA, connects them together and to the input and output pins, and optimizes the layout for the user goals (typically speed). A design may require hundreds, or thousands of CLBs (Combinational Logic Elements), LEs (Logic Elements) or LUTs (Look-Up Tables).
- **Assembler:** Converts the fitted design into a file which can be used to program the FPGA.

2.2.4 Simulation

Altera Quartus-II comes with a simple simulator which is limited by its graphical nature. The input stimulus is applied as waveforms, and then the simulation is run and the output analyzed. The simulation is not interactive, and so can be quite slow for debugging. It also does not allow one to view anything but input and output pins and internal registers.

To simulate your circuit, a vector waveform File must be created. This specifies the inputs to the circuit. To do this click on **File->New** then select the **Verification/Debugging Files** tab and select **Vector Waveform File**. That will open a file typically called Waveform1.wmf. In the left window of that file, right click and select Insert Node or Bus and then click on the **Node Finder** button. In the filter tab scroll up and select **Pins: Input** and then click the **List** button to show all input pins. Click on the ">" arrow to add these input pins sw(0), sw(1), sw(3), and key(0) to the **Selected Nodes** column. You could repeat this for **Pins: Output** (ledg[0], ledg[1], ledr[0]) but this is not necessary as the simulator adds all outputs by default, then click **OK**. For simulation you need to set a series of ‘0’s or ‘1’s to inputs (input test pattern). To assign periodic values, right click on a signal and select **Value->Clock** and then set the following period for each signal (Table 1). It will switch each signal between ‘0’ and ‘1’ at a certain interval. By choosing intervals that increase by a factor in power of two ($2^{number\_of\_inputs}$), all possible combinations of inputs will be generated (Figure 3).

Table 1 Input signals clock assignments

Inputs	sw[0]	sw[1]	sw[3]	key[0]
Period	50ns	100ns	200ns	400ns

Don’t forget to hit “Ctrl+W” or right click in the waveform window and select **Zoom->Fit in Window**. By these assignments, all possible combinations to the inputs will be generated (an exhaustive test), which is only suitable for automated testing or where there are a few combinations to visually check. The “OR” and “AND” logic gates are easy to verify. One can force inputs to values other than a periodic wave. For instance, the **Value->Count Value** lets you go through all possible inputs - as does the random method - but for a more complex circuit one would test specific combinations of inputs, not all. When you are done creating your vector waveform inputs, you should save the resulting file into your project directory.

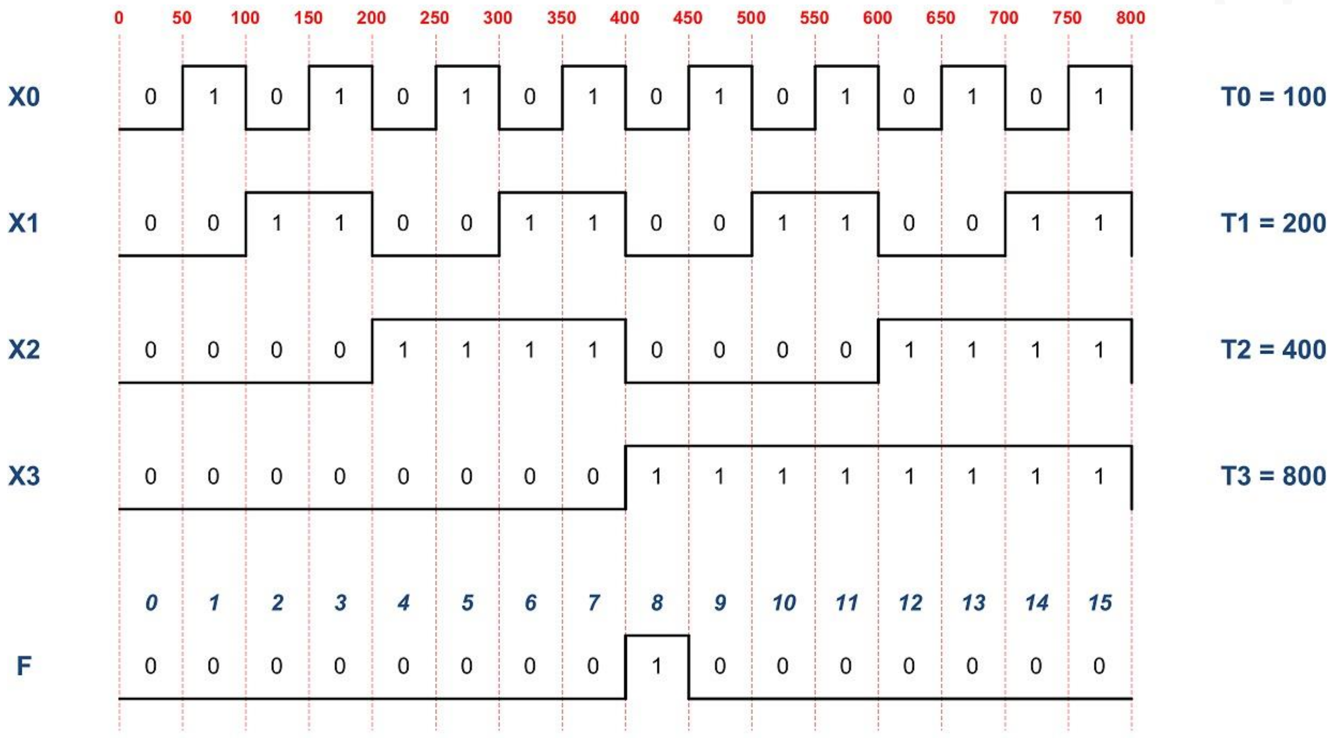

0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800
X0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
T0 = 100
X1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
T1 = 200
X2 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
T2 = 400
X3 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
T3 = 800
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
F 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

Figure 3 Exhaustive test input generation for function F(X0,X1,X2,X3) = ~X0 & ~X1 & ~X2 & X3

There are two types of simulation:

- **Functional**: This is a verification of the basic functionality of the circuit with no timing information (i.e. gate delays) included.
- **Timing**: Timing simulation uses information about the circuit design, and how the circuit is fit into an FPGA, to estimate accurate time delays for signals. This simulation is slower; but it is also necessary to determine if a design can meet the speed requirements.

Now, you can run the simulator by clicking on **Processing->Simulator Tool**. Set the **End simulation at** value to 1us (default). Then Select **Timing** as Simulation mode. For the **Simulation Input** field select your simulation file just created. Then click on **Start**. Wait until it finishes and then click the **Report** button to see the simulator output. Hit “Ctrl-W”, or right click and choose **Zoom->Fit in Window** to view the entire simulation.

Remember from “lab1.vhd” code (line 45) that for example: “ledg (0) = sw (0) | sw (1)”.

