EE2020: Digital Fundamentals – Laboratories Schedule

SCHEDULE	ACTIVITY TYPE	DESCRIPTION	ASSESSABLE CONTENTS	MODULE WEIGHT
Week 1				
Week 2				
Week 3	Laboratory 1	Getting Started	In lab demonstration. No submission	Laboratory activities and tasks: 15% *The weight distribution is not equal among the laboratory activities. Laboratory 2 and 4 have higher weight*
Week 4				
Week 5	Laboratory 2	Combinational Circuits in Verilog	In lab demonstration. Post-lab report submission	
Week 6	Laboratory 3	Sequential Circuits in Verilog - Part 1	In lab demonstration. No submission	
Recess Week				
Week 7	Laboratory 4	Sequential Circuits in Verilog - Part 2	Post-lab design task. Post-lab report submission	
Week 8	Project 1	Introduction to BASYS3 I/O Devices	Laboratory 4 assessment	
Week 9	Project 2	Basic Features Implementation	Project assessment of introductory contents	5%
Week 10	Project 3	Advanced Features Implementation	Project assessment of basic features	15%
Week 11	Project 4	Full System Testing		
Week 12	Project 5	Project Assessment	Project assessment of advanced features	25%
Week 13				
Reading Week				
Examination	No examinations for EE2020. Two quizzes, each one being 20% of the module weight, are carried out during the semester			

LAB 1

Using a simple Boolean design problem, an introductory approach to the Vivado software used in EE2020 will be covered. Quick instructions on downloading and installing the Vivado software on your personal computer are provided. The Vivado software is provided by Xilinx, which is an industry leader of FPGA. It is a comprehensive integrated development environment (IDE) for FPGA design flow …

LAB 2

A combinational circuit is one where the outputs depend only on the current inputs. In this lab, we will be designing some combinational circuits that are able to perform addition and subtraction …

LAB 3

A sequential circuit is one where the outputs depend on the current inputs and the sequence of past inputs. As a result, a sequential circuit has memory, also called states. In this lab, some basic sequential circuits will be designed to make an LED blink at various speeds …

LAB 4

This lab is a continuation of Lab 3, and requires that you have a good mastery of the all the previous labs. In this Lab 4, more applications of sequential circuits will be explored, and this will be the last lab session before the EE2020 project …

Suggestion: If you are redirected to this page upon opening this PDF document, you may not be able to access the in-built attachments, nor make optimal use of this document. Consider using the freely available Adobe Acrobat Reader DC to access the in-built attachments and its proprietary features. Adobe Acrobat Reader DC is obtainable from: https://get.adobe.com/reader/

NATIONAL UNIVERSITY OF SINGAPORE

Department of Electrical and Computer Engineering

EE2020: DIGITAL FUNDAMENTALS - LAB 1

Getting Started with Vivado 2016.2, Basys 3 Development Board, and Verilog HDL

AY 2016/2017, Semester 2

Teaching Assistants

Christopher M. Shin (Monday EE2020 Session)
Gao Jieyi (Wednesday EE2020 Session)

IMPORTANT NOTES TO REMEMBER THROUGHOUT THE SEMESTER

- Please use the **D:\MyWork** folder for your work in the Digital Electronics Lab. You are encouraged to **delete** all folders within the **D:\MyWork** folder before starting your work. You are further encouraged to **copy** your work folder to your personal flash drive after the session is over.
- Please **delete** your work folder from the laboratory's computers after your session is over. You are responsible to **safeguard** your confidential programs. For assessable programs, you will be penalised if two programs with similarities beyond empirical evidence is detected. Both the source(s) and recipient(s) of plagiarised programs are equally penalised.
- You are free to use your own laptop to work on the lab and project contents in the Digital Electronics Lab.
- Do not work with your work folder and files on your thumb drive in Vivado, as this may significantly reduce your productive time. Work folders and files should preferably be on solid state disks or hard disks for faster accesses and processing.

OVERVIEW

Using a simple Boolean design problem, an introductory approach to the Vivado software used in EE2020 will be covered. Quick instructions on downloading and installing the Vivado software on your personal computer are provided. The Vivado software is provided by Xilinx, which is an industry leader of FPGA. It is a comprehensive integrated development environment (IDE) for FPGA design flow.

In this lab:

- An introduction to basic Verilog HDL (Hardware Description Language) is provided.
- The overall process flow to of designing, synthesising, simulating and implementing a program is covered.
- Programming Digilent's Basys 3 development board, which features an FPGA from Xilinx's state-of-the-art Artix-7 family, is illustrated.

1.0 VIVADO DOWNLOAD AND INSTALLATION

The Vivado 2016.2 software is already installed on the computers in the Digital Electronics Lab, and are ready for immediate usage. It is also required that you install such software on your own personal computer. Some quick guidelines on installing the required software for EE2020 on your personal computer is provided in this section.

Software Name	Vivado Design Suite - HLx Editions - 2016.2 **Warning: Do not use** other versions of the software. Only the Vivado 2016.2 Windows version has been tested. Computer compatibility issues will occur with other versions of the software, and assessment of your project may not be possible. This will lead to loss of project marks if your project cannot be assessed.
Software Weblink	http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2016-2.html
OS Requirement	Windows 7 and above, Linux (Refer to the software documentation). Mac version is not supported
OS Architecture	64 bit versions. 32 bit versions are not supported
Quick Download and Installation Guide	Select the installer that best suits your preferences. All downloads require registration with Xilinx, and you will be prompted for registration during the download:

Vivado HLx 2016.2: Windows Web Installer (EXE - 50.41 MB)
MD5 SUM Value: 31edee9382d432aca96a9fc80befdd40

Installer download takes multiple seconds. Registration is required for installation. Download time during installation may be short, but loss of internet connection may result in installation failure. Use this if you want to minimise the total time from download to installation, as compared to downloading a full installer.

Vivado HLx 2016.2: Linux Web Installer (BIN - 80.67 MB)
MD5 SUM Value: d5607db40f368915052e885a9470d752

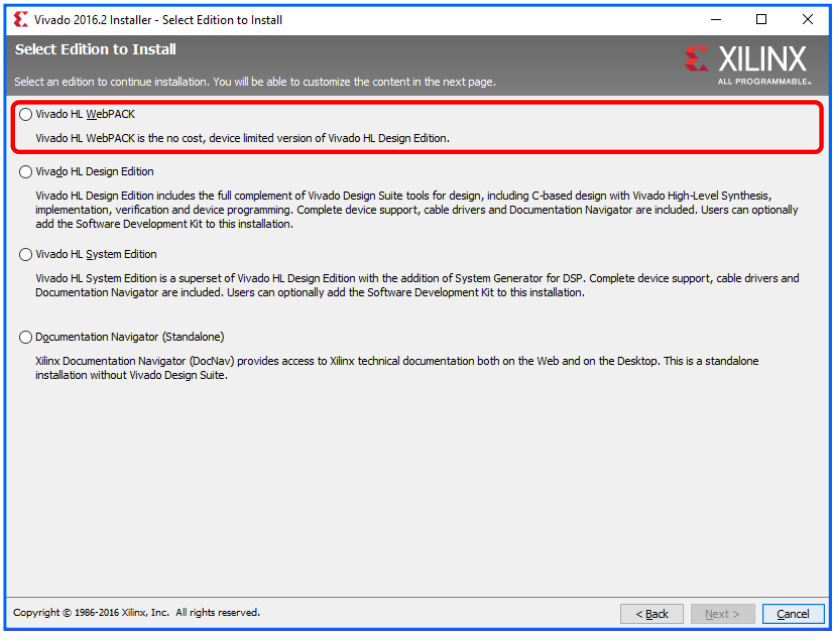
Linux version has not been tested in EE2020. Use at your own risk

Vivado HLx 2016.2: All OS Installer Single-File Download (TAR/GZIP - 11.17 GB)
MD5 SUM Value: 0e41f991e5d89410ad5ed6d30407f379

Installer download takes time, or can be obtained from alternate sources. Registration is not required for installation. There is no download during installation and it takes a few minutes to install, while being the most reliable method for installation

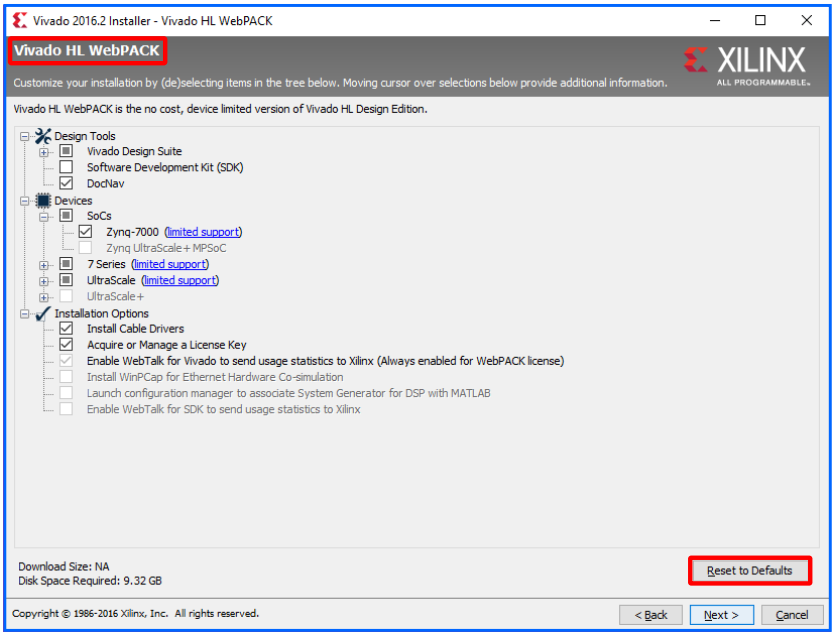
After obtaining the installer file, click on the setup executable of the software. Along the installation process, you may wish to take note of the following:

▶ Select the **Vivado HL WebPACK** option when given the choice of installation edition.



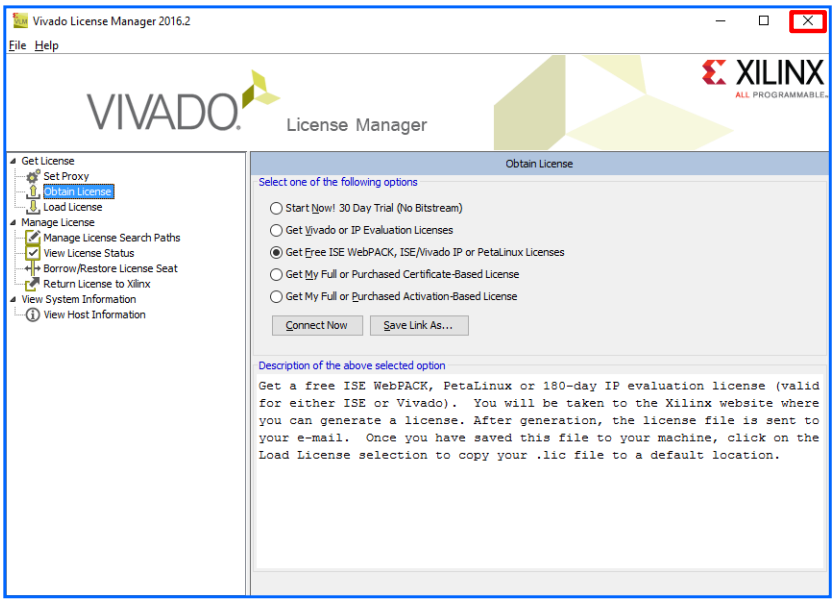
Quick Download and Installation Guide

▶ Select the default options during installation. There is no need to change the installation options.



▶ Your firewall software may prompt you to allow the installation of certain devices. The verified publishers should be allowed access.

▶ After the installation is complete, a license window will appear. You can safely close the window without selecting any license.



Please restart your computer before using the Vivado 2016.2 software. You may wish to uninstall the **Xilinx Information Centre** from the control panel as it is not needed. Not uninstalling it will cause pop-up messages to appear whenever updates for Xilinx are available. You are now ready to use the Vivado 2016.2 IDE for EE2020 purposes.

2.0 DESCRIPTION OF THE SIMPLE BOOLEAN DESIGN TASK

The following task is required to be implemented on the Basys 3 development board:

- When switch **A** turns on, only **LED1** lights up.
- When switch **B** turns on, only **LED2** lights up.
- When both switches **A** and **B** turn on, **LED1**, **LED2**, and **LED3** light up.

UNDERSTANDING | TASK 1

Complete the truth table for the design task:

INPUT		OUTPUT			MINTERM
A	B	LED1	LED2	LED3	
0	0				$\bar{A}\bar{B}$
0	1				$\bar{A}B$
1	0				$A\bar{B}$
1	1				AB

2.1 Deriving an SOP Boolean Equation for the Design Task

Given any truth table with any number of input variables, the sum-of-products (SOP) or product-of-sums (POS) canonical form may be used to write out a Boolean equation for each output variable. Let us use the SOP form as it leads to a shorter equation, albeit not necessarily the simplest.