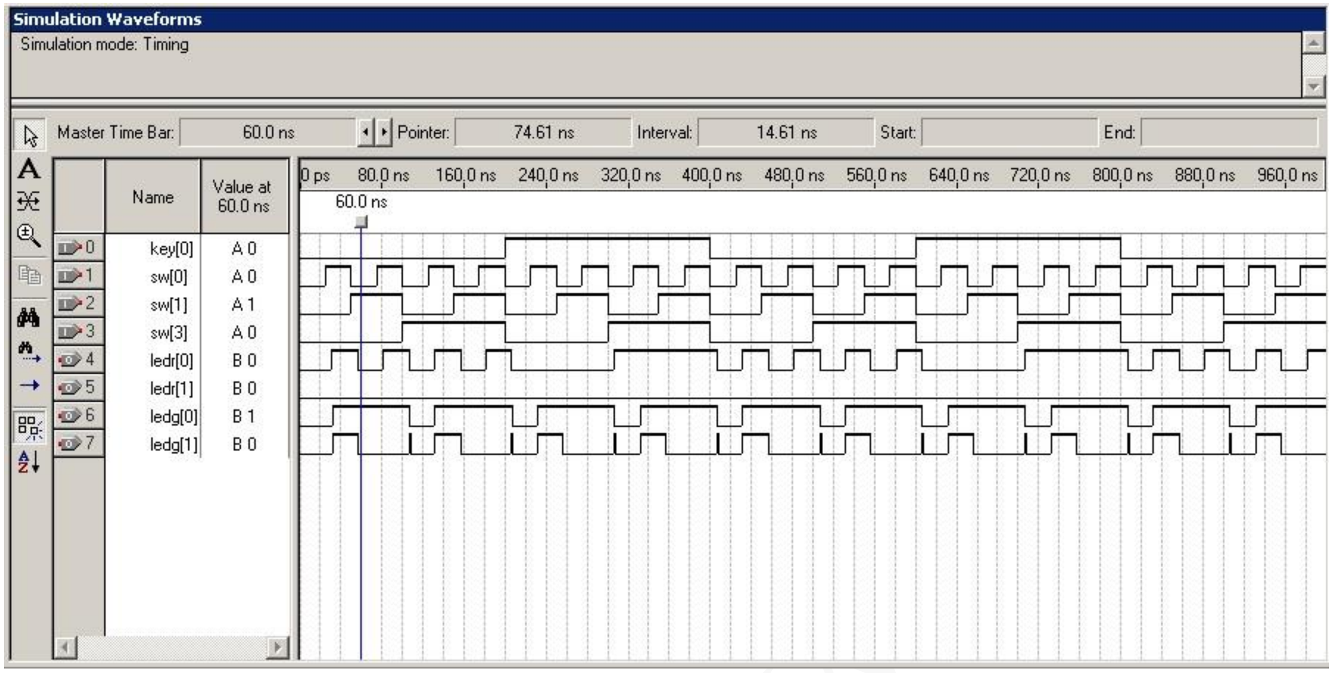


Figure 4 Timing-mode simulation report for Lab1.vhd design

For timing simulation, you may notice that there are some very narrow spikes. These are glitches due to the circuit, or the way that simulation is done and can be generally ignored for this course. You'll also notice that there is an approximately 10ns delay between the inputs changing and the outputs changing. Under **View**, or via the magnifying glass icons, you will find zoom controls. Zoom out to show the full simulation.

Sometimes, a functional simulation is also required in each lab. You must Select **Functional** as Simulation mode and then click on **Generate Functional Simulation Netlist** tab to generate the necessary files. For a functional simulation, you will not see any time delay. Here is a sample result of a timing simulation, verify its functionality matches with the VHDL code.

2.2.5 Timing analysis

Timing analysis can only be run after a design is successfully implemented and it gives an accurate indication of how fast the circuit runs, and if speed and timing constraints can be met. Electronic circuits always have speed requirements to be met and being able to ballpark how fast a design works without having to build and measure it greatly speeds up design time. To run the analyzer, click on **Processing->Classic Timing Analyzer Tool** and click the **Start** button. Click on the **t$_{pd}$** tab and note that the slowest signal is at the top of the list. This is the time delay from "P2P" (Pin to Pin). You likely have a value around 11ns. Other tabs provide you time delays for more complex circuits with flip-flops.

By default, all paths in the circuit get analyzed and listed. There are two categories to consider:

- **t$_{pd}$**: This is the time required for a signal to go from an input pin to an output pin through combinational logic

- **t_{co}**: For registers and flip-flops, this is the time required for an output to become valid after the a clock signal transition. Also pay attention to the **t_{SU}** table. That table lists the length of times for which each data must be present (setup) before the clock transition. So the worst case is the slowest sum for **t_{CO}** plus any associate setup times. Note that we've neglected the hold time of the data for clocked data and this can be important in real-world designs.

To quickly get timing information, click on the **Report** button and note the first row information ("Worst-case t_{pd}"). From this analysis one can see that the circuit would work at a maximum speed of >90MHz; which isn't very fast compared to the 3+GHz speed of modern computers. However, much more complex circuits would also work at the same speed, and this FPGA has other resources such as adders and multipliers which work at 250MHz allowing for real-time HDTV image manipulation.

2.2.6 Programming the FPGA

Make sure that the power to the FPGA board is on (the red button in the upper right corner). The programmer tool can be found under **Tools->Programmer**. You will need to make sure that the correct programmer is selected. Click on **Hardware Setup** and select **USB-Blaster**, if necessary. Ensure that the "Currently selected hardware" says **USB-Blaster [USB-?]**. Next, ensure that for your project the "Lab1.sof" file in the list has a check mark in the box under "Program/Configure", and then just click on the **Start** button in order to program the FPGA.

2.2.7 Test the design on DE2 board

Now the design can be tested on FPGA board. Verify the functionality regarding to the "lab1.vhd" file:

- Turn on/off SW0 and SW1 inputs, to verify the functions behind outputs LEDG0 and LEDG1 (check with VHDL code in lab1.vhd)
- Turn on/off SW1, SW3 and KEY0 inputs to verify the function behind output LEDR0 (check with VHDL code in lab1.vhd)

2.3 Design your own circuit – Car-Controller

For your demo, modify the "lab1.vhd" code to make its function as an automotive controller. The inputs and outputs are defined in below table.

Table 2 Car-Control circuit IO definition

Signal Type	Signal Name	Assigned Port	Description
Inputs	Gas	KEY[0]	
	Clutch	KEY[1]	
	Brake	KEY[2]	
	Override	SW[1]	Master switch to shut down the car
Outputs	GasControl	LEDG[0]	When ON (logic '1'), acceleration is given to the motor
	BrakeControl	LEDR[0]	When ON (logic '1'), the brakes are engaged

Consider the following guidelines in your design. A typical block diagram for your design is also shown.

- The pushbuttons (KEY[3:0]) on the DE2 board are inverted. They are '1' when NOT pressed and become '0' when pressed.
- Note that some keys (KEY[3:0]) or switches (SW[17:0]) do not work properly sometimes (have physical damage). Replace them with other ones.
- Make sure that the inputs and outputs you need are in the VHDL code ENTITY declaration. Remove extra ports and signals from the design.
- Add a brake safety so that the "Brake" being ON, turns on the "BrakeControl" and turns off the "GasControl".
- Add a clutch safety so that the "Clutch" being ON, turns off the "GasControl".
- Add an engine safety so that the "Override" being ON, turns off the "GasControl" and turns on the "BrakeControl".

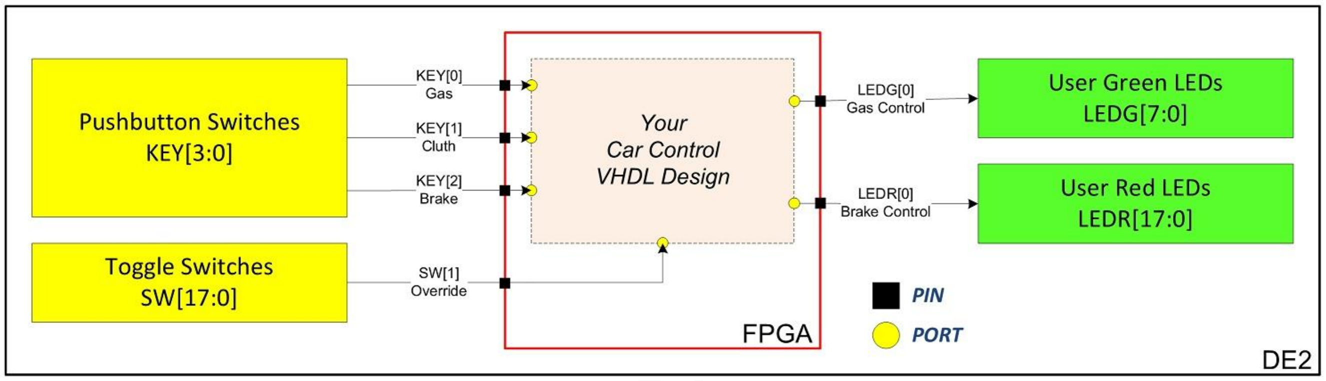


Figure 5 Car-Control circuit block diagram

2.4 Postlab

Download, print and fill out the "Lab1SubmissionForm.pdf" from and demonstrate your Car-Control design on scheduled date. Then hand in the report, one day after demo session. The submitted report for Car-Control design must include:

1. Completed "Lab1SubmissionForm" as the report front page. Don't forget to fill out the "total logic elements" in the form.
2. Implementation procedure, design decisions, encountered problems or bugs with solution to them and debugging techniques (2 pages max). Don't forget to include the RTL view of your circuit (**Tools->Netlist Viewers->RTL Viewer**).
3. Fully commented VHDL code printout (2-up, landscape and double sided). Use meaningful name for your signals.
4. Timing simulation waveform for the circuit showing that the circuit works in all cases (exhaustive test).