A minterm, which is a product involving all input variables, that is true for each row of the table has been provided for the truth table. Computing a minterm does not require any consideration of the output. The canonical SOP Boolean equation for an output variable can then be worked out, by summing up each minterm for which the output is TRUE. For example, the canonical SOP form for **LED1** is written by summing up each minterm for which the output **LED1** is TRUE, as indicated below:

$$\mathbf{LED1} = A\bar{B} + AB$$

UNDERSTANDING | TASK 2

Work out the canonical SOP Boolean equations for **LED2** and **LED3** based on your truth table from **UNDERSTANDING | TASK 1**

$$\mathbf{LED2} =$$

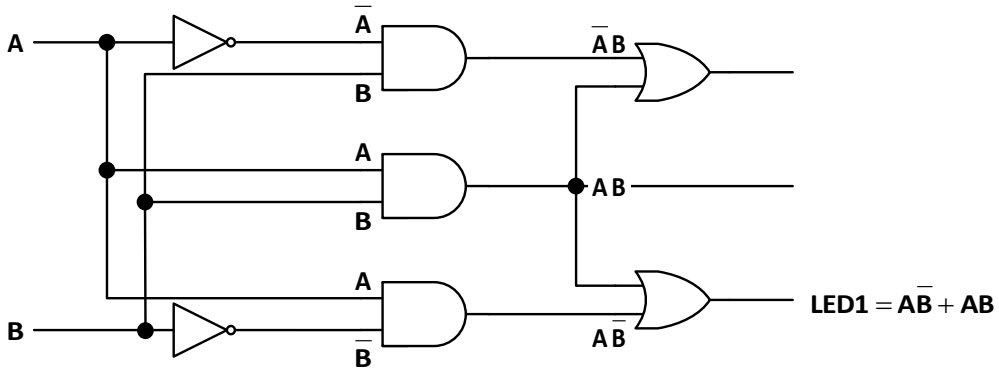
$$\mathbf{LED3} =$$

2.3 Illustrating Logic Expressions by Using a Schematic of Gates

The Boolean equations for **LED1**, **LED2**, and **LED3**, as obtained from section 2.1, can be implement by using:

- 2 NOT gates (NOT gates are also called inverters)
- 3 AND gates
- 2 OR gates

The gate-level schematic is as depicted below:



2.4 Verilog Hardware Description and FPGA Implementation

Xilinx's Vivado software is an integrated design environment that has numerous amount of advanced features used in the industry, and among which we will be introducing the following:

- Writing and editing HDL codes for digital system designs.
- Simulation of the design's behaviour.
- Synthesis of the codes, in order to convert the design from textual description into logic gates.
- Implementation of the design to map and route the logic to a target FPGA.
- Optimising the synthesis, implementation, and bitstream generation according to the user's strategies. The default optimisation strategies shall be used in EE2020, as changing them is beyond the scope of introductory digital designs.
- Programming an FPGA with the optimised bitstream.

The remaining part of this lab manual will now show the steps required to go from the design task, to the FPGA implementation. The Basys 3 development board shall be used, and it is expected that students train themselves to be able to easily go through all these steps before the EE2020 lab 2 starts.

3.0 INTRODUCTORY STEP-BY-STEP GUIDE TO XILINX'S VIVADO 2016.2 SOFTWARE

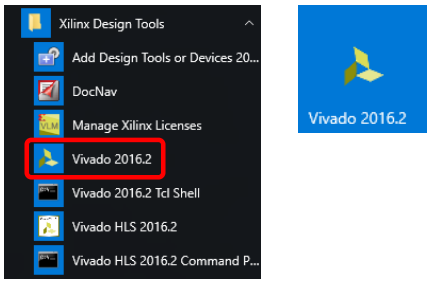
During your lab session, your EE2020 lab assistant may provide you helpful hints on the usage of the Vivado 2016.2 software, beyond the most basic things that are described in this section.

3.1 Creating a New Verilog Project in Vivado

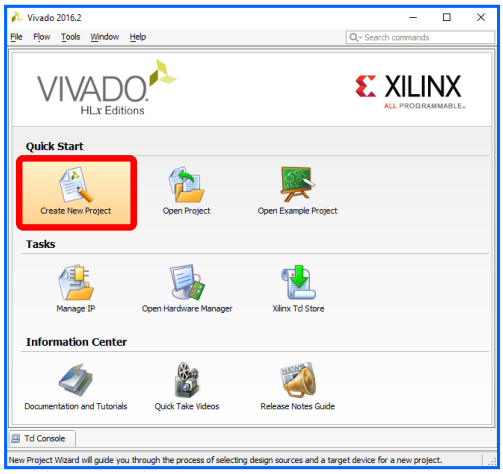
1. Open the executable: **Vivado 2016.2**

   You will need to wait multiple seconds before the program opens. Avoid clicking on the icon multiple times, as errors while using the program may occur if multiple copies of Vivado 2016.2 are opened at the same time.

   If you have used the default installation, and depending on your operating system, the icon may look similar to what is shown to the right.

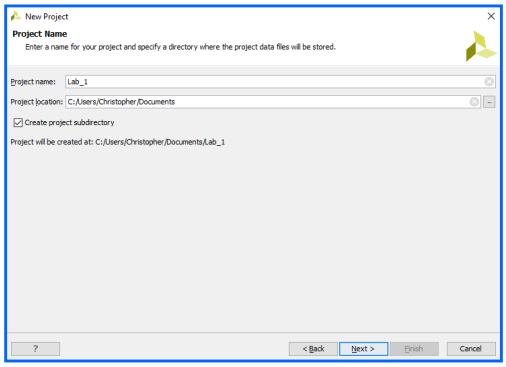


2. **Create New Project** and continue until the next step.



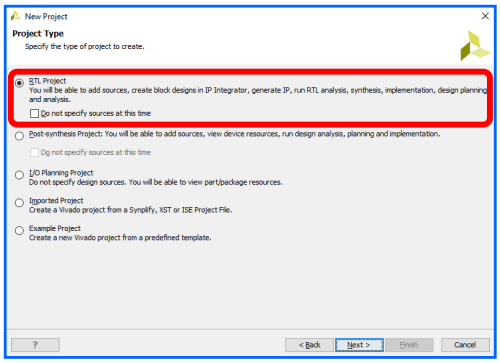
3. Enter a **Project name** and continue.

   Ensure that the complete **Project location** name for your project folder does not have any spaces, and that your **Project name** does not start with a number. Failure to follow these rules will prevent successful designs in Vivado



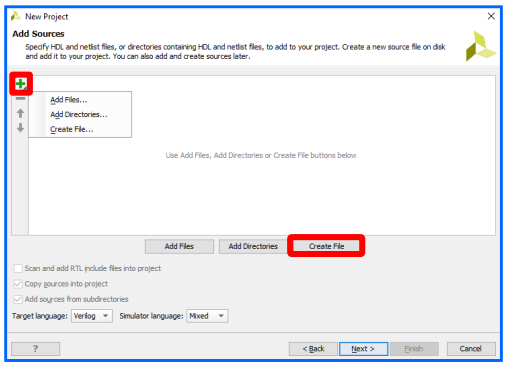
4. Select **RTL Project**.

   Uncheck "**Do not specify sources at this time**".



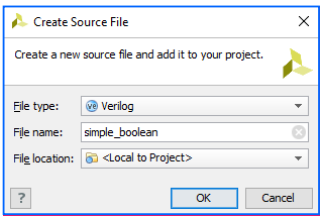
5. Click on **Create File**.

   Target language: **Verilog**
   Simulator language: **Mixed**



6. Enter a **File name** for the source file.

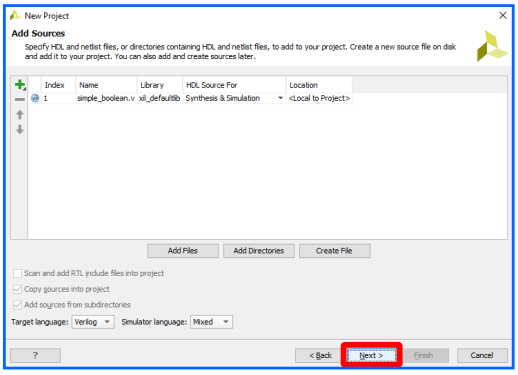
   Ensure that the **File name** does not have any spaces and does not start with a number.



7. In the **Add Sources** window, click on **Next** after the file has been added. This leads to the **Add Existing IP (optional)** window.

   In the **Add Existing IP (optional)** window, just click on **Next**. This leads to the **Add Constraints (optional)** window.

   In the **Add Constraints (optional)** window, just click on **Next**. This leads to the **Default Part** window.

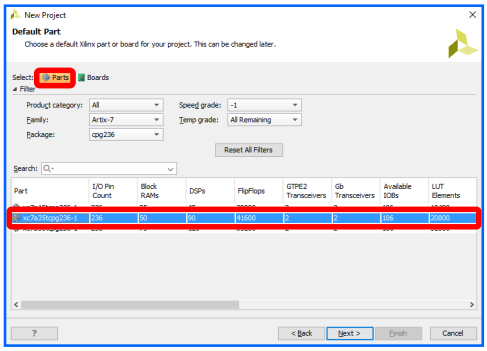


8. Select **Parts**.

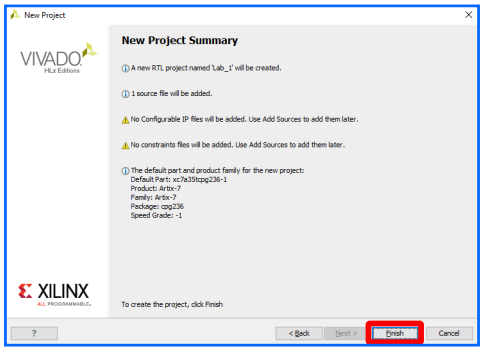
   The FPGA chip in use needs to be specified. The Basys 3 development board contains the **xc7a35t** chip from the Artix-7 family. Use the filter to fasten the search:

   Family: **Artix-7**
   Package: **cpg236**
   Speed grade: **-1**

   Select **xc7a35tcpg236-1** and click on **Next**.



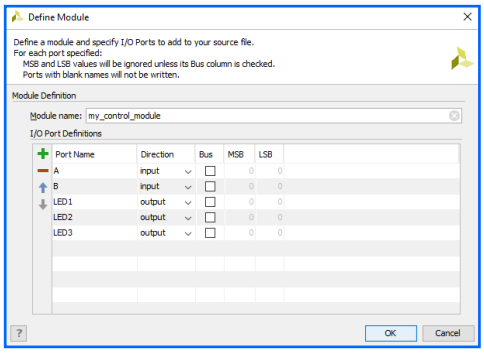
9. Click **Finish**.



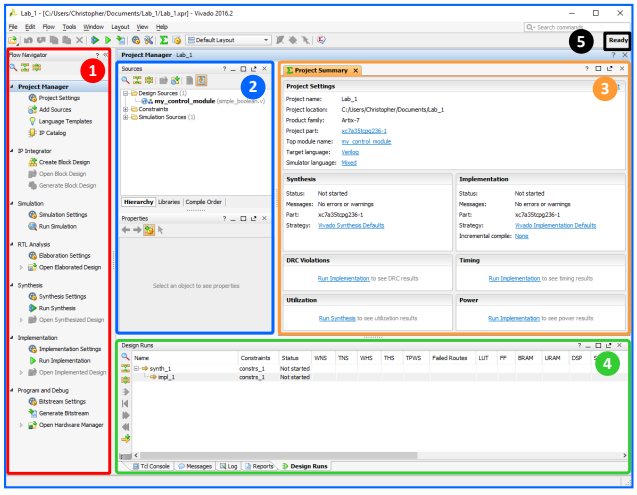
10. A module, that resides within the source file that had been created in the previous steps, need to be created. You may wish to change the Module name.

    Fill in the remaining cells with:

    Port Name: **A**, Direction: **input**
    Port Name: **B**, Direction: **input**
    Port Name: **LED1**, Direction: **output**
    Port Name: **LED2**, Direction: **output**
    Port Name: **LED3**, Direction: **output**



11. Upon creating a new project, the Vivado IDE viewing environment is shown. The main parts of the graphical user interface are:



1. The **Flow Navigator** provides quick access to tools from design entry to bitstream generation.
2. The **Data Windows Area** displays design sources and data. In the screenshot, the **Sources** window is currently selected.
3. The **Workspace** displays the text editor, schematic, project summary, and other report.
4. The **Results Window** displays messages and log files during simulation, synthesis, implementation, and so on.
5. The **Project Status Bar** displays the current status of the active design.

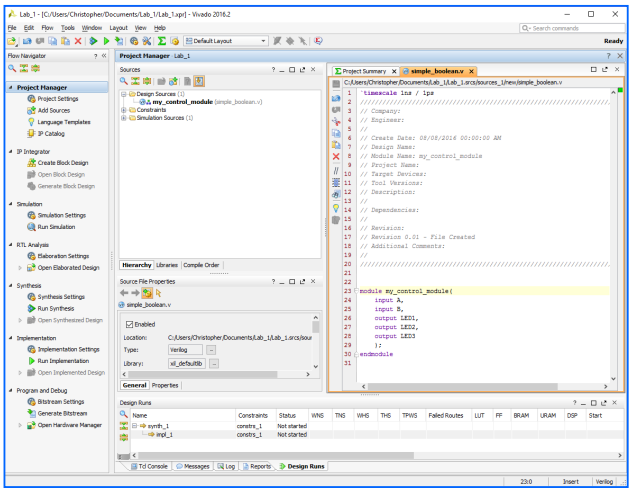
3.2 Using Vivado Text Editor to Write Verilog HDL Code

1. Double click on the design source, for example: **my_control_module**, to open and display the file in the **Workspace** for editing.

   In the text editor, it can be seen that a Verilog module begins with the module name, and the inputs and outputs to the design.

   Words that are highlighted in **purple** are reserved words that have specific functions, and should not be used as labels.

   Currently, the module is empty in terms of the types of outputs to produce when given certain inputs.



2. Some codes for the behaviour of the program need to be inserted, between **module** and **endmodule**. Using the Boolean expressions obtained in section **2.1**, the operators are translated to Verilog representations, as tabulated below:

Operators		Verilog Representation
OR	$A + B$	|
AND	AB	&
NOT	$\overline{A}$	~
XOR	$A \oplus B$	^

In the screenshot, the **assign** statement causes the left hand side of the expression to be updated *every* time there is a change on the right hand side of the expression. It is therefore called a *continuous assignment* statement, describing combinational logic whereby the output on the left is a function of current inputs on the right.

For the image shown on the right, it means that the statements on line 31 till line 33 execute concurrently. This is in contrast to sequential execution of statements in a computer programming language such as C.

```
23 module my_control_module(
24     input A,
25     input B,
26     output LED1,
27     output LED2,
28     output LED3
29     );
30
31     assign LED1 = (A & ~B) | (A & B);
32     // Delete this comment and write the Verilog code for LED2
33     // Delete this comment and write the Verilog code for LED3
34
35 endmodule
```

Code Templates

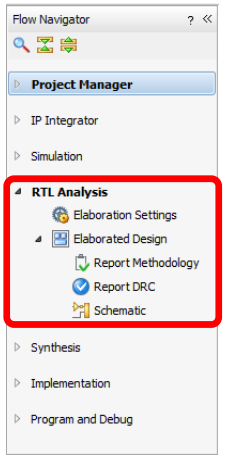
UNDERSTANDING | TASK 3

Complete the code for line 32 and line 33 based on your Boolean expressions from the section **2.1**

3. Save your current file by clicking on **File → Save File**, or by pressing **Ctrl+S**.

   Each time a file is saved, a syntax check is carried out. If there are any syntax errors in your code, they will be indicated in the **Messages** tab of the **Results Window**. Indications also appear in the **Sources** window of the **Data Windows Area**.