3 Lab 2 – Combinational circuits; Arithmetic Logic Unit - VHDL Design

The goal is to build a simple VHDL circuit to choose between various calculations and logical operations. This is basically a calculator which can perform multiple operations with no memory, i.e. the circuit is completely combinational.

3.1 Prelab

First download the "lab2.vhd" code from web site and open it a context sensitive editor or Quartus II. At the top of the file, a new entity for a seven-segment is defined. Try to understand its IO mapping (ENTITY) and function (ARCHITECTURE). In this lab, we will instantiate this entity multiple times to display binary values in hexadecimal format.

3.2 Lab requirement – ALU VHDL design

Create a new project for Lab2 and add the "lab2.vhd" file to the project. After compiling, program your FPGA to watch its functionality. For every VHDL design entry, you need to follow all instructions in section 2.2. Test the circuit with different combinations of input signal values and check its output with the VHDL code.

Now, for your demo, modify the code in "lab2.vhd" to make its function as an ALU. The inputs and outputs are defined in Table 3. You may use other meaningful signal names in your design. Again, you need to follow all instructions in section 2.2.

Table 3 ALU design IO definition

Signal Type	Signal Name	Assigned Port	Comment
Inputs	Operand 1	SW[7..0]	8-bit input to be displayed on HEX5 and HEX4
	Operand 2	SW[15..8]	8-bit input to be displayed on HEX7 and HEX6
	Operator	SW[17..16]	2-bit input to be displayed on LEDR[17..16]
Outputs	OperationResult	HEX2,HEX1,HEX0	9-bit output result to be displayed on LEDR[8..0] too

Your design is supposed to implement a simple calculator with four pre-defined operations.

Table 4 ALU operations

SW[17..16]	Operator	Description
00	AND	Logical AND of 8-bit inputs
01	OR	Logical OR of 8-bit inputs
10	XOR	Logical XOR of 8-bit inputs
11	ADD	Binary ADD of 8-bit inputs

Consider the following guidelines in your design. A typical block diagram for your design is also shown.

- SW[7..0] and SW[15..8] represent the inputs from most significant bit (MSB) down to least significant bits (LSB) for the first and second operands. Here, SW[7] and SW[15] are most significant bits.

- The result is displayed on HEX2, HEX1 and HEX0. HEX2 shows the most significant hex digit of the result, while HEX0 shows the least significant hex digit of the result. This makes your result readable on the board. Notice that we are using 3 hex digits to display the result, though we have 8-input operands. That's because addition of two 8-bit numbers can cause carry (9th bit) to occur. This is the only case in which HEX2 will display 1 instead of being blank. To blank HEX2 use the blanking input to blank or turn it off.
- Signals are intermediate values that must be declared using "signal" statement in the VHDL architecture before the "begin**"** statement.
- Simulation for this lab has some extra complications. Do not add the vector for the inputs (e.g. KEY or SW) but add individual wires (e.g. KEY[0) and when they are added, in the Vector Waveform File one can group them together. One bundle of 8 wires can be labeled as "A". To group wire together, select them and then press right click and choose "**Grouping->Group**".
- To assign values to input operands and operators, first group them with meaningful signal names and then use "**Value->Arbitrary Value**", "**Value->Random Values**" or "**Value->Count Value**".
- Instead of looking at the raw 7-segment outputs in the simulator report (they are not meaningful), it makes more sense to look at the LEDR[8..0] value (group them together) to have faster and more intuitive testing. Remember to display the operation result on LEDR[8..0].
- A **WHEN** statement can be used instead of an **IF** statement to implement a multiplexer as we have in sample VHDL file "lab2.vhd". This multiplexer should be taken into consideration. As mentioned earlier, HEX2 only displays the carry out value in ADD operation. This can be achieved by concatenation of A and B to a string of 4 zeros and then performing addition. This will preserve carry and should produce a 12 bit result. This result can be used to drive the 3 seven segment decoders. This is the heart of the project and will take 5 lines of VHDL code.
- Note that if the result is declared as a 12-bit signal, all operations must produce 12 bit results. So concatenation is not only required only for addition but also for all other operations too.
- The most difficult part will be the add operation while the result has 9 bits and for binary addition in VHDL signals must be declared in **unsigned** type. We need to recast the signals to **unsigned** type (which are originally in logical type **std_logic_vector**) and then recast again to **std_logic_vector** type. You may use statement like this to recast signals:

R <= std_logic_vector(unsigned (A) + unsigned (B))

- In total, about 15 lines of VHDL code are required to be written. If your design is taking much more than this STOP, and talk to the lab staff.

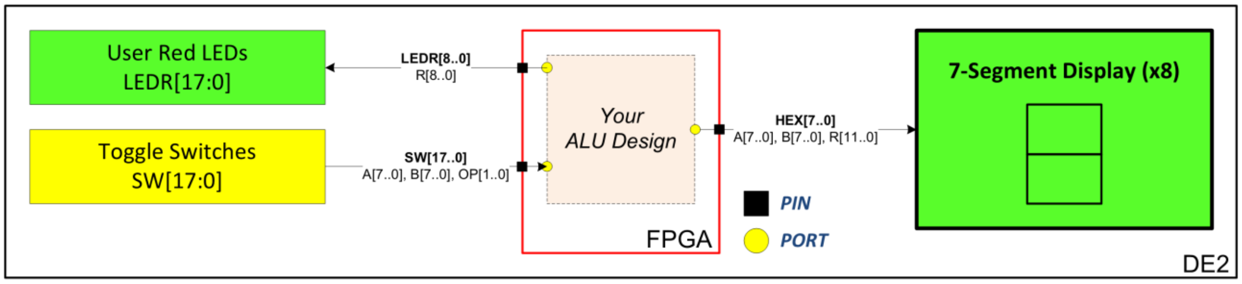
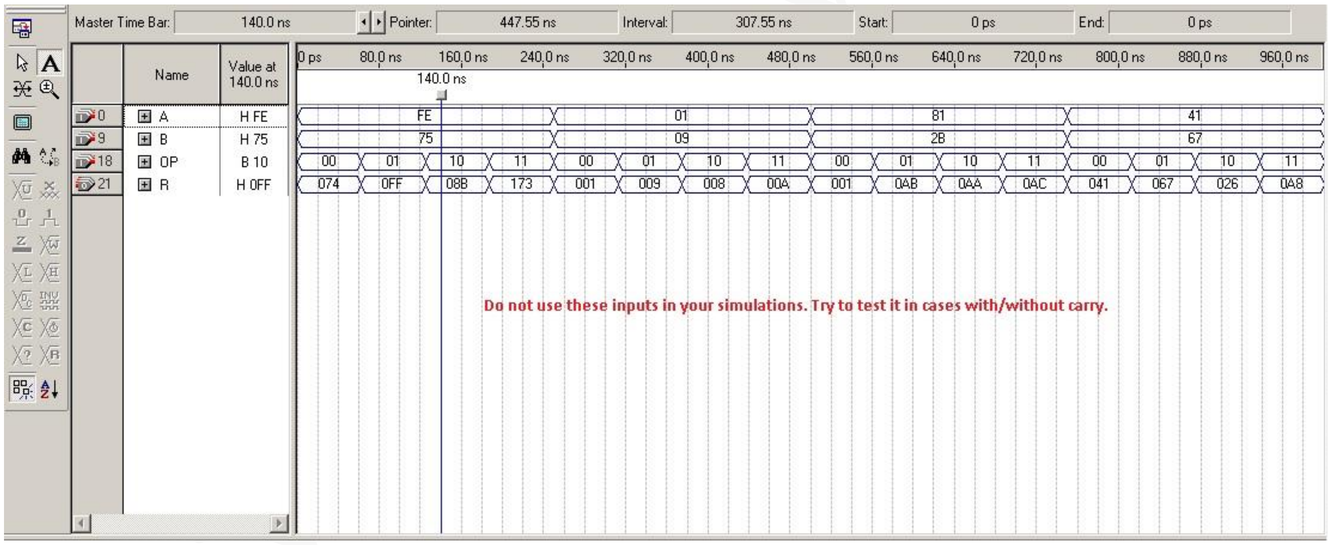