   After saving, perform the following:
   - Expand **RTL Analysis** in the **Flow Navigator Panel**
   - Expand **Elaborated Design**
   - Click on **Schematic**
   - If an **Elaborate Design** window pops up, click **OK**



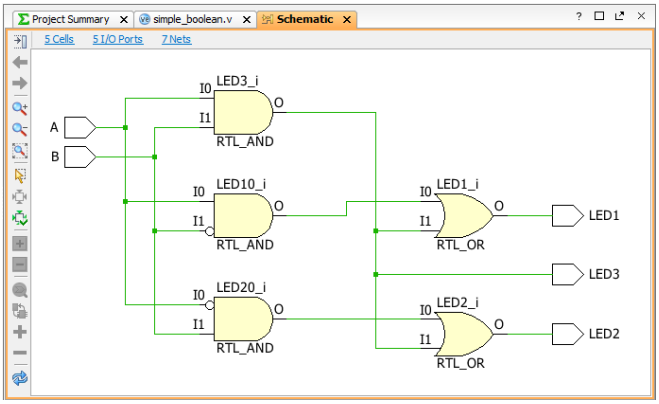
4. The schematics window will appear, showing the RTL schematic of the design.

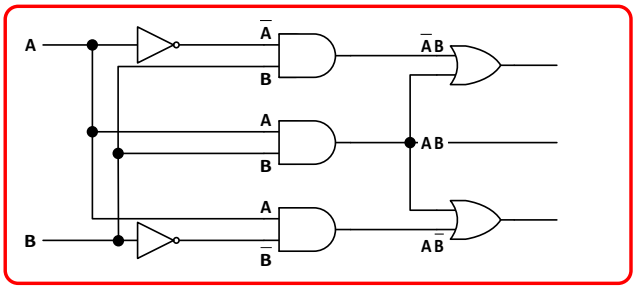
   Register Transfer Level (RTL) schematic is a pre-optimised design in terms of generic symbols, such as AND gates, OR gates.

   In other words, the RTL schematic allows one to view a schematic representation of the design, with symbols such as logic gates, adders, multipliers, counters providing a functional view of the design.

UNDERSTANDING | TASK 4

What similarities and differences do you notice between the RTL schematic and the schematic obtained from the previous section, both shown to the right. How do they compare to the actual schematic obtained on your computer screen?





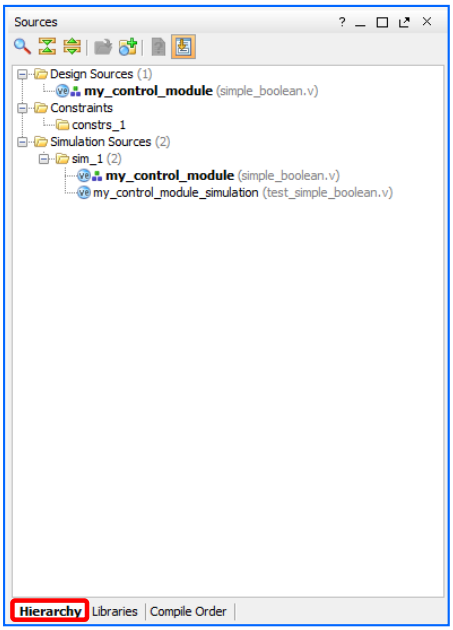
3.3 Testbench and Behavioural Simulation

1. After writing the codes, there is a need to test them to check their behaviours. Inputs are applied to a module, and the outputs are checked to verify whether the module operates as intended. This ensures that a system is tested before it is physically built.

   A testbench is an HDL module that is used to test another module, and which is called the Device Under Test (DUT) or Unit Under Test (UUT). The testbench contains statements to apply inputs to the DUT, and in our case, it is the **my_control_module.**

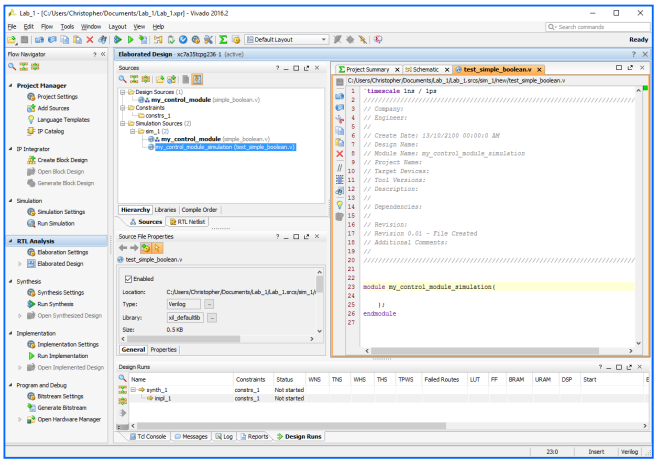
   As a start, perform the following:
   - Expand **Project Manager** in the **Flow Navigator Panel**
   - Click **Add Sources**
   - Select **Add or create simulation sources,** then **Next**
   - Similar to section **3.1**, create a Verilog source file, and give it a file name, such as **test_simple_boolean**
   - In the **Define Module** window, you may change the name to **my_control_module_simulation** if you wish
   - Do not input any **I/O Port Definition**, and click on **OK**
   - The **Sources** window will appear as depicted on the right

   You may click on the **Libraries** and **Compile Order** tab to see how the **Sources** window look like. This will help you in the future if ever the **Hierarchy** tab is not selected by default. Furthermore, understand the relationship between the different file names and module names that have been used here.



2. Double-click on the **my_control_module_simulation** from the **Sources** window to open the **Workspace** text editor for the **test_simple_boolean.v** file.

   The timescale directive at the top of the **test_simple_boolean.v** file specifies the time unit used in the file. **'timescale** 1ns / ps specifies a time unit in the file to be 1 ns, and the simulation to have a precision of 1 ps. Time unit delays are indicated by the symbol **#**



3. Add the codes as shown in the image on the right.

   An **initial** block is used to execute statements in its body once, starting from t=0. For this example, the following happens:
   - At time t = 0 ns, A = 0 and B = 0
   - At time t = 10 ns, A = 0 and B = 1
   - At time t = 20 ns, A = 1 and B = 0
   - At time t = 30 ns, A = 1 and B = 1

   An **initial** statement should only be used in testbenches for simulation, not in modules intended to be synthesised into actual hardware.

   With reference to the screenshot, all signals on the left of assignments must be declared as **reg**. Note that **reg** does not necessarily imply that the signal is a register.

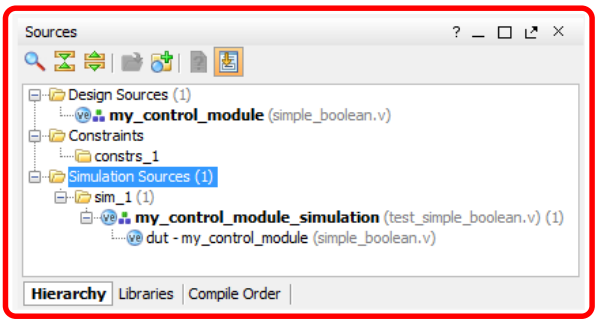
   The **LED1**, **LED2** and **LED3** outputs have been declared as **wire** here. **wire** is used when an output is expected to be used as input to another module, or in an assignment statement. In our case, since the outputs are not used by other modules, the simulation will succeed even without the **wire** declarations for **LED1**, **LED2** and **LED3**.

```
23 module my_control_module_simulation(
24
25     );
26
27     // Inputs
28     reg A;
29     reg B;
30
31     // Outputs
32     wire LED1;
33     wire LED2;
34     wire LED3;
35
36     // Instantiate Device under Test (DUT)
37     my_control_module dut(A, B, LED1, LED2, LED3);
38
39     //Stimuli
40     initial begin
41         A = 0; B = 0; #10;
42         A = 0; B = 1; #10;
43         A = 1; B = 0; #10;
44         A = 1; B = 1; #10;
45     end
46
47 endmodule
48
```

Code Templates

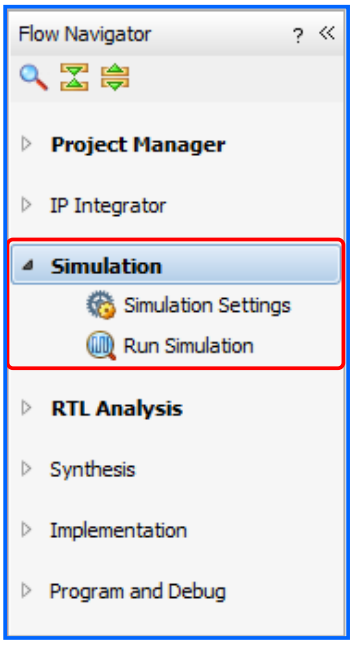
4. After saving the file, note how the **Sources** window has changed. Testbench **my_control_module_simulation** *instantiates* or *calls* **my_control_module**.

   This causes **my_control_module** to be a leaf node in the hierarchy for **my_control_module_simulation**.



5. Perform the following:
   - Expand **Simulation** in the **Flow Navigator Panel**
   - Click on **Run Simulation**
   - Select **Run Behavioral Simulation**

   Several seconds later, the vivado simulator will provide a waveform window.

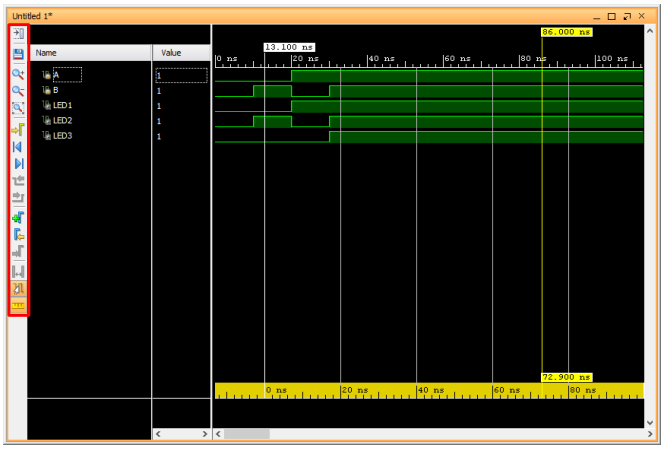


6. A noticeable waveform pattern may not be seen by default, as the time resolution used in the simulation is very small as compared to the amount of time the simulation is ran.

   Hence, with the simulation windows being the active window and from the menu, select **View → Zoom Fit**, or press **Ctrl+0**

   Look at the simulation results closely. How do the waveforms show that your design is indeed working as desired?

   Consider trying out the various options provided in the simulation window before going back to the **Workspace**.

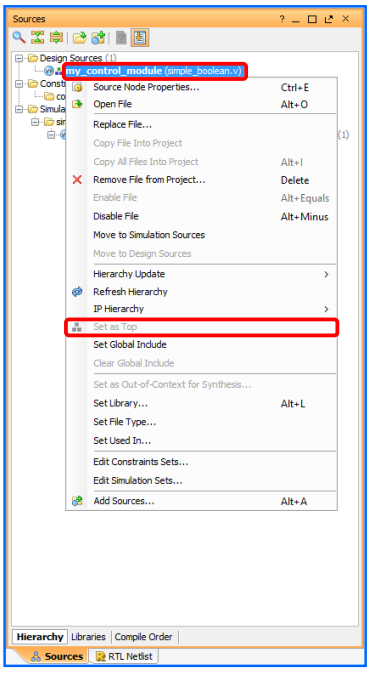


3.4 Synthesis

1. Logic synthesis transforms HDL code into an optimised set of logic gates to reduce the amount of hardware and efficiently perform the intended function. Different optimisation strategies are available, but this will not be the focus of this course.

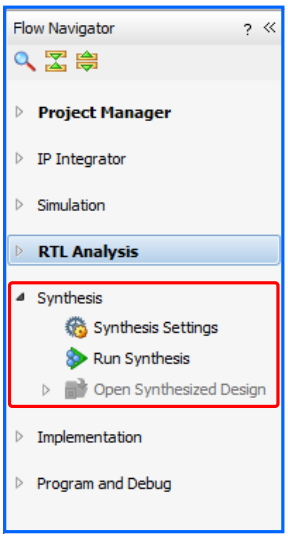
   Right-click on the **simple_boolean.v** file in the **Sources** window, and select **Set as Top**. This option is disabled if the file is already the top module, and in such a case, proceed directly to the next step.

   In general, when there are multiple design and simulation modules, the "Set as Top" option selects the design and simulation modules to be considered when performing the different stages of the project flow. Simply right-click on the module that you wish to process, and select the "Set as Top" option.



2. Perform the following:
   - Expand **Synthesis** in the **Flow Navigator Panel**
   - Click on **Run Synthesis**

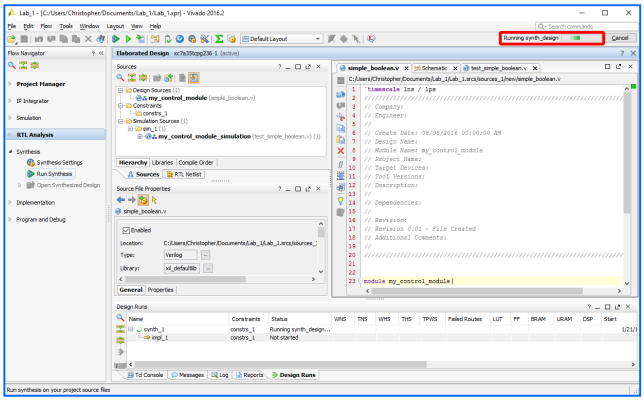
   Synthesis is usually one the most time-consuming part of the FPGA design flow. However, for this very simple example, it should take only multiple seconds.



3. While Vivado performs synthesis, the **Project Status Bar** at the top right provides an indication of the ongoing progress.

   Upon completion of the synthesis, the progress will change from **Running synth_design** to **Synthesis Complete**. A log will appear in the **Log** tab of the **Results Window**, while warnings and errors are displayed in the **Messages** tab.

   If a **Synthesis Completed: Synthesis successfully completed** windows appears, you may close it.



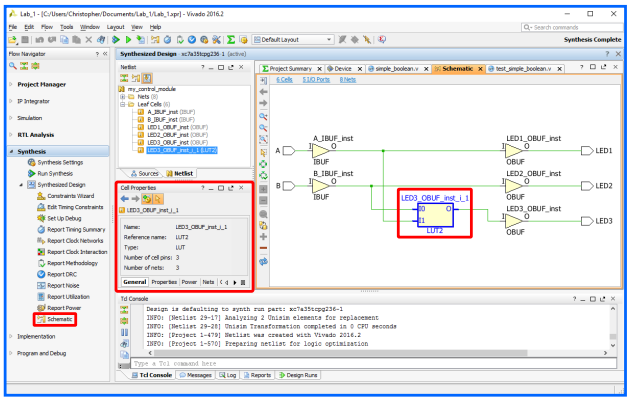
4. Perform the following:
   - Expand **Synthesis** in the **Flow Navigator Panel**
   - Expand **Synthesized Design** under **Synthesis**
   - Click on **Schematic**

   The schematic of the synthesised design will be generated and this synthesised circuit is an optimised version of the RTL schematic that was obtained in section **3.2**

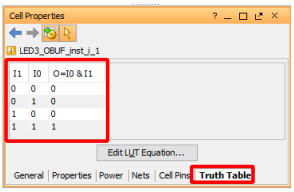
   The IBUF or OBUF symbols represent buffers. A buffer behaves like an inverter without its bubble, whereby its output is equivalent to its input.

   Click on the Look-up Table (LUT) that defines how the output **LED3** behaves. The **Cell Properties** window will appear for that specific LUT.

   Note that Field-Programmable Gate Arrays (FPGAs) consists of large amount of LUTs that can be configured to behave as multi-input gates.



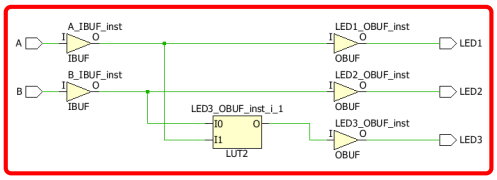
5. In the **Cell Properties** window for the LUT of **LED3**, open the **Truth Table** tab. Notice how for this simple example, this LUT is behaving as a simple AND gate.

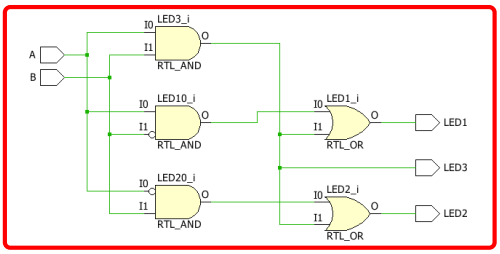


6. Compare the optimised and non-optimised schematics.

UNDERSTANDING | TASK 5

How is this optimised circuit equivalent to the SOP equations of section **2.1**?



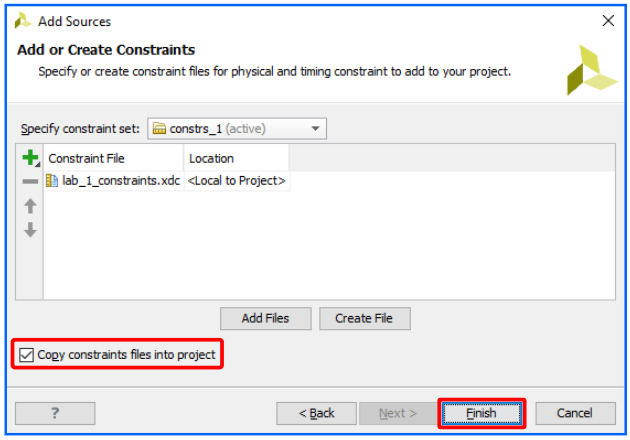


3.5 Design Constraints

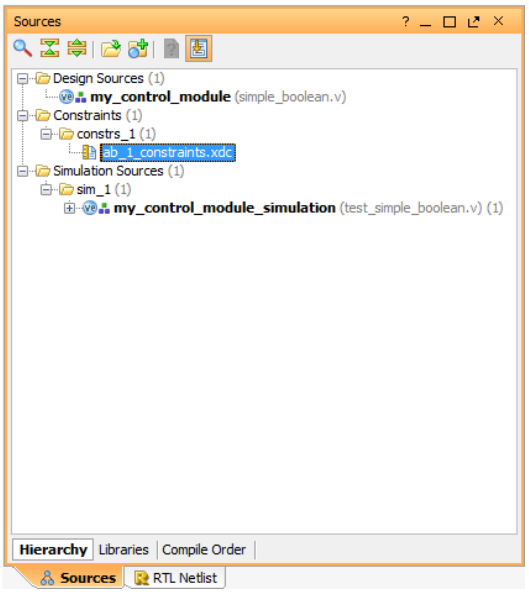
1. Design constraints, such as timing and physical I/O pin mapping, must be defined before doing an implementation. Given that our example is a combinational circuit, timing constraints will not be specified as there are no critical sequential elements in the design.

   Such design constraints may be set any time after writing the HDL code described in section **3.2**, including before the synthesis phase discussed in section **3.4**

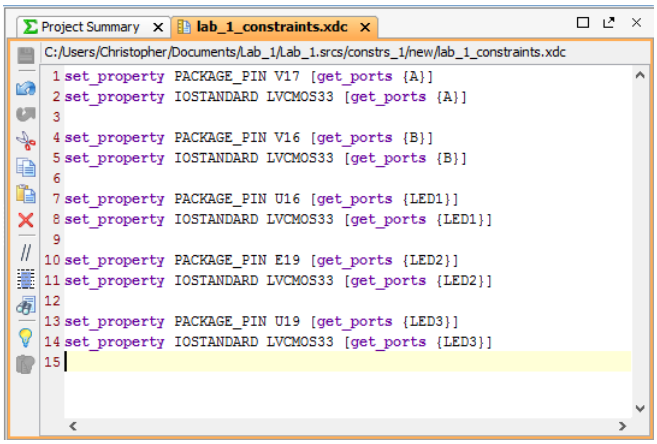
   Proceed with the following sequence:
   - Expand **Project Manager** in the **Flow Navigator Panel**
   - Click **Add Sources**
   - Select **Add or create constraints** and click **Next**
   - Click on **Create File** and give the XDC file a file name, such as **lab_1_constraints**. The XDC format stands for Xilinx Design Constraints here
   - When adding or creating a source file, it is recommended to select the option to copy the file into the project. This ensures that the all the files you are working with is located in one work folder, and not on multiple places on your storage device
   - Finally, click on **Finish**

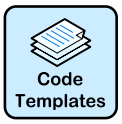


2. Open the **lab_1_constraints.xdc** file from the **Sources** window.

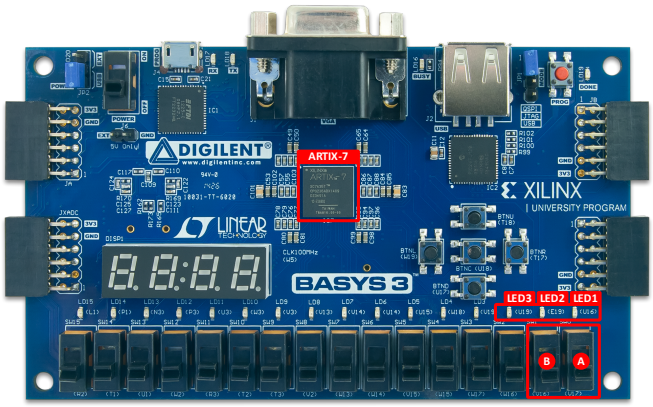


3. Enter the constraints as shown in the screenshot. The statements tell the Artix-7 FPGA how to link the signals in the Verilog top level design source module, to physical I/O pins on the FPGA.





4. The Artix-7 is located around the middle of the Basys 3 development board, and the switches and LEDs being used are located in the bottom right corner.

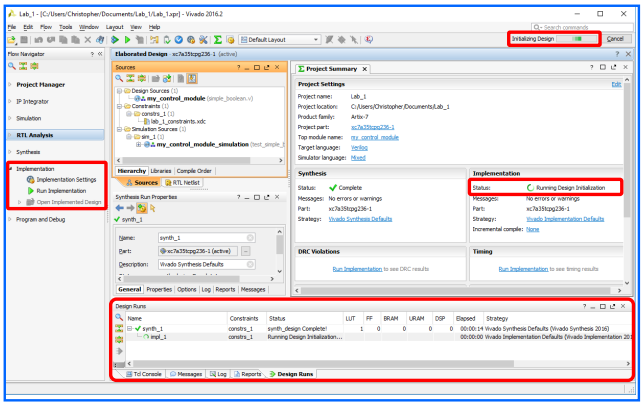


3.6 Implementation and Bitstream Generation

1. Do the following:
   - Expand **Implementation** in the **Flow Navigator Panel**
   - Click **Run Implementation**. You may be prompted and required to perform synthesis again if it is out-of-date due to changes in the HDL code

   Similar to the synthesis phase, the status of the implementation phase can be monitored at multiple places, including in the **Project Summary** window, the top-right corner of the **Project Status Bar**, or the **Design Runs** tab. Monitoring might be useful as implementation is another time-consuming process.

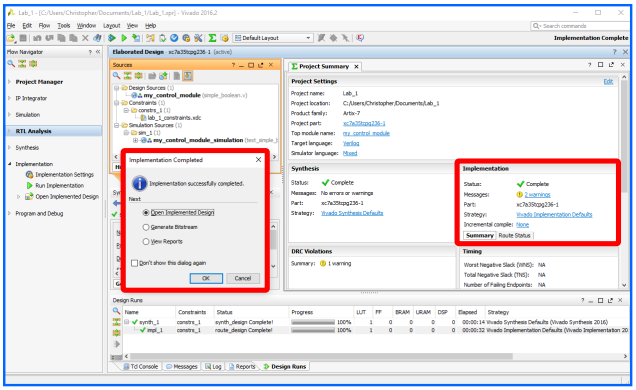
   If there are any warnings or errors, they are displayed in the **Messages** tab.



2. After successful completion of the implementation process, the **Implementation Status** will change to **Complete**.

   Notice that there are warnings but these can be ignored. However, critical warning and errors cannot be ignored, and will lead to failed implementation.

   If an **Implementation Completed** window pops up, you can choose **Generate Bitstream** and click on **OK**, or else you can click on **Cancel**.

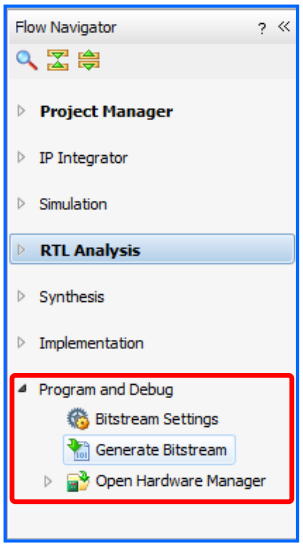


3. After the implementation phase, there is a need to generate a file that can be uploaded to the FPGA. Such a file is called a bitstream file, and it consists of binary values 0's and 1's that tells the FPGA how to behave.

   The bitstream file is obtained by doing:
   - Expand **Program and Debug** in the **Flow Navigator Panel**
   - Click on **Generate Bitstream**

   After waiting for several moments, the bitstream generation will be completed. Monitoring of the bitstream generation is done in the same manner as for the synthesis and implementation phase.

   A successful bitstream generation is the last step required before uploading the program to the Artix-7 FPGA.



3.7 Using the Basys 3 Development Board and Program Upload

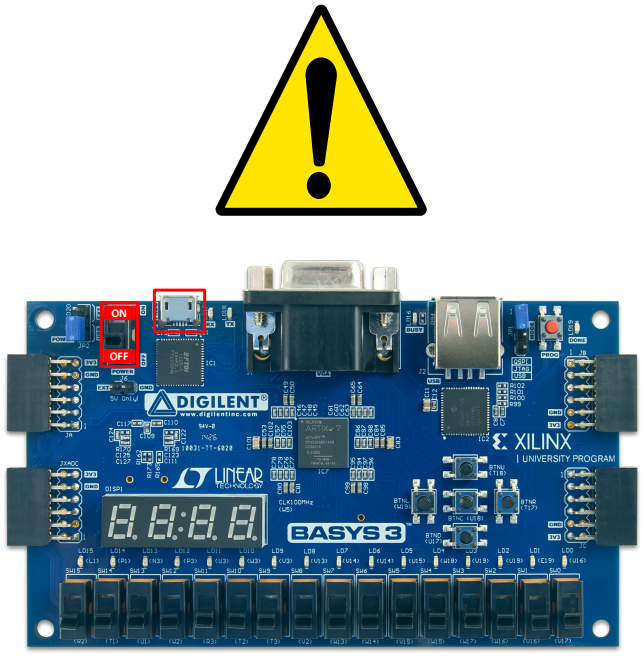
1. Before using your Basys 3 development board, and to prevent potential damage to it, take note of the following recommendations and warnings:

   ⚠ The Basys 3 development board is powered OFF by placing SW16 in the OFF position before connection to/removal from the USB port of the computer.

   ⚠ Do not force in the micro-USB cable upside down to the Basys 3 development board, as this will damage your micro-USB port and device. Carefully connect to the micro-USB cable in the correct orientation.

   ⚠ The chips on the board are electrostatic sensitive. Avoid touching them. Handle the board by the edges to prevent damage.

   ⚠ Make sure the board is not in contact with any metal components, whether above or below. Do not place any liquid sources near the FPGA board.

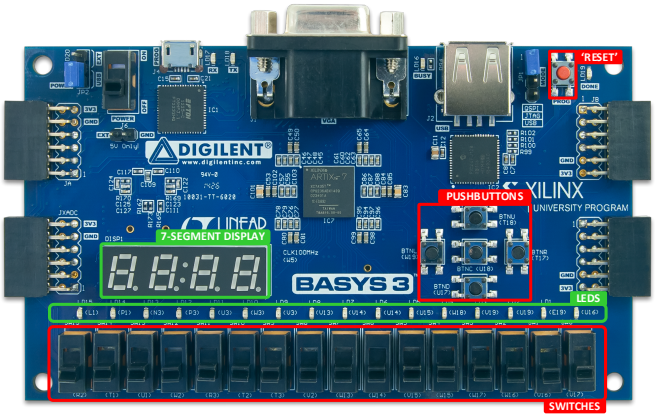


2. After connection of the Basys 3 development board to the computer, turn on the power by setting SW16 in the ON position.