Figure 6 A typical block diagram for ALU design

3.3 Postlab

Download, print and fill out the "Lab2SubmissionForm.pdf" from and demonstrate your design on scheduled date. Then hand in the report, one day after demo session. The submitted report must include:

1. Completed "Lab2 Submission Form" as the report front page. Don't forget to fill out the "total logic elements" and the Worst Case Speed (ns) t_{pd} in the form.
2. Implementation procedure, design decisions, encountered problems or bugs with solution to them, debugging techniques and RTL view of your circuit (2 pages max).
3. Fully commented VHDL code printout (2-up, landscape and double sided).
4. Functional simulation waveform with coverage for critical cases. Mark your simulation waveforms explaining several different scenarios for each operation. You must prove that what you have designed is working. For instance, testing with the inputs set to 0 does not allow one to distinguish one operation from another or if any operation in particular is fully working. Often one tests critical cases - the limits where things may break (i.e. overflow). Do not print waveforms of all possible cases - give samples for critical cases of each operation and explain how you checked for proper operation. The goal is to have a simulation to prove that the circuit works; without doing a full exhaustive test of all possible inputs. It should prove that all operations work correctly for enough numbers to give confidence that the circuit is fully functional.


```
Master Time Bar: 140.0 ns   Pointer: 447.55 ns   Interval: 307.55 ns   Start: 0 ps   End: 0 ps

     Name  Value at 140.0 ns
0    A     H FE   FE                      01                      81                      41
9    B     H 75   75                      09                      2B                      67
18   OP    B 10   00   01   10   11   00   01   10   11   00   01   10   11   00   01   10   11
21   R     H 0FF  074  0FF  08B  173  001  009  008  00A  001  0AB  0AA  0AC  041  067  026  0A8
```
Do not use these inputs in your simulations. Try to test it in cases with/without carry.

Figure 7 Sample functional simulation output

4 Lab 3 – Combinational circuits; Elevator Controller – Schematic design

The goal of this lab session is to design two simple circuits using truth tables and K-Maps. Translating a written circuit description into a Truth Table is the basis of this circuit design. Optimizing the design with K-maps allows one to make the design smaller and faster. You are to design and implement a circuit to control an elevator. The circuit has four inputs to define current and requested floors and two outputs to enable the motor and show the up/down direction.

4.1 Prelab

To prepare for this lab the student should reduce the design descriptions to truth tables. The truth tables should be then used to build K-Maps. Extract logical expressions from the K-Maps and realize the two circuits required as gates.

4.2 Schematic design entry using Altera Quartus-II

In this lab, we first create a simple project using the schematic editor and then modify it to implement Elevator Controller. To start, create a project and name it Lab3, repeat the same steps to import the pins assignment file and to set unused pins to input tri-state in section 2.2.1. Proceed by adding the schematic using the schematic editor. Click on **File->New** and select **Design Files->Block Diagram/Schematic File**. Primitive parts (AND, OR, other gates) can be entered via the **Symbol Tool** which looks like an AND gate. Click on it and then expand the **Libraries** item to reveal **primitives**. Expand that to reveal **logic** primitives such as gates. You will also need to use the **pin** and **other** primitives such as **input**, **output**, **gnd** (ground or logic '0') as well as **vcc** (high or logic '1'). The **storage** primitives include flip-flops. To draw wires you will need to use the **Orthogonal Node Tool** from the toolbox on the left. Drawing wires can be painful as you will discover. Draw wires in pieces to make the task easier. To connect nets or wires to the outside world you will also need to use the input and output **pin** primitives. To name your pins just double click on them and use the standard names (e.g. sw[0], ledg[0]) which is defined in your pin out file, on the schematic above (documented in Appendix A). Do NOT right click on a wire or node, and in the Properties tab assign a name. That sometimes results in a circuit which does nothing. Now, draw the schematic shown in Figure 8:

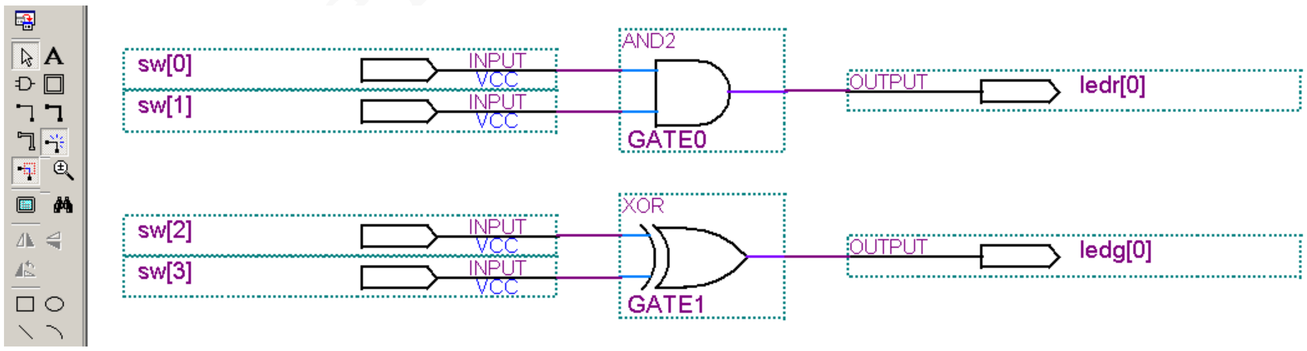


Figure 8 Lab3.bdf - Schematic design file

Compile this design and download it to the FPGA for testing. If you have time, simulate it by setting the proper input values as clock periods for all possible scenarios (as done in previous labs). Remember to zoom and view the whole simulation with "Ctrl+W".

4.3 Lab requirement – EC schematic design

For your demo, modify the "lab3.bdf" file to make it operate as an elevator controller. The inputs and outputs are defined in the Table 5.

Table 5 Car-Control circuit IO definition

Signal Type	Signal Name	Assigned Port	Description
Inputs	CurrentFloor	SW[1..0]	
	NextFloor	SW[3..2]	
Outputs	Enable	LEDG[0]	When ON (logic '1'), the motor is turned on
	Direction	LEDR[0]	Define moving direction (upwards: '1', downwards:'0')

Design two versions of this circuit one using any types of 2-input logic gates and the other only 2-input NAND gates. Note that K-maps help you minimize a circuit with any type of gates, but it will not directly help you to minimize a design where you are restricted to use a specific gate type (e.g. NAND, NOR). There are two ways to design the circuit:

1. Using only 2-input NAND gates exclusively for the first design and minimizing wires and gates using any gates available for the second design
2. Starting with designing a circuit with 2-input logic gates of any types and then replacing the gates with 2-input NAND equivalents

Considering one of above approaches, start the schematic editor to build your two designs using switches for the four inputs and LEDs for the outputs. Simulate your designs and upload it to the FPGA for physical testing.