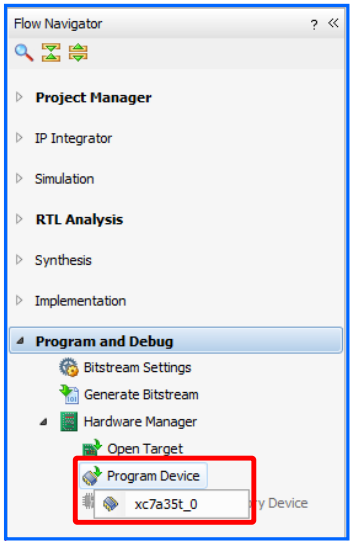
   Each Basys 3 development board has a unique identifier, and if this is the first time you are connecting your device to a specific computer, it will take a few moments for it to be detected and installed for that specific computer. Internet connection and windows device update is recommended to be enabled if your device cannot be installed.

   On faster computers and optimised computer system configurations, the device detection and installation takes a few seconds. In the lab, the desktop computers sometimes take a few minutes for the detection and installation.

   You may test your Basys 3 development board afterwards. Whenever powered up, the Basys 3 on-board flash device loads a pre-configured demo program. Observe the 7-segment display, use the pushbuttons and switches, or try the 'reset' button.

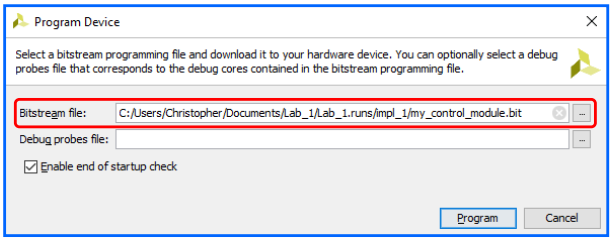


3. After successful connection and installation of your Basys 3 development board to the computer, and verification through the default pre-configured demo program, it is now time to upload your developed program, by doing the following:
   - Expand **Program and Debug** in the **Flow Navigator Panel**
   - Expand **Open Hardware Manager**
   - Click **Open Target**
   - Select **Auto Connect.** In case connection fails, consider pressing the 'reset' button, or turn your device OFF for a few seconds and ON again, while ensuring that it is detected and installed on your computer. Then try **Auto Connect** again.
   - If successful, the **Program Device** will be enabled, and you will be able to select **xc7a35t_0**



4. Clicking on **xc7a35t_0** leads to the window shown on the right. By default, if the bitstream was successfully generated, the path name in the **Bitstream file** is automatically provided.

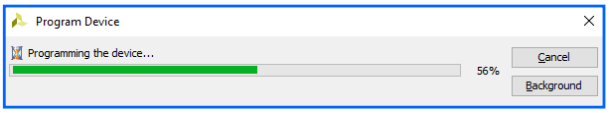
   Upload the .bit program to the FPGA by clicking on **Program**.



5. Your program will then be uploaded within a few seconds.

UNDERSTANDING | TASK 6

Verify the functionality of the design by using the input devices you have assigned to **A**, **B** and observing the output devices assigned to **LED1**, **LED2** and **LED3**. Check what happens if the 'reset' pushbutton is pressed. Once you are confident with the correctness of the uploaded program, please inform your assigned Graduate Assistant (G.A.) for verification and to note down your progress.



4.0 CLOSING NOTES FOR LAB 1

Congratulations on successfully completing your possibly first introductory FPGA design flow **^_^**

The steps provided throughout section **3.0** should have been relatively straightforward for you to complete. They included:

- **3.1 Creating a New Verilog Project in Vivado**
- **3.2 Using Vivado Text Editor to Write Verilog HDL Code**
- **3.3 Testbench and Behavioural Simulation**
- **3.4 Synthesis**
- **3.5 Design Constraints**
- **3.6 Implementation and Bitstream Generation**
- **3.7 Using the Basys 3 Development Board and Program Upload**

Subsequent labs will not be detailed with hints throughout, and shall focus more on the programming aspect and your thinking abilities. Hence, before the upcoming labs, and project, you need to be at ease with all these steps, without referring to this lab 1 manual excessively.

FINAL UNDERSTANDING | TASK FOR LAB 1

Your lab assistant will provide you a task at the end of the lab, and which you can optionally attempt. It will allow you to self-check whether you have understood all the stated steps. There is no need to show this part to your assigned Graduate Assistant (G.A.). It is your responsibility to ensure that you are confident with the steps described in section **3.0** as much as possible, before the upcoming labs.

NATIONAL UNIVERSITY OF SINGAPORE

Department of Electrical and Computer Engineering

EE2020: DIGITAL FUNDAMENTALS - LAB 2

Combinational Circuits in Verilog

AY 2016/2017, Semester 2

Teaching Assistants

Christopher M. Shin (Monday EE2020 Session)
Gao Jieyi (Wednesday EE2020 Session)

OVERVIEW

A combinational circuit is one where the outputs depend only on the current inputs. In this lab, we will be designing some combinational circuits that are able to perform addition and subtraction.

The pre-requisite for this lab requires one to be able to:

- Create a Verilog project and design source in Vivado.
- Create a testbench to simulate the design source.
- Generate the RTL schematic and the synthesised circuit schematic of design source.
- Map and implement a design on the Basys 3 development board.

As multiple designs will be used in this lab, use the "*Set as Top*" option to select the current design and testbench being worked with. The design source that is "*Set as Top*" will be the one selected when performing the different stages of the design.

[Brief notes on how to *"Set as Top"*…]

This lab will cover the following:

- Designing a one-bit full adder circuit using the dataflow modelling method.
- Designing a two-bit parallel adder circuit using the structural modelling method.

Tasks for this lab include:

- Designing the Verilog code of a one-bit full adder.
- Designing, simulating and implementing a four-bit parallel adder on the FPGA.
- Designing and simulating a four-bit subtractor.

To be well prepared for this lab 2, please be confident with lab 1, such as by doing the ***FINAL UNDERSTANDING | TASK FOR LAB 1*** *from scratch (Do not modify existing Verilog modules) without any help and with minimal reference to the lab 1 manual.*

1.0 ONE-BIT FULL ADDER

Consider the binary addition shown in ***Figure 2.1***. To design a circuit that would perform the addition of two one-bit values, the circuit would need to have three input bits and two output bits.

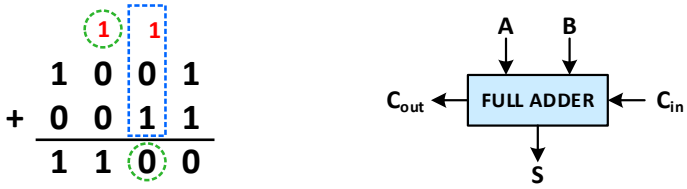


Figure 2.1: *Binary Addition and functional block diagram of the one-bit full adder*

Such a circuit is called a one-bit full adder. It adds two bits (**A**, **B**) and the carry ($\mathbf{C_{in}}$) from a previous stage of addition, and produces a sum (**S**) and a carry ($\mathbf{C_{out}}$), as illustrated through a truth table in ***Figure 2.2***. By simplifying the truth table, the Boolean expressions for **S** and $\mathbf{C_{out}}$ can be obtained.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + C_{in}\,(A \oplus B)$$

Figure 2.2: *Truth table and boolean expressions of the one-bit full adder*

UNDERSTANDING | TASK 1

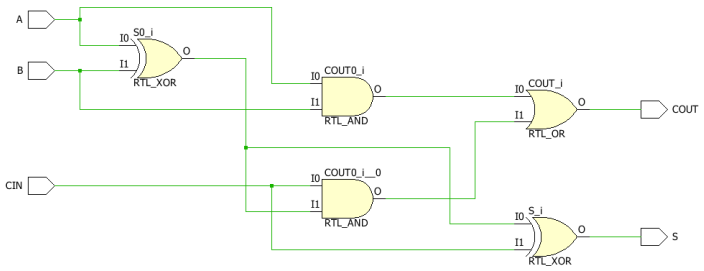
Using the dataflow method, complete the Verilog code for the one-bit full adder. Verify that the RTL schematic is as shown.

Verilog skeleton code for the one-bit-full adder

```
module my_full_adder(input A, B, CIN, output S, COUT);

    assign S =
    assign COUT =

endmodule
```

Code Templates

RTL schematic for the one-bit full adder



Note: A half adder, in contrast to a full adder, does not involve a carry input. Thus, for a half adder, $S = A \oplus B$*, and* $C_{out} = AB$

2.0 TWO-BIT FULL ADDER

By cascading one-bit full adder blocks, the one-bit adder can be reused and parallel adders that add multiple bits can be created through the structural modelling method. A two-bit ripple-carry adder is illustrated in ***Figure 2.3***.

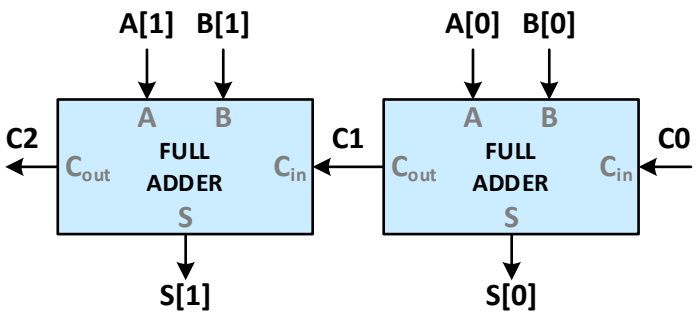


Figure 2.3: *Functional block diagram of the two-bit ripple-carry adder*

With the code for a one-bit full adder available from section **1.0**, a new module is created and two full adder blocks (fa0, fa1) are instantiated. By specifying the inputs and outputs to these blocks, the connection **C1** between them is made. Note that the order of signals during instantiation should respect the order in which they were declared in the one-bit full adder module **my_full_adder**.

This approach to hardware description is called structural modelling, whereby a more abstract module (for example, **my_2_bit_adder**) is built from simpler components describing gate-level hardware (such as **my_full_adder**).

Verilog code for two-bit ripple-carry adder, using structural modelling

```
module my_2_bit_adder(input [1:0] A, input [1:0] B, input C0,
                      output [1:0] S, output C2);

    wire C1;

    my_full_adder fa0 (A[0], B[0], C0, S[0], C1);
    my_full_adder fa1 (A[1], B[1], C1, S[1], C2);

endmodule
```

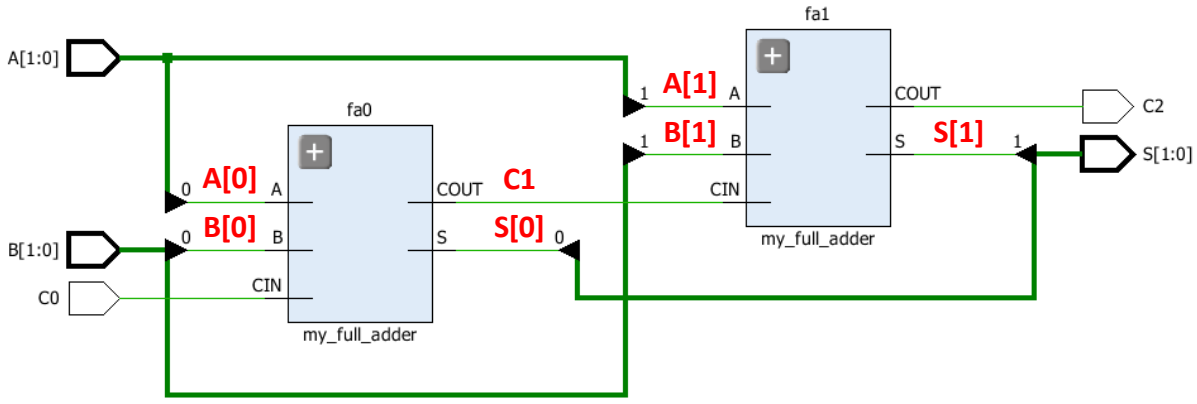
Code Templates

Two-bit port declaration for A, B, S

Instead of having multiple input and output ports (A1, A0, B1, B0, S1, S0), multi-bit vector ports are defined.

`input [5:0] apple` means:
Input port named **apple**, with size of 6 bits. apple[5] refers to the MSB, apple[0] refers to the LSB.

RTL schematic for the two-bit ripple-carry adder



3.0 FOUR-BIT FULL ADDER

The functional block diagram of a four-bit ripple-carry adder is shown in ***Figure 2.4***.

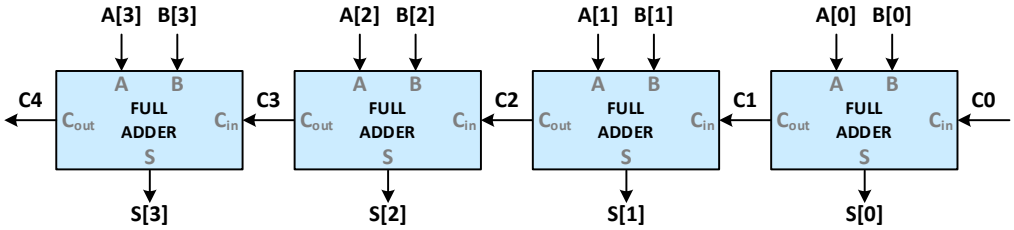


Figure 2.4: *Functional block diagram of a four-bit ripple-carry adder*

UNDERSTANDING | TASK 2

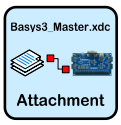
Start by adding a new design source for the four-bit full adder.

1. By using the structural modelling method, design a four-bit ripple-carry adder.
2. Generate the RTL schematic and check that connections between the blocks are correct.
3. Simulate your code with the following three sets of input values, and check that the simulation outputs are correct:

A:		0	0	1	1	A:		1	0	1	1	A:		1	1	1	1
B:	+	0	0	1	1	B:	+	0	1	1	1	B:	+	1	1	1	1
		□	□	□	□			□	□	□	□			□	□	□	□

Brief guidelines for multi-bit vector ports in the testbench

- The port size should be indicated when declaring the signals. Inputs to the *dut* are declared using **reg**, whereas outputs from the *dut* are declared using **wire**:
  `reg [3:0] A; wire [3:0] S;`
- The parameters for the *dut* do not need the port size of the signals:
  `my_4_bit_adder dut (A, B, CARRY_IN, S, CARRY_OUT);`
- The testbench stimuli for multi-bit vector ports can be written as:
  `A = 4'b0011; B = 4'b0011; CARRY_IN = 1'b0;`



4. Synthesise and implement your code on the FPGA, using appropriate switches and LEDs on the Basys 3 development board to represent the inputs and outputs. Consider using the Xilinx's design constraint file template (**Basys3_Master.xdc**) for ease of implementation.

After you have confirmed your design and results to be **correct**, show to your assigned Graduate Assistant (G.A.) **in one go**, the following items related to the four-bit full adder:

- The RTL schematic of your design (item 2 of this task)
- The simulation waveform results of the three testbench stimuli (item 3 of this task)
- The four-bit ripple-carry adder on the Basys 3 development board (item 4 of this task)

UNDERSTANDING | TASK 3

Instead of using four one-bit adder blocks, can you think of an alternative way (still using structural modelling) in creating the four-bit ripple-carry adder? Consider doing the same things as indicated in UNDERSTANDING | TASK 2 by using such alternative way. This task should not be shown to the G.A. and is meant only to improve your Vivado skills and Verilog understanding.

4.0 CLOSING NOTES FOR LAB 2 – THE SUBTRACTOR

FINAL UNDERSTANDING | TASK FOR LAB 2

The final task of this lab is to create a four-bit subtractor, by making use of the four-bit parallel adder that had been created in **UNDERSTANDING | TASK 2**

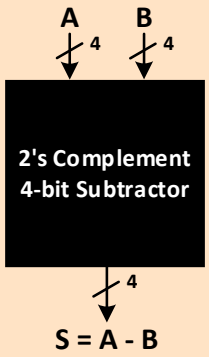
Since $A - B = A + (-B)$, by adding A to $-B$, the answer to $A - B$ can be obtained.

1's or 2's complement can be used to find $-B$ from B.

Explain in a 1 sheet (2 pages) report on how one can design a complete four-bit subtractor circuit. Write the Verilog code to implement the circuit in **EITHER** 1's **OR** 2's complement. Do not do both.

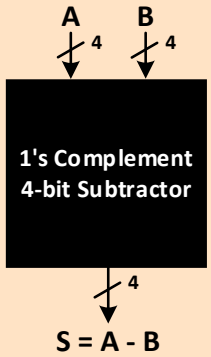
Examples on the expected output S, for some input values of A and B

Two's Complement



A	B	S
0101 (+5)	0001 (+1)	0100 (+4)
0101 (+5)	1111 (−1)	0110 (+6)
1111 (−1)	0101 (+5)	1010 (−6)
1111 (−1)	1011 (−5)	0100 (+4)

One's Complement



A	B	S
0101 (+5)	0001 (+1)	0100 (+4)
0101 (+5)	1110 (−1)	0110 (+6)
1110 (−1)	0101 (+5)	1001 (−6)
1110 (−1)	1010 (−5)	0100 (+4)

Further instructions:

- The addition and subtraction operators (**+**, **−**) are **not allowed** within the code
- As much as possible, make your code efficient and reduce redundancies.
- Improve the subtractor's ease-of-use by considering the user's point of view.

Include the following in the report:

- Name, official EE2020 lab session day, and matriculation number.
- Title: **EITHER** "1's Complement Subtractor" **OR** "2's Complement Subtractor".
- Verilog codes for all the sub-modules used in the subtractor.
- Simulation waveform results with at least the 4 test cases (Choose the set of test cases that corresponds to the complement method that you have used) mentioned in "**Examples on the expected output S, for some input values of A and B**". Ensure that they are all readable in the screenshot.
- Note that the simulation testbench code is not required in the report. Just the simulation waveform results are required.
- You may include any other information and screenshots that you deem necessary for the subtractor.
- G.A.s and T.A.s will not provide additional information on what can be included.
- You are not required to show the FPGA implementation for the subtractor on the Basys 3 development board.

Submission instructions:

- Maximum of two A4 pages (1 sheet).
- Font size can vary between 6 to 20, depending on how much information you wish to fit in.
- Report should be as a softcopy. Hardcopies are not accepted.
- Report in PDF format, with good quality screenshots (readable simulation waveforms, no camera images).
- Naming of the report to follow this template as close as possible:
  Lab2_Official EE2020 lab session day_**Name as indicated on your matriculation card**_Matriculation number_Report
  Examples:
  Lab2_Wednesday_John Doe Jr._A0102020X_Report
  Lab2_Monday_Maurice Agaune_A0131086Y_Report
- Upload to IVLE, in the folder corresponding to your official EE2020 lab session day.
- The IVLE upload must be completed within 7 days of your official EE2020 lab session day. If you have attended other lab session days on a temporary basis, your EE2020 official lab session day is the one considered for the upload deadline.

Feedbacks: (Note that feedbacks, whether positive or negative, DO NOT have any effects on your grades ☺)

We are continually trying to enhance your learning experience. Please include the following in your report:

- What did you like most about the lab sessions and assignments, and why?
- What did you like least about them, and why?
- Were there any sections of the lab manuals that were unclear? If so, what was unclear? Please provide suggestions to improve the clarity.
- What suggestions do you have to improve the overall lab sessions and assignments?

NATIONAL UNIVERSITY OF SINGAPORE

Department of Electrical and Computer Engineering

NUS
National University of Singapore

EE2020: DIGITAL FUNDAMENTALS - LAB 3

Sequential Circuits in Verilog - Part 1

AY 2016/2017, Semester 2

Teaching Assistants

Christopher M. Shin (Monday EE2020 Session)
Gao Jieyi (Wednesday EE2020 Session)

OVERVIEW

A sequential circuit is one where the outputs depend on the current inputs and the sequence of past inputs. As a result, a sequential circuit has memory, also called states. In this lab, some basic sequential circuits will be designed to make an LED blink at various speeds.

The pre-requisites for this lab are:

- A very good understanding of Lab #2 and the various steps involved in designing modules.
- Knowing how to use the Vivado IDE well, as practised during the previous lab sessions.
- Familiarity and knowledge on how to use "*Set as Top*", "*reg*" and "*wire*", as acquired during the previous lab sessions.

This lab will cover the following:

- Introduction to the implementation of a 1-bit two-to-one multiplexer.
- Using a signal that inverts itself periodically, which shall be called **CLOCK**.
- Making a physical LED blink by using the FPGA clock signal.

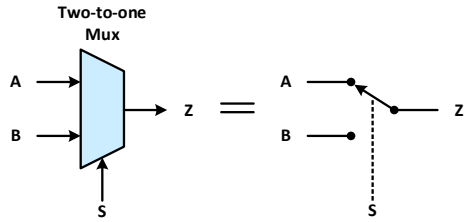
Tasks for this lab include:

- Understanding a 1-bit two-to-one multiplexer that can be used in the final task of this lab.
- Creating a slower clock from a faster clock.
- Having a physical LED blink noticeably on the Basys 3 development board, by using the slower clock.
- Using a switch to make a physical LED blink at two different speeds on the Basys 3 development board.

Recommendation: *Though it is not required to show to your assigned G.A. certain* ***UNDERSTANDING | TASK*** *in this manual, you are highly encouraged to fill in the answers required from these* ***UNDERSTANDING | TASK****. This is one of the ways students will be able to keep up with the learning objectives of this EE2020 module, as the contents get more challenging. All labs build upon the skills that should have been dutifully acquired through practice from previous labs.*

1.0 1-BIT TWO-TO-ONE MULTIPLEXER

A multiplexer (MUX) is a combinational circuit that connects one of its input signals to the output, based on the control signal. A simple 1-bit two-to-one mux, with inputs **A**, **B**, control signal **S**, and output **Z**, is illustrated as a functional block diagram, together with its simplified truth table, in ***Figure 3.1***.



S	Z
0	A
1	B

Figure 3.1: *Functional block diagram and truth table of a 1-bit two-to-one multiplexer*

UNDERSTANDING | TASK 1

A quick way to implement the Verilog code for a 1-bit two-to-one multiplexer is using the conditional syntax:

condition ? expression1 : expression2;

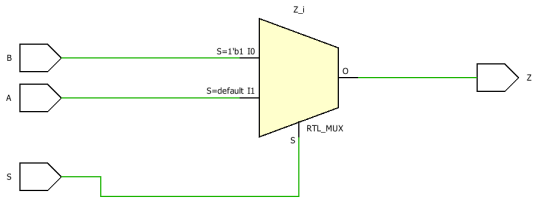
Notice in the schematic, how the code is automatically recognised as a MUX.

Verilog code for 1-bit two-to-one mux, using the dataflow method

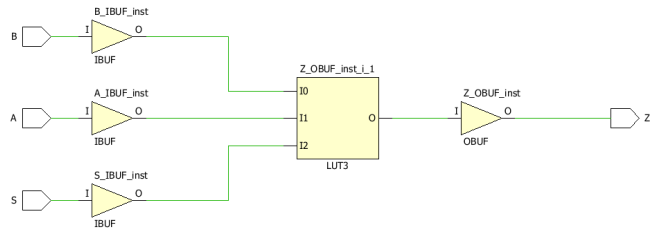
```
module my_2_to_1_mux (input A, B, S, output Z);

    assign Z = S ? B : A; // assign B to Z if S = 1 or assign A to Z if S = 0;

endmodule
```

Code Templates

RTL schematic for the 1-bit two-to-one mux



Synthesised design schematic for the 1-bit two-to-one mux



Fill in the truth table for the **LUT3**, as extracted from the synthesised design schematic for the 1-bit two-to-one mux:

Explain how the truth table for the **LUT3** matches that of the truth table indicated in ***Figure 3.1***:

_____

_____

_____

_____

_____

_____

2.0 THE BLINKING LED

A simple blinking LED is required to be implemented on the FPGA. To do this, a new signal, **CLOCK**, will be introduced.

The **CLOCK** signal is an external input signal that resembles a square wave of 50% duty cycle. If this **CLOCK** signal is connected directly to a physical LED, the latter will light up when the signal is HIGH, and will switch off when the signal is LOW, as illustrated in ***Figure 3.2***.

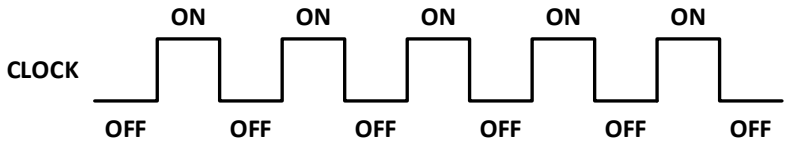


Figure 3.2: *A **CLOCK** signal with 50% duty cycle*

A simple dataflow description in Verilog for a blinky module is written first, followed by a simulation source to verify the design. To create the square wave, or **CLOCK** signal, in the simulation source, a new section of codes will now be introduced:

Verilog code for blinky, using the dataflow method

```
module blinky (input CLOCK, output LED);

    assign LED = CLOCK;

endmodule
```

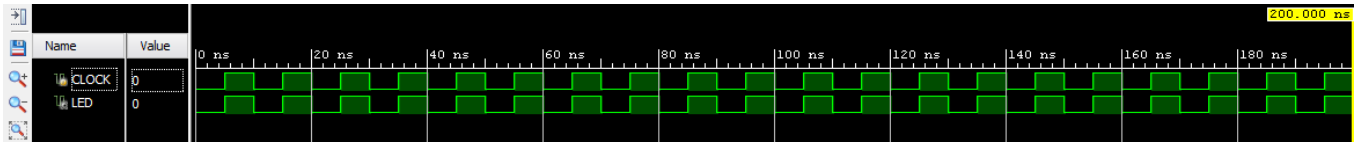
Simulation source code to test the blinky design

Code Templates

```
`timescale 1ns / 1ps

module test_blinky( );

    reg CLOCK; wire LED;

    blinky dut (CLOCK, LED);

    initial begin
        CLOCK = 0;
    end

    always begin
        #5 CLOCK = ~CLOCK;
    end

endmodule
```

The value of CLOCK is inverted every 5 units of time

Expected simulation waveform for the blinky design



UNDERSTANDING | TASK 2

Based on the Verilog code and simulation results, answer the following questions:

1. What is the unit of time being used in the simulation source?

   _____

2. Every 5 units of time, the value of **CLOCK** is being inverted. What is the clock frequency being used in this simulation?

   _____

3. What would happen if the testbench code **CLOCK = 0** is removed?

   _____

For the hardware implementation, instead of using an external signal generator for the **CLOCK** signal to the Artix-7 FPGA, the Basys 3 development board includes a single 100 MHz clock generator connected to pin W5 of the Artix-7 FPGA.

Import and **copy into the project** the **complete and unedited** Xilinx's design constraint file template (**Basys3_Master.xdc**), followed by these steps:

1. Uncomment lines 7 to 9 to create a clock signal of 100 MHz with 50% duty cycle. If required, rename the signal to the name used in your **blinky** code. In our example, the name **CLOCK** was used, and the final changes may look similar to ***Figure 3.3***.

2. Configure the output signal **LED** that is present in your **blinky** code (or the name chosen by you while writing the code) by linking it to any physical LED on the Basys3 development board.

```
1  ## This file is a general .xdc for the Basys3 rev B board
2  ## To use it in a project:
3  ## - uncomment the lines corresponding to used pins
4  ## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
5
6  ## Clock signal
7  set_property PACKAGE_PIN W5 [get_ports CLOCK]
8      set_property IOSTANDARD LVCMOS33 [get_ports CLOCK]
9      create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports CLOCK]
10
11 ## Switches
12 #set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
13     #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
14 #set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
15     #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
```

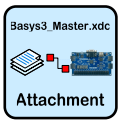


Figure 3.3: *Modifying the Basys3_Master.xdc*

UNDERSTANDING | TASK 3

Generate the bitstream and upload your code to the Basys3 development board.