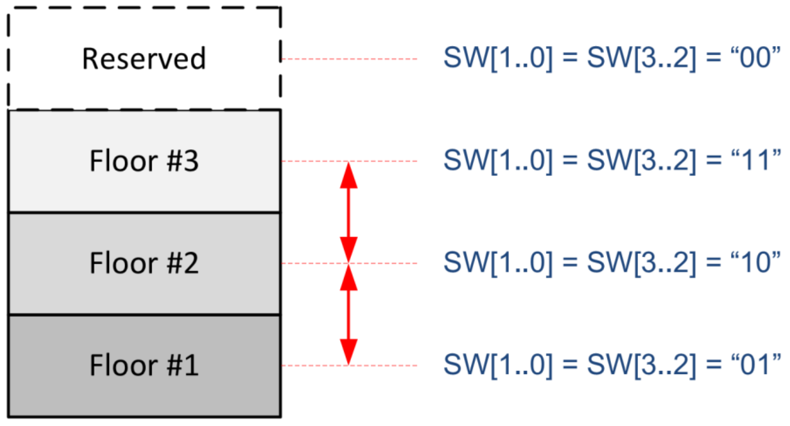

Reserved
Floor #3
Floor #2
Floor #1
SW[1..0] = SW[3..2] = "00"
SW[1..0] = SW[3..2] = "11"
SW[1..0] = SW[3..2] = "10"
SW[1..0] = SW[3..2] = "01"

Figure 9 3-floor elevator

Consider the following guidelines in your design.

- You can find a schematic inverter in the Quartus software called NOT under **primitives** -> **logic**
- K-Maps should be used to minimize the elevator circuit size. Note that there is no easy way to do this, for the NAND only design, without lots of experience. Note that there are many things that can be optimized for. One may choose to design for the minimum number of wires OR gates - it's hard to minimize both at the same time. We suggest trying this several times and include all in your Lab Report.

4.4 Postlab

Download, print and fill out the "Lab3SubmissionForm.pdf" form and demonstrate your design on scheduled date. Then hand in the report, one day after demo session. The submitted report must include: The submitted report must include:

1. Completed "Lab3 Submission Form" as the front page of your report. Don't forget to fill out the "number of gates" and "number of wires" in the form for demonstrated circuit.
2. Implementation procedure, design decisions, encountered problems or bugs with solution to them, debugging techniques and RTL view of your circuit (2 pages max).
3. Discussion of expandability of the design if the 4th floor is added and effect of that on reliability (you don't need to actually implement the 4th floor extra logic). Consider the number of wires and gates as being indicators of reliability.
4. How does the all NAND design compare to the other design? How many gates and wires are required for each? Note: This lab used to be built and so you should count ALL gates - inverters too - as they all had to be wired up. Which circuit would be more reliable?
5. Include the truth table for the elevator controller, K-maps and how you deduced logical expressions for the two circuits.
6. Schematic printout for the two circuits
7. Simulation waveforms for the two circuits showing that they work in all cases (please circle 4 points and identify what the inputs represent and why the outputs are correct according to the requirements). Demonstrate that you can interpret the simulation waveforms.

5 Lab 4 – Sequential circuits; Traffic Light Controller – VHDL design

The goal of this lab is to design a traffic light control system as a sequential circuit with clock. The system controls two traffic lights on an intersection using a state machine. First you will learn to implement and test, two clock dividers using modulus and binary counters.

The standard way to implement a clock divider is to use a binary counter. Binary counters increment their value by one on every rising/falling edge of its input clock signal. If you consider an n-bit binary counter, you notice that the first bit of the counter (LSB) toggles in the period of half-speed of the original clock. If you look at the second bit you would find it toggling at half speed of the first bit which means 1/4 the speed of the original clock. Thus Nth bit in the counter output is a clock in frequency of input clock frequency divide by 2^{N+1} where N is the bit position starting from bit 0. The binary counter is not very accurate if you need a clock signal with precise frequency because the divisor is always in power of two. So for accurate timing or clock division a modulus counter can be used. A modulus counter increments up to a certain number (terminal count value) and then resets to initial value. It toggles the clock signal at the terminal which is equal to half the period of your desired output clock. Thus you need to calculate the period of your clock to determine the terminal value. Figure 10 shows outputs of two 25-bit binary and modulus counters generated from a 50MHz input clock.

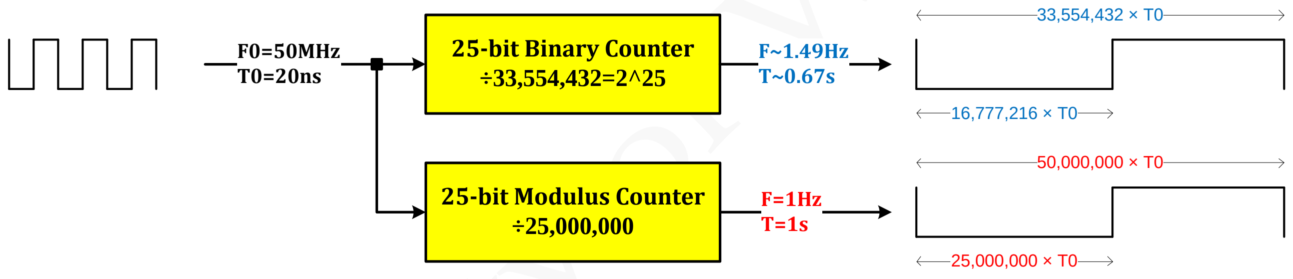


Figure 10 Binary and Modulus clock dividers by 25-bit counters

In this lab you need to implement sequential logic in VHDL. Sequential logic has "memory" and the output depends on the inputs and what the current state or "memory" is. This requires a **PROCESS** structure in your architecture to implement flip-flops and other memory elements.

```
Label_name: Process (Sensitivity_List)
Begin
.
.
.
End process;
```

The `Label_name` can be any name like apple, orange, etc. except reserved names in VHDL. The `Sensitivity_List` is the list of signals/variables/inputs that the `Process` is sensitive to. These are the signals that will trigger execution of the `Process` inside the simulator. Thus, any signal that is read in the `Process`, should appear in the sensitivity list. Missing signals from the sensitivity list cause simulation-synthesis mismatch.

The key statement to implement flip flops or sequential logic, is the **IF (rising_edge (clock)) THEN** statement.

In VHDL, state machines are built inside a process block. A case statement can be used to determine the next state depending on the current state and other signals (inputs or time events). On the edge of the clock the current state is assigned the value of the next state (determined by the case statement). It is worth mentioning that sometimes your decision can be staying in the same state. To name your states you may declare a new type in your architecture and then declare two signals of this type to keep your current state and next state.

5.1 Prelab

In this lab experiment, it is important to design and build your circuit incrementally. Do **NOT** attempt designing the whole circuit at once. First download the "lab4.vhd" file from web site and open by Quartus II. Browse the VHDL code carefully and try to understand all statements in the file. Look at the clock divider circuits and study how they work, how to change the frequency of the system clock, and how it could be further divided by 10. Try to extract the embedded state machine in this code and understand sequential logic design statements in VHDL.

5.2 Lab requirement

There is two steps for this lab. The first step is just to get you familiarized with creating sequential hardware. The second step is what you will demonstrate and will be marked for.

5.2.1 Step 1: Creating a sequence of synchronized events

Create a new project for Lab4 and add the "lab4.vhd" file to the project. For every VHDL design entry, you need to follow all instructions in section 2.2, i.e., import pin assignment file, set unused pins to tri-state, compile, program the device, etc. Test the circuit with different combinations of input signal values and check its function with the VHDL code. Do not try to simulate the design at this moment.