What do you *notice* about the *blinking* LED?

_____

_____

3.0 THE NOTICEABLE BLINKING LED

To be able to observe a blinking LED at a frequency that is visible to the human eyes, modifications need to be done to the Verilog code. Let us introduce a temporary variable **COUNT** that is incremented by 1 at every rising edge (transition from low to high, and also called a positive edge) of the **CLOCK** signal, as shown in ***Figure 3.4***. By making use of **COUNT**, a lower frequency signal can be obtained, while the Verilog code for **COUNT** can be created by using the behavioural method of modelling.

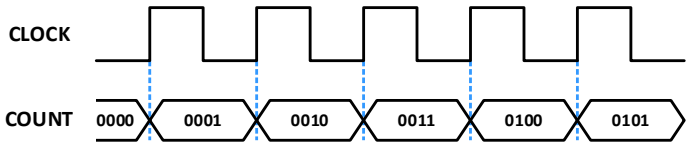


Figure 3.4: *Increasing **COUNT** at each rising edge of **CLOCK***

Verilog code for a slower blinky, using behavioural modelling

```
module slow_blinky_module (input CLOCK);

    reg [3:0] COUNT = 4'b0000;

    always @ (posedge CLOCK) begin
        COUNT <= COUNT + 1;
    end

endmodule
```

Code Templates

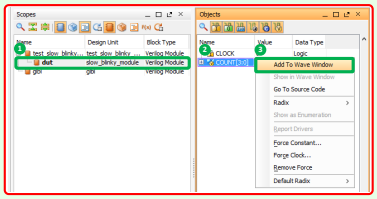
UNDERSTANDING | TASK 4

Create a simulation source for the **slow_blinky_module** design, and observe the waveform of signal **COUNT.**

Analysing a variable in the simulation waveform window

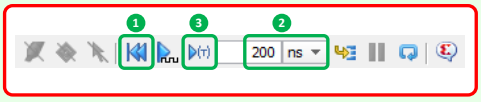
By default, the simulation window only shows the waveforms of input and output signals. To see the waveforms of variables during the simulation, such as the variable **COUNT**:

1. Select the **dut** under the simulation module being used
2. In the **Objects** window, right click on the **COUNT** variable
3. Choose **Add To Wave Window**

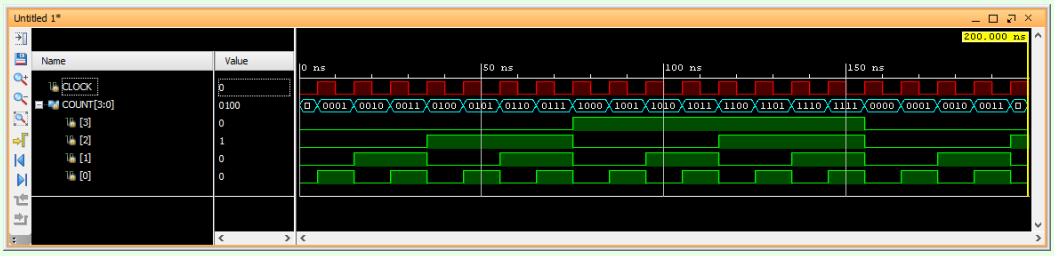


After adding the variable **COUNT** to the wave window, the current simulation needs to be re-run. Follow these steps:

1. Restart the simulation
2. Set the simulation time and units
3. Run the simulation for the amount of time set in step 2



The **COUNT** variable can then be expanded by clicking on the **+** symbol to the left of **COUNT**. This allows for every individual bit to be observed as independent waveforms:



State the frequency of **COUNT[3]**, **COUNT[2]**, **COUNT[1]** and **COUNT[0]**: _____ , _____ , _____ , _____

UNDERSTANDING | TASK 5

In the **slow_blinky_module**, insert the following line of code in the always block:

Code Templates

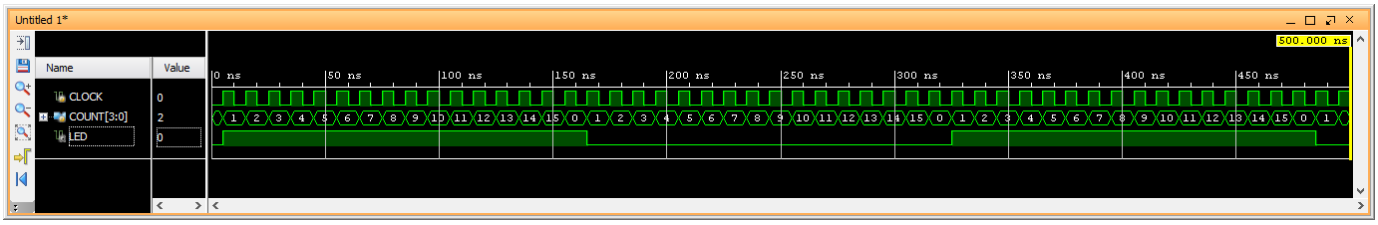
```
always @ (posedge CLOCK) begin
    COUNT <= COUNT + 1;
    LED <= ( COUNT == 4'b0000 ) ? ~LED : LED ;
end
```

Do additional modifications in the **slow_blinky_module** and the testbench code, and simulate the **slow_blinky_module** design.

Hints on the modifications related to slow_blinky_module and its related testbench

- `LED` is an output signal of the design module
- If an output signal is reused as input within the design module, it should be declared as **`reg`**
- A design module signal declared as **`reg`** can be given an initial value, if toggling is involved
- In the testbench, **`wire`** is used to declare output signals from the dut

If the codes have been correctly written, a waveform similar to what is shown below can be obtained:



Observe the waveform that you have obtained in your simulation window, and calculate the frequency of the signal **LED**:

f = _____ MHz

From your understanding of the multiplexer, state what the following line of code means:

```
LED <= ( COUNT == 4'b0000 ) ? ~LED : LED ;
```

_____

_____

UNDERSTANDING | TASK 6

Based on your knowledge obtained from UNDERSTANDING | TASK 4 and UNDERSTANDING | TASK 5, calculate the number of bits for **COUNT** which will allow **LED** to have a frequency of around 0.5 Hz to 1.5 Hz.

Number of bits required for **COUNT**: _____

Modify the **slow_blinky_module** design, and implement the LED blinking at a frequency of around 0.5 Hz to 1.5 Hz on the Basys 3 development board.

** **Warning:** Do not simulate the code. Why? **

4.0 CLOSING NOTES FOR LAB 3

FINAL UNDERSTANDING | TASK FOR LAB 3

At this stage, the physical LED on the Basys 3 development board should be blinking in a noticeable manner.

The final task is to modify the Verilog code such that there are two blinking speeds for the LED based on the state of a switch:

- If the switch is in the OFF position, the LED should blink at a frequency that lies between 0.5 Hz to 1.5 Hz
- If the switch is in the ON position, the LED should blink at a frequency that lies between 5.0 Hz to 15 Hz.

One way to accomplish this task is to use multiple multiplexers in Verilog, as learnt in section **1.0** of this lab manual. It may also be implemented using other methods.

Show this final task to your assigned G.A. That is, the LED on the Basys 3 development board blinks at two different speeds depending on the state of a specific switch.

Advice: *The **UNDERSTANDING | TASK** presented in this manual should be dutifully understood and practiced by each individual. The upcoming lab and project depend heavily on the knowledge and skills acquired here, and knowing them early on will make the subsequent lab and project more manageable.*

NATIONAL UNIVERSITY OF SINGAPORE

Department of Electrical and Computer Engineering

EE2020: DIGITAL FUNDAMENTALS - LAB 4

Sequential Circuits in Verilog - Part 2

AY 2016/2017, Semester 2

Teaching Assistants

Christopher M. Shin (Monday EE2020 Session)
Gao Jieyi (Wednesday EE2020 Session)

OVERVIEW

This lab is a continuation of Lab 3, and requires that you have a good mastery of the all the previous labs. In this Lab 4, more applications of sequential circuits will be explored, and this will be the last lab session before the EE2020 project.

The pre-requisites for this lab are:

- Knowing how to obtain multiple slower clock frequencies from a faster one.
- Brief understanding of the purpose of a D-Flip-Flop (D-FF) in a sequential circuit.

This lab will cover the following:

- Schematic and implementation of a single pulse circuit, by using two D-FFs with a slow clock signal.
- Incrementing a counter value by 1.
- Multiplying a counter value by 2.

Tasks for this lab include:

- Using a slow clock signal to create a single pulse signal, by using a mixture of modelling methods.
- Observing a pattern in the physical LED array, through a counter that uses the single pulse circuit.
- Observing the effects of using a binary shift left operator on the physical LED array.
- Using your mastery of all lab sessions to create a “Don’t Fall Off the Cliff!” game.

Recommendation: *Though it is not required to show to your assigned G.A. certain* ***UNDERSTANDING | TASK*** *in this lab manual, you are highly encouraged to fill in the answers required from these* ***UNDERSTANDING | TASK****. This is one of the ways you will be able to keep up with the learning objectives of this EE2020 module, as the contents get more challenging. All labs build upon the skills that should have been dutifully acquired through practice from previous labs. You are required to independently complete the final task and* ***learn from the processes that go into achieving it****, instead of rushing to complete the task within the same lab session.*

1.0 DEBOUNCING: CREATING A SINGLE PULSE CIRCUIT

UNDERSTANDING | TASK 1

The single pulse circuit is best analysed and understood with a slow clock. Using your knowledge from Lab 3, create a design module that generates a square wave with 50% duty cycle and frequency of around 4 Hz. Call the output signal from this design module as **SLOWCLOCK**.

In this section, a debouncing / single pulse circuit will be created. Mechanical switches cause bounce in the signal when toggled. There are many ways to implement a debouncing but for signal bounce of less than one clock period, the circuit from Figure 1 can be used to generate a debounced output signal. The output signal will be a synchronised logic true signal that lasts for the duration of one clock cycle. In order to create the single pulse circuit, two D-FFs and an AND gate will be used, as illustrated in ***Figure 4.1***.

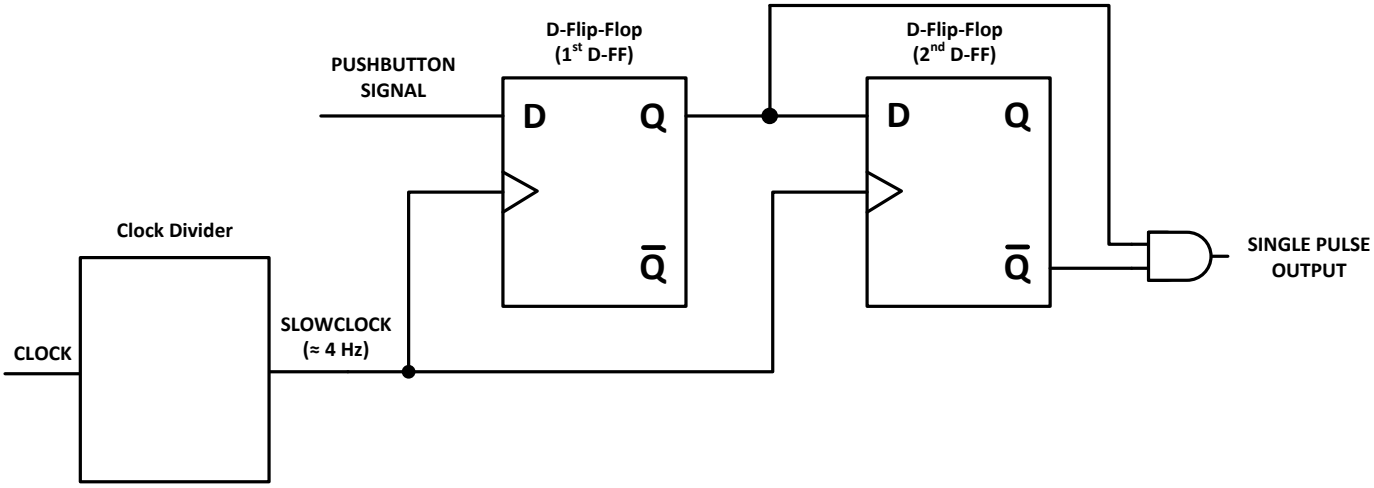


Figure 4.1: *Single pulse circuit*

The waveform for the single pulse circuit is as shown in ***Figure 4.2***.