Now, for your demo, modify the "lab4.vhd" file to make a sequencer to generate the following pattern (Figure 11) repeatedly on green and red LEDs. The pattern starts with a flashing light on a green LED for two seconds followed by solid pattern for 5 seconds and then the same sequence with different durations on red LED. The inputs and outputs are defined in Table 6. You may use other meaningful signal names in your design. Again, you need to follow all the instructions provided in section 2.2.

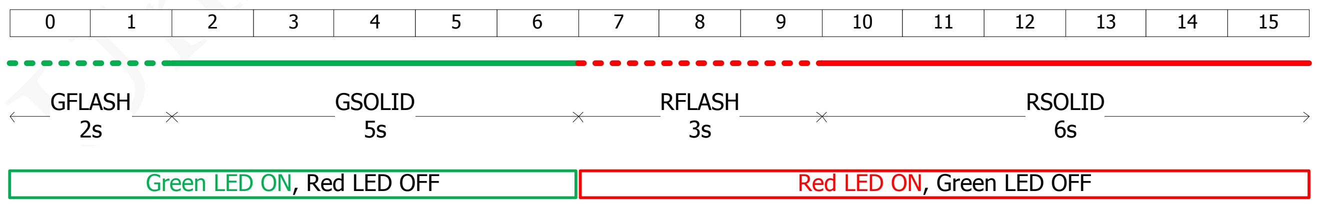


Figure 11 Sequencer timing diagram

Table 6 Sequencer circuit IO definition

Signal Type	Signal Name	Assigned Port	Description
Input	Input Clock	CLOCK_50	50MHz on-board clock
Outputs	Green LED	LEDG[8]	Green light pattern
	Red LED	LEDR[11]	Red light pattern
	Clock 1Hz Bin	LEDG[2]	1Hz output from binary counter
	Clock 1Hz Mod	LEDG[1]	1Hz output from modulus counter
	Clock 10Hz Mod	LEDG[0]	10Hz output from modulus counter
	State Number	HEX0	4-bit internal state number
	State Counter	HEX2	4-bit internal state counter

Start your design with modulus 10Hz and 1Hz clock dividers. Use this 1Hz clock to drive your state machine. Keep the binary 1Hz clock output for your demo. Figure 12 shows a typical block diagram for your design. You must be able to implement the sequencer FSM in four states. Display your internal signals (state number and counter) on Seven-Segment displays to debug your circuit.

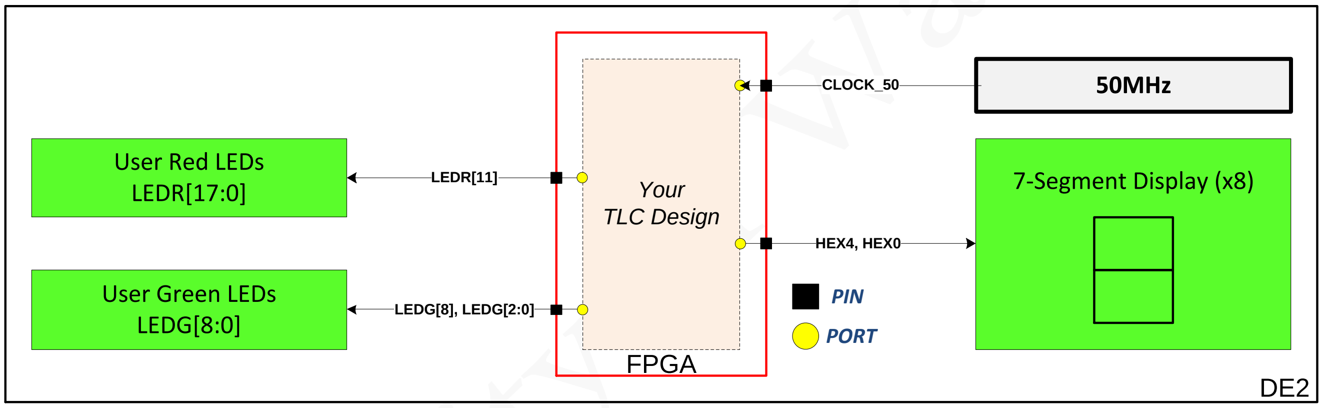


Figure 12 A typical block diagram for a sequencer design

Try to build the required components one by one to generate the timing diagram in Figure 11:

1. A modulus clock divider to create a 10Hz clock from the board's 50MHz clock input.
2. A modulus clock divider to create a 1Hz clock from the 10Hz clock.
3. The 1Hz clock will drive a counter to determine the time of transitions between states. The current state determines which LEDs are ON or OFF. Your states can be defined as:

- **GFLASH:** the green LED is flashing at 10Hz, while red LED is OFF.
- **GSOLID:** the green LED is ON, while red LED is OFF.
- **RFLASH:** the red LED is flashing at 10Hz, while green LED is OFF (like amber state).
- **RSOLID:** the red LED is ON, while green LED is OFF.

5.2.2 Step 2: Traffic Light Controller

After building the simple state machine and testing that it is functioning according to the given timing diagram, you are now required to extend the designed sequencer to implement a real traffic light controller in transportation system like Figure 13 .

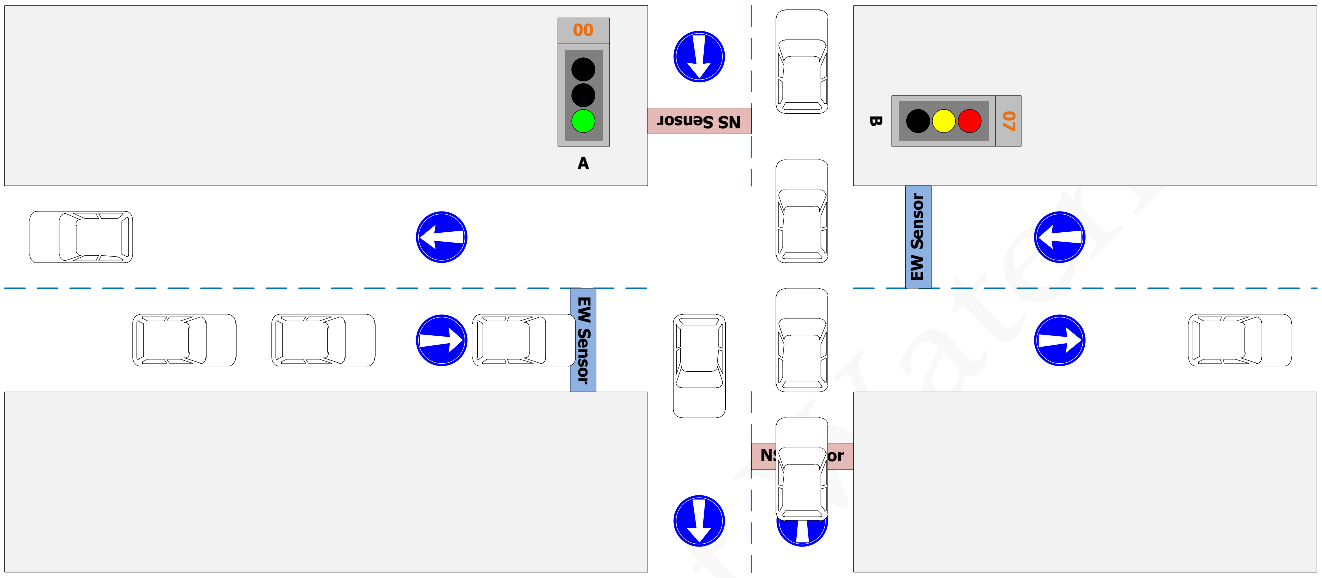


Figure 13 Traffic Light System

The traffic light acts normally as a simple sequencer, to transit between green, amber and red lights in real traffic light systems. As we don't have yellow or orange LEDs on our evaluation board, we replace them with a flashing red light in our design. You may design it in three states as:

- `Go:` - the green LED is ON. First two seconds is flashing at 10Hz for turning left cars.
- `PrepareToStop:` the red LED is flashing at 10Hz to warn drivers to be prepared to stop.
- `Stop:` - the red LED is ON.

As shown in Figure 14, the traffic light controller switches between above states with the predefined durations (i.e. 6 seconds, 2 seconds and 8 seconds respectively) continually. Note that in the first 2 seconds of "Go" state in this mode, the green LED is flashing.

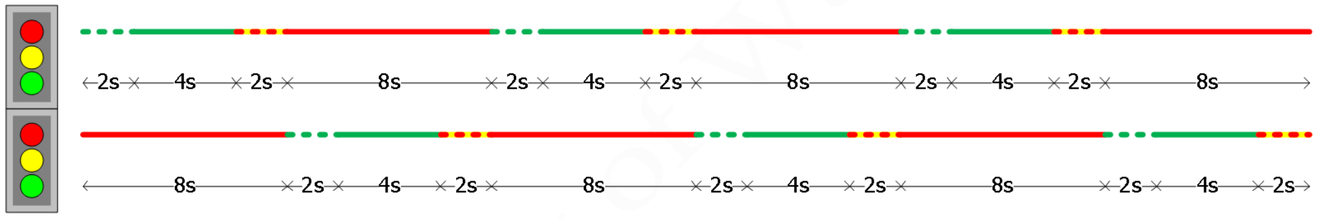


Figure 14 Traffic Light Controller Timing Diagram

The inputs and outputs for the extended traffic light logic are shown below:

Table 7 TLC circuit IO definition

Signal Type	Signal Name	Assigned Port	Description
Input	Input Clock	CLOCK_50	50MHz on-board clock
Outputs	Green LED NS	LEDG[8]	Green light pattern for north/south bounds
	Red LED NS	LEDR[11]	Red light pattern for north/south bounds
	Green LED EW	LEDG[7]	Green light pattern for east/west bounds
	Red LED EW	LEDR[0]	Green light pattern for east/west bounds
	Clock 1Hz	LEDG[1]	1Hz output from modulus counter
	Clock 10Hz	LEDG[0]	10Hz output from modulus counter
	State Number	HEX0	4-bit internal state number
	State Counter	HEX2	4-bit internal state counter

Consider the following guidelines in your design. A typical block diagram for your design is shown in Figure 12.

- If your **PROCESS** has nested **IF** statements then the resulting hardware will be a disaster and hard to debug! Each **IF** statement builds a 2x1 multiplexer and nesting **IF** statements builds a circuit that is deep and slow at the least. For this circuit you never need more than a single **IF** statement.
- Flip-flops are logic elements which can store information. In the sample VHDL code you see a **PROCESS** statement. This is used to create flip flops so that information can be stored in memory or a counter. The D-type flip-flop is the one primarily used in FPGAs.
- The **PROCESS** statement, as given in the sample VHDL code, must be used to build a counter or register. Anything within the clock edge detection statement **IF (rising_edge clock) THEN** is automatically latched. So **A <= B** will automatically latch a signal A, which is set equal to B at the rising edge of the clock.
- Be sure that the sensitivity list **includes all signals 'read' by a process** or simulation will not work as expected.
- Try to minimize the number of states to minimum in your design.
- Note that your Quartus simulation has to employ the 50MHz provided by the board (without using the clock divisor). Your board test and demo should employ the divided (slow) clock so that you are able to see your design running on the board.
- Display internal state number and transition counter. They are in great help for debugging your design.

5.2.3 Simulation

In order to simulate your design before programming the FPGA device, you should change the terminal count value for your first modulus counter in clock chain (10Hz) to skip from waiting for 5,000,000 clock cycles in your simulations to generate one cycle of 10Hz clock from 50MHz. You can set your terminal count to "000000000000000000000001" and then find a proper clock period for CLOCK_50 to have exactly 10Hz and 1Hz clocks in simulation waveforms (Figure 14).

Another way of simulating your design is completely bypassing the clock divider and providing the 50MHz clock to the state machine. Please notice that this strategy is only to simulate your design.

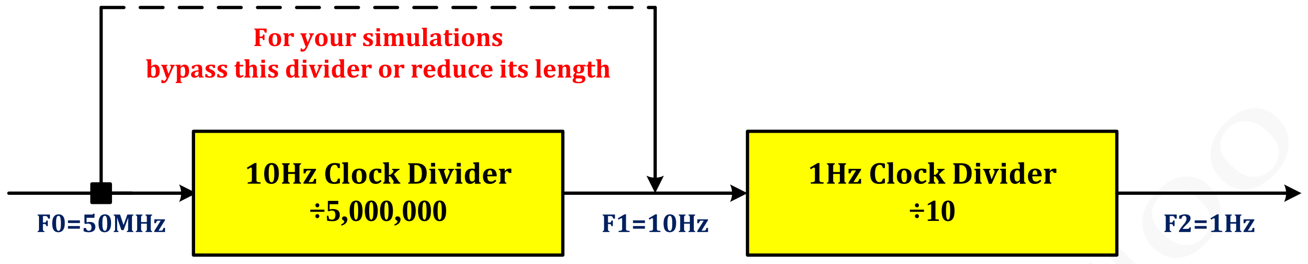


Figure 14 Clock generation scheme

5.3 Postlab

Download, print and fill out the "Lab4SubmissionForm.pdf" form and demonstrate your design on scheduled date. Save your work in a separate project file, you will need it again in lab-5.

6 Lab 5 – Sequential circuits; Advanced TLC – VHDL design

This lab is an extension to the hardware that you developed in lab-4. If you imagine the traffic light controller developed in lab-4 was for normal operation, let's call it 'Day mode', then in this lab session you will add a 'Night mode' to the system that somehow gives priority to one direction (North-South or East-West).

6.1 Prelab

Think of adding the night mode in a way that you employ all circuit developed in lab-4 without duplicating the hardware area. In other words, do **not** implement it like this:

```
If day_mode then

    The code you developed in lab-4

Else -- Night-mode

    The code you developed in lab-4 modified to work in night-mode

End if ;
```

Instead, think of methods that distinguish day-mode from night-mode when the time comes to switch state. In this method the same hardware is used in both day and night modes.

6.2 Lab requirement

In the night mode the traffic light controller has a default (priority) side (SW[16]) that traffic light is always green for it if there is no car on the other side. When the time to switch lights (amber to red) reaches (at the end of amber light period), if no car is detected on the non-default side, the system starts another green-amber (only solid green) period for the default side, otherwise it acts like day mode with green-amber-red periods for both sides. For this mode, we use car detection sensors output for non-default side to decide for transition between states. These sensors can be implemented using ON/OFF switches (SW[15:14]) on the board. When the switch is ON, it indicates the presence of a car on that side. Figure 15 compares the two modes of day and night.

The inputs and outputs for the extended traffic light logic are shown in Table 8.