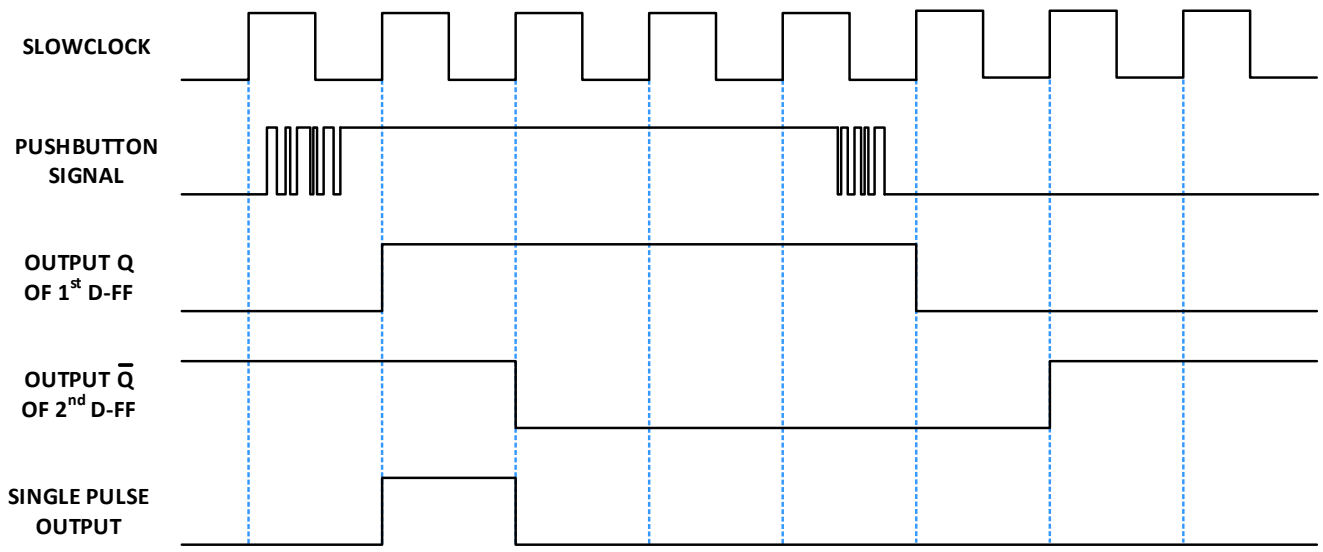


Figure 4.2: *Waveform from a single pulse circuit*

UNDERSTANDING | TASK 2

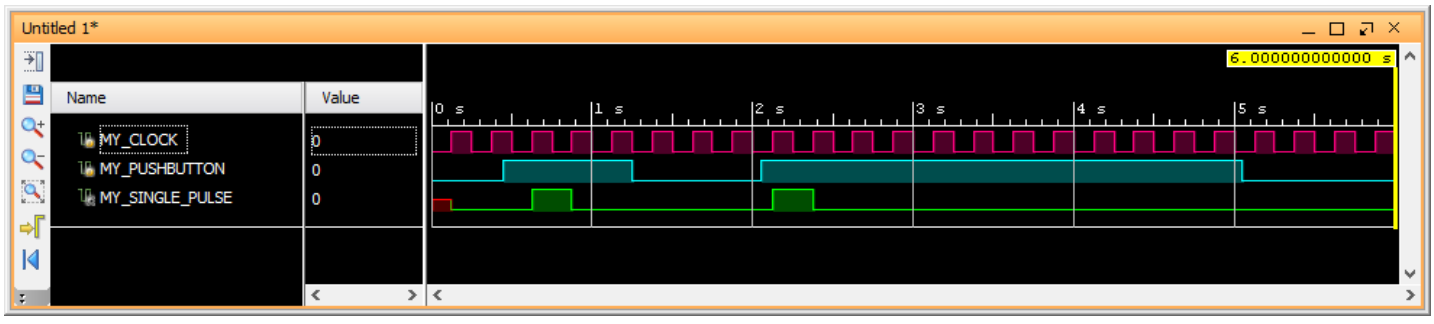
Through behavioural modelling in verilog, the code for a positive-edge triggered D-flip-flop can be created:

Verilog code for a positive-edge triggered D-flip-flop

```
module my_dff(input DFF_CLOCK, D, output reg Q);

    always @ (posedge DFF_CLOCK) begin
        Q <= D;
    end

endmodule
```

Code Templates

Using structural, dataflow and/or behavioural modelling, **create the circuit shown earlier in *Figure 4.1***. Simulate your single pulse design module with an appropriate testbench.



2.0 APPLYING THE SINGLE PULSE

UNDERSTANDING | TASK 3

From UNDERSTANDING | TASK 2, it is noted that no matter how long the pushbutton is held, a single synchronous pulse of only one clock period is created. Using this knowledge, an 8-bit counter whose 8 bits are represented by **LEDARRAY**, and which increments by only 1-bit no matter how long the pushbutton is pressed, can be created. Create a design module that achieves this task, and which includes the Verilog code shown below:

Partial Verilog code for a counter that increments by one at the positive edge of the synchronised single pulse signal

```
reg [7:0] LEDARRAY = 8'b0000_0000;

always @ (posedge SINGLE_PULSE) begin
    LEDARRAY <= LEDARRAY + 1;
end
```

Code Templates

Simulate your design to verify that your code is functioning as intended. Subsequently, implement the code on the Basys 3 development board. Press and release the pushbutton 5 times, each time recording the 8-bit **LEDARRAY** value below:

Assert Pushbutton	1st	2nd	3rd	4th	5th
Value of LEDARRAY					

What kind of pattern do you notice in the values?

_____

_____

We will now learn how to multiply a value by 2. Instead of using the multiplication operator, a much more efficient way is to use the binary shift left << operator. Given an original value for **LEDARRAY**, the number of bits shifted to the left indicates a multiplication of the original value by 2^n, where *n* is the number of bits shifted to the left. Three different shift operations are illustrated in ***Figure 4.3***.

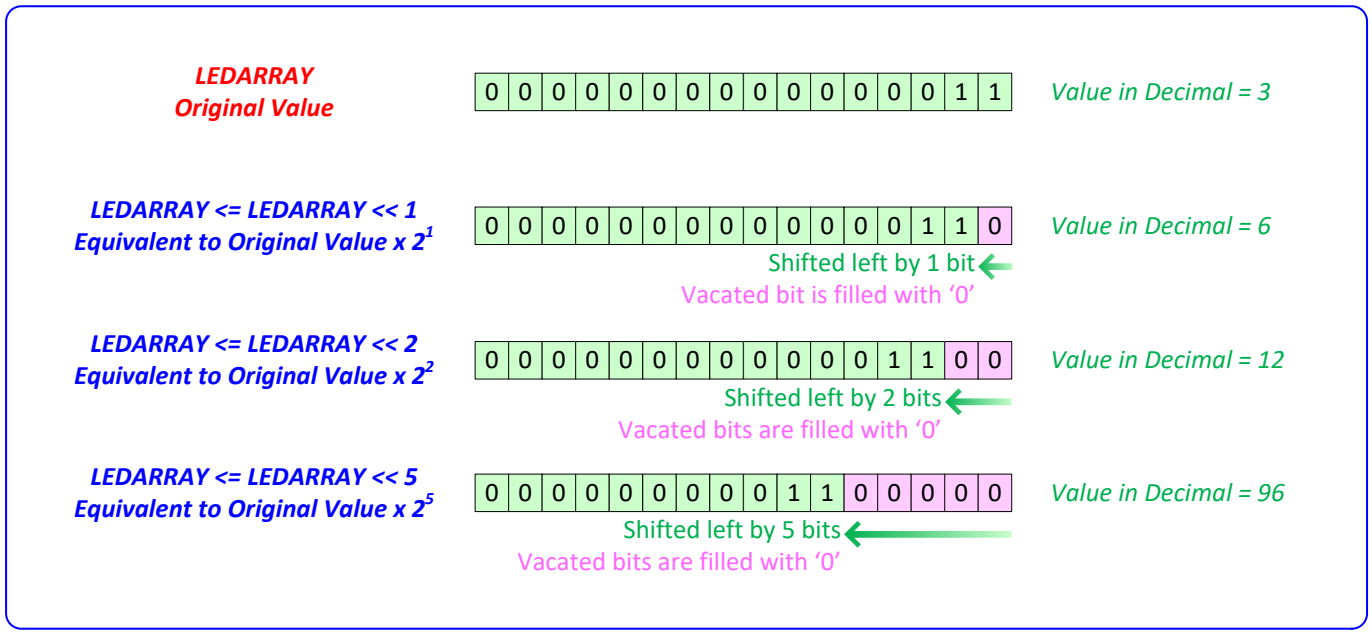


Figure 4.3: *Illustrating the binary shift left operator*

UNDERSTANDING | TASK 4

Create another design module that is very similar to **UNDERSTANDING | TASK 3**. Instead of incrementing the **LEDARRAY** by 1 at the positive edge of the synchronised single pulse, the **LEDARRAY** is value is multiplied by two. The Verilog code which achieves this task is shown below, assuming that the initial value of the **LEDARRAY** is 0x0C:

Partial Verilog code for a counter that multiplies by 2 at the positive edge of the synchronised single pulse signal

```
reg [7:0] LEDARRAY = 8'h0C;

always @ (posedge SINGLE_PULSE) begin
    LEDARRAY <= LEDARRAY << 1;              // Try LEDARRAY >> 1 to see what happens
end
```

Code Templates

Implement your design on the Basys 3 development board, assuming that the original value of **LEDARRAY** is set to decimal 12. Press and release the pushbutton 5 times, each time recording the 8-bit **LEDARRAY** value below.

Assert Pushbutton	Original	1st	2nd	3rd	4th	5th
Value of LEDARRAY	00001100					
Unsigned Decimal	12					

What kind of pattern do you notice in the values?

_____

_____

What happens for presses after the 4th one?

_____

_____

Describe your answer while showing this task on the Basys 3 development board to your assigned G.A.

3.0 CLOSING NOTES FOR LAB 4

FINAL UNDERSTANDING | TASK FOR LAB 4

Using all your knowledge and skills that have been dutifully acquired over all the labs, lectures, and tutorials, you can now test them on a design task as described below:

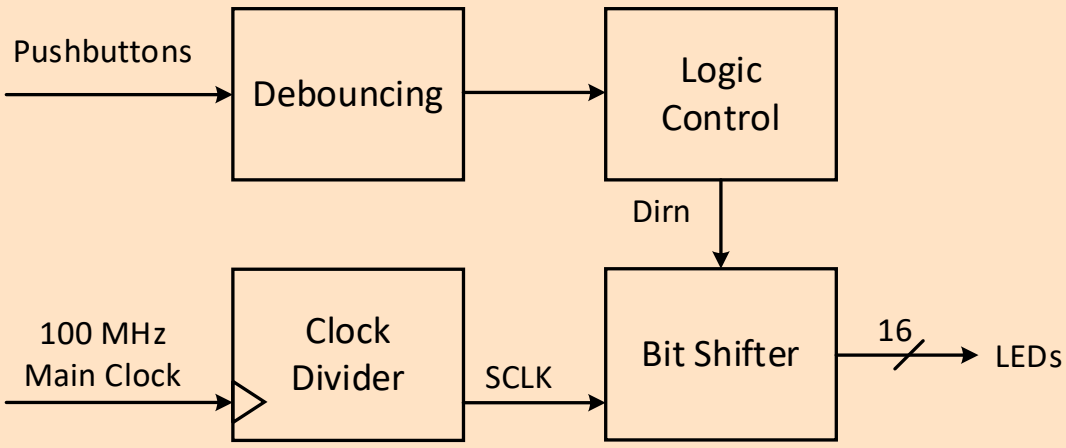
Design a "Don't Fall Off the Cliff!" game:

There are 3 people (represented by 3 LEDs) tied together on the top of a cliff! They are running left and right in a panic but you need to prevent them from falling off the cliff or it's game over!

- At the start of the game, these three people appear on the cliff.
- Pressing the middle pushbutton starts the game and these three people will run towards the left / right.
- Toggling the left and right pushbuttons will change their direction and make them move towards the left or right respectively.
- There are two ways to increase the difficulty of the game:
  (1) Reducing the width of the cliff:
  Changing the left cliff boundary: Use SW15 / SW14 / SW13
  Changing the right cliff boundary: Use SW0 / SW1 / SW2
  (2) Using the top and bottom pushbuttons, increase the speed at which they are running! Design for your game to operate at three speeds.
  Easy: Between 0.5 Hz – 2.0 Hz
  Moderate: Between 3.0 Hz – 6.0 Hz
  Difficult: Between 8.0 Hz – 10.0 Hz
- If any of the three persons (LEDs) falls over the cliff, game over and all LEDs light up and blink!
- Lastly, all pushbuttons should be debounced to ensure that the game is fair!

Suggestion for the "Don't Fall Off the Cliff!" game:

You may consider creating design modules of "Clock Divider", "Debouncing", "Logic Control" and "Bit Shifter" etc., followed by instantiating them in your main design module.



Note that this is just one possible example of how to make the "Don't Fall Off the Cliff!". You are free to choose your own design / modelling alternatives in meeting the objectives of the game.

Report submission for the "Don't Fall Off the Cliff!" game:

- Minimum of 2 pages with useful contents, of which at least one page describes your implementation through basic flowcharts, and at least one page shows and explain your main "Don't Fall Off the Cliff!" Verilog code in detail. You may have additional pages in case you need to include other information that you think is necessary to demonstrate your understanding.
- Report should be as a softcopy. Hardcopies are not accepted.
- Report should be in PDF format, with good quality screenshots (no camera pictures) if required.
- Naming of the report to follow this template as close as possible:
  Lab4_Official EE2020 lab session day_**Name as indicated on your matriculation card**_Matriculation number_Report
  Example:
  Lab4_Monday_Maurice Constance_A0131086Y_Report
- Upload to IVLE, in the folder corresponding to your official EE2020 lab session day.
- The IVLE upload must be completed within 7 days of your official EE2020 lab session day. If you have attended other lab session days on a temporary basis, your EE2020 official lab session day is the one considered for the upload deadline.
- To make your report complete, you are required to include feedback regarding the labs and the difficulties faced. Note that the feedbacks themselves, whether positive or negative, do not have any effects on your grades ☺.

Laboratory assessment for the "Don't Fall Off the Cliff!" game:

- Your Vivado workspace for this "Don't Fall Off the Cliff!" game needs to be submitted to IVLE within 7 days of your EE2020 lab session day. If you have attended other lab session days on a temporary basis, your EE2020 official lab session day is the one considered for the upload deadline.
- Refer to the document "*Archive Project in Vivado*".
  Instructions are given to properly make a copy of your complete Vivado workspace.

- Naming of the Vivado workspace archive to follow this template as close as possible:
  Lab4_Official EE2020 lab session day_**Name as indicated on your matriculation card**_Matriculation number_Workspace
  Example:
  Lab4_Monday_Maurice Constance_A0131086Y_Workspace
- Upload to IVLE, in the folder corresponding to your official EE2020 lab session day.
- You are required to ensure that your bitstream has been successfully generated before submitting your Vivado workspace. You will not be allowed to change your Verilog codes or generate your bitstream during the lab session in week 8.
- Your Vivado workspace archive and report will be downloaded by your assigned G.A.s before the lab session in week 8.
- A marks penalty will be applied if either the Vivado workspace archive or report have not been submitted to IVLE, using the correct naming scheme, by the deadline.
- The G.A. will upload the bitstream to a Basys 3 development board, and you will need to demonstrate your program.
- Technical questions regarding your codes may be asked by any of the EE2020 teaching team.

Academic Integrity:

Reminder from the NUS Code of Student Conduct:
http://www.nus.edu.sg/registrar/adminpolicy/acceptance.html

"Any student found to have **committed** or **aided and abetted** the offence of plagiarism may be subject to disciplinary action"

Both the **source** and **recipient** of assessable codes and/or report are equally responsible in cases of academic dishonesty.

Ending of regular lab tasks: With the focus being on the project from week 8 onwards, no more lab progress will be recorded after your EE2020 official lab session in week 8. You are required to focus on the project only as soon as week 8 lab starts.