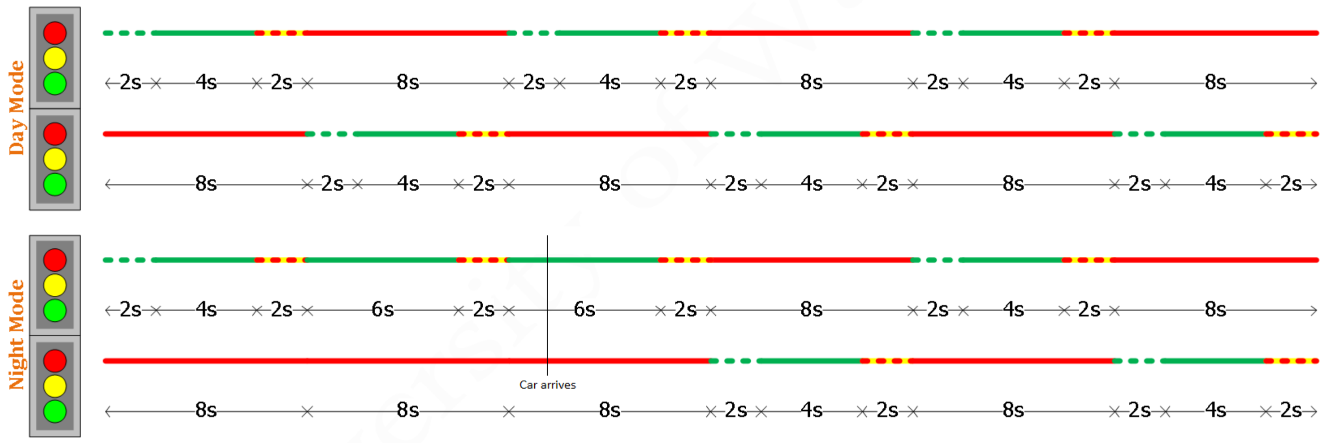


Figure 15 – Traffic Light Controller Timing Diagram

Table 8 – TLC circuit IO definition

Signal Type	Signal Name	Assigned Port	Description
Inputs	Input Clock	CLOCK_50	50MHz on-board clock
	Operation Mode	SW[17]	‘0’: day, ‘1’: night
	Default Side	SW[16]	‘0’: NS, ‘1’: EW
	Car Sensor NS	SW[15]	Car detection sensor output for NS bound
	Car Sensor EW	SW[14]	Car detection sensor output for EW bound
Outputs	Green LED NS	LEDG[8]	Green light pattern for north/south bounds
	Red LED NS	LEDR[11]	Red light pattern for north/south bounds
	Green LED EW	LEDG[7]	Green light pattern for east/west bounds
	Red LED EW	LEDR[0]	Green light pattern for east/west bounds
	State Number	HEX0	4-bit internal state number
	State Counter	HEX2	4-bit internal state counter
	Wait Counter NS	HEX4	4-bit internal wait counter for NS bounds
	Wait Counter EW	HEX6	4-bit internal wait counter for EW bounds

Consider the following guidelines in your design.

- Worst approach: Duplicating the circuit area by constructing a hardware that only works at day mode, and a hardware that only works at night mode.
- Incorporate the night mode with your current lab-4 design. The number of states need not be changed. Only the transition conditions are required to be changed.
- The wait-counter increments (counts up starting 0h) only if the light is red for the corresponding side and car-detection sensor is on for the same side otherwise it **must** be reset to 0h on the display.
- The state-counter is reset to 0h once the system enters a new state. In other words, it only increments within a state, and resets to zero in each new state.
- Display internal state number and transition counter. They are in great help for debugging your design.

6.3 Simulation

Similar to lab-4, you will need to change the VHDL code slightly in order to do the simulation. Follow the same procedure and generate a simulation waveform that proves the circuit is working properly in night-mode as well as day-mode.

6.4 Postlab

Use the "Lab5SubmissionForm.pdf" form to demonstrate your design on scheduled date. Then hand in the report, one day after demo session. The submitted design must include:

1. Completed "Lab5 Submission Form" as the front page of your report. Don't forget to fill out the "Total logic elements" and "Worst Case Speed Parameters" in the form for demonstrated circuit.
2. Implementation procedure, design decisions, state machine diagram with transition conditions, encountered problems or bugs with solution to them, debugging techniques and RTL and State view of your circuit (4 pages max). Use **Tools->Netlist Viewers->State Machine Viewer.** It reveals all registers and states within a circuit.
3. Fully commented VHDL code printout (2-up, landscape and double sided). **Do not include** the **SevenSegment** design (ENTITY+ARCHITECTURE) in your printouts.
4. Functional Simulation Waveforms: Simulation must be done to prove that the design works as desired. You need to show that both night mode and day mode work as desired. Also you need to show the different scenarios that can happen in night mode (different default sides and different car sensor values) using numerous waveforms. Finally the wait counters functionality should also be shown. Please try to cover and explain what happens in your waveforms at different time points and how this is related to the requirements.

7 Appendix I – DE2 pin assignment file

Name	Location
SW[0]	PIN_N25
SW[1]	PIN_N26
SW[2]	PIN_P25
SW[3]	PIN_AE14
SW[4]	PIN_AF14
SW[5]	PIN_AD13
SW[6]	PIN_AC13
SW[7]	PIN_C13
SW[8]	PIN_B13
SW[9]	PIN_A13
SW[10]	PIN_N1
SW[11]	PIN_P1
SW[12]	PIN_P2
SW[13]	PIN_T7
SW[14]	PIN_U3
SW[15]	PIN_U4
SW[16]	PIN_V1
SW[17]	PIN_V2
HEX0[0]	PIN_AF10
HEX0[1]	PIN_AB12
HEX0[2]	PIN_AC12
HEX0[3]	PIN_AD11
HEX0[4]	PIN_AE11
HEX0[5]	PIN_V14
HEX0[6]	PIN_V13
HEX1[0]	PIN_V20
HEX1[1]	PIN_V21
HEX1[2]	PIN_W21
HEX1[3]	PIN_Y22
HEX1[4]	PIN_AA24
HEX1[5]	PIN_AA23
HEX1[6]	PIN_AB24
HEX2[0]	PIN_AB23
HEX2[1]	PIN_V22
HEX2[2]	PIN_AC25
HEX2[3]	PIN_AC26
HEX2[4]	PIN_AB26
HEX2[5]	PIN_AB25
HEX2[6]	PIN_Y24
HEX3[0]	PIN_Y23
HEX3[1]	PIN_AA25
HEX3[2]	PIN_AA26
HEX3[3]	PIN_Y26
HEX3[4]	PIN_Y25
HEX3[5]	PIN_U22
HEX3[6]	PIN_W24
HEX4[0]	PIN_U9
HEX4[1]	PIN_U1
HEX4[2]	PIN_U2
HEX4[3]	PIN_T4
HEX4[4]	PIN_R7
HEX4[5]	PIN_R6
HEX4[6]	PIN_T3
HEX5[0]	PIN_T2
HEX5[1]	PIN_P6
HEX5[2]	PIN_P7
HEX5[3]	PIN_T9
HEX5[4]	PIN_R5
HEX5[5]	PIN_R4
HEX5[6]	PIN_R3
HEX6[0]	PIN_R2
HEX6[1]	PIN_P4
HEX6[2]	PIN_P3
HEX6[3]	PIN_M2
HEX6[4]	PIN_M3
HEX6[5]	PIN_M5
HEX6[6]	PIN_M4
HEX7[0]	PIN_L3
HEX7[1]	PIN_L2
HEX7[2]	PIN_L9
HEX7[3]	PIN_L6
HEX7[4]	PIN_L7
HEX7[5]	PIN_P9
HEX7[6]	PIN_N9
KEY[0]	PIN_G26
KEY[1]	PIN_N23
KEY[2]	PIN_P23
KEY[3]	PIN_W26
LEDR[0]	PIN_AE23
LEDR[1]	PIN_AF23
LEDR[2]	PIN_AB21
LEDR[3]	PIN_AC22
LEDR[4]	PIN_AD22
LEDR[5]	PIN_AD23
LEDR[6]	PIN_AD21
LEDR[7]	PIN_AC21
LEDR[8]	PIN_AA14
LEDR[9]	PIN_Y13
LEDR[10]	PIN_AA13
LEDR[11]	PIN_AC14
LEDR[12]	PIN_AD15
LEDR[13]	PIN_AE15
LEDR[14]	PIN_AF13
LEDR[15]	PIN_AE13
LEDR[16]	PIN_AE12
LEDR[17]	PIN_AD12
LEDG[0]	PIN_AE22
LEDG[1]	PIN_AF22
LEDG[2]	PIN_W19
LEDG[3]	PIN_V18
LEDG[4]	PIN_U18
LEDG[5]	PIN_U17
LEDG[6]	PIN_AA20
LEDG[7]	PIN_Y18
LEDG[8]	PIN_Y12
CLOCK_27	PIN_D13
CLOCK_50	PIN_N2