

Faculty of Automation and Computer Science

Year I, Group 30413

Variable Lighting System

Coordinating Professor: Student:

Timar Mihai Cristea Tudor

Content

1. Purpose of the Project 3
2. Pulse-Width Modulation – Theoretical Aspects 4
3. Block Diagram of the Project 5
4. Details about the Project:

4.1. List of Components 5

4.2. Pinout of the Inputs 11

4.3. Instructions for Use and Maintenance 11

4.4. Possible Improvements 17

5. Justification for the Chosen Solution 17

6. Bibliography 18

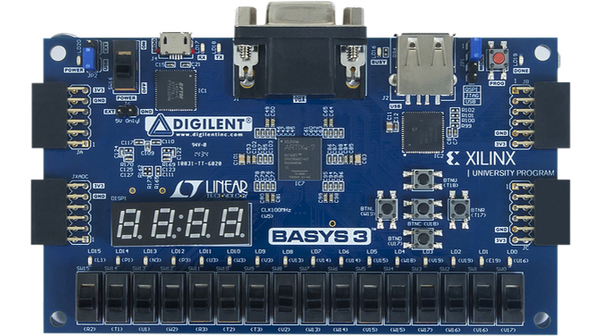
Purpose of the Project

Design a **Variable Lighting System** using the LEDs on the FPGA board. The system will have multiple functioning modes:

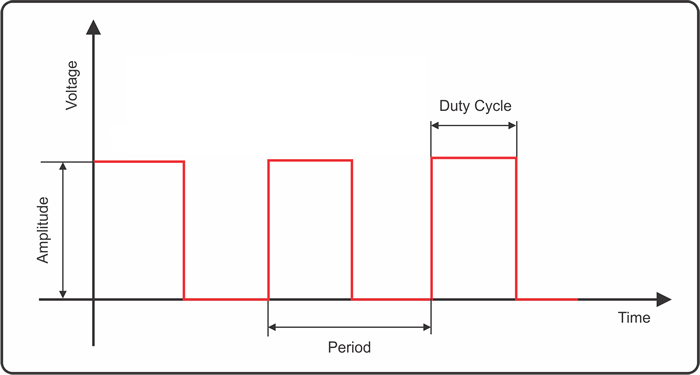
* Manual mode – the value of the brightness of the LEDs will be given using the switches (8 bits)
* Automatic mode – the brightness of the LEDs will go from their minimum value to their maximum value and back to minimum during an interval of the time measured in seconds. This interval of time is given as an input of the system (triangle wave)
* Test mode – the brightness of the LEDs will go from their minimum value to their maximum value during an interval of time specific to every LED (LED0 – 1 second, LED1 – 2 seconds, ..., LED15 – 16 seconds) (sawtooth wave)

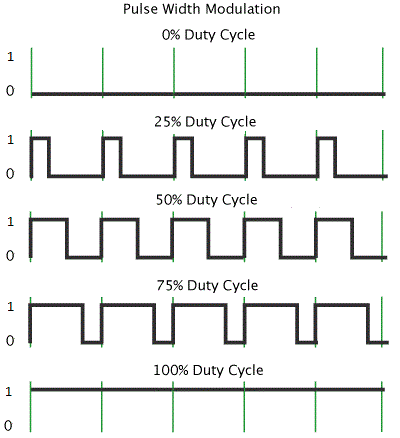
For the variation of the brightness, the PWM method will be used. The current working mode, the value of the brightness of the LEDs, as well as the period of the current wave will be displayed on the 7-Segment Display.

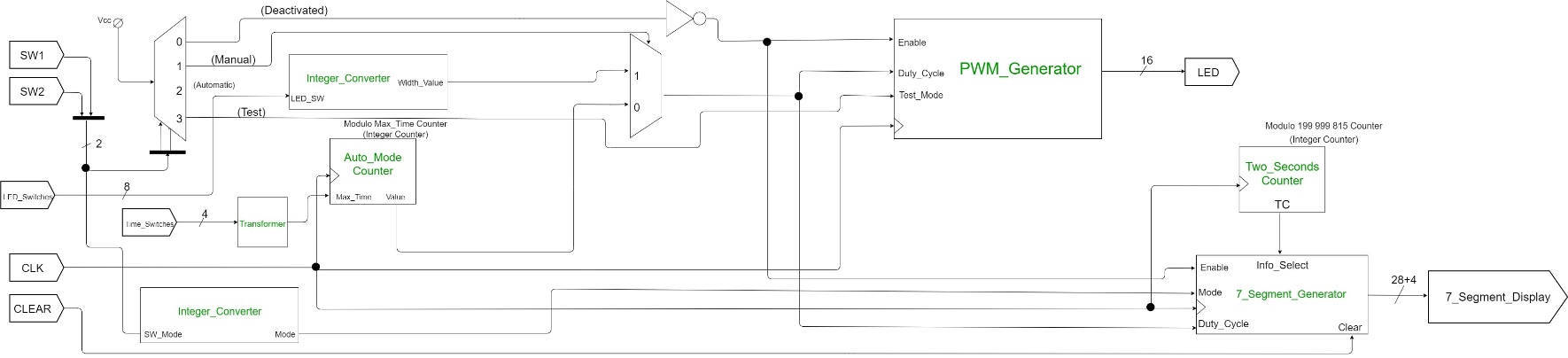
For this project, I will use a Basys3 FPGA board.



Pulse-Width Modulation – Theoretical Aspects

**Pulse-width modulation** (**PWM**), or **pulse-duration modulation** (**PDM**), is a method of reducing the average power delivered by an electrical signal, by effectively chopping it up into discrete parts. The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power supplied to the load.

The term *duty cycle* describes the proportion of “on” time to the regular interval or “period” of time. A low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on. When a digital signal is on half of the time and off the other half of the time, the digital signal has a duty cycle of 50% and resembles a “square” wave. When a digital signal spends more time in the on state than the off state, it has a duty cycle greater than 50%. When a digital signal spends more time in the off state than the on state, it has a duty cycle smaller than 50%. Here is a pictorial that illustrates these five scenarios:

Block Diagram of the Project

Details about the Project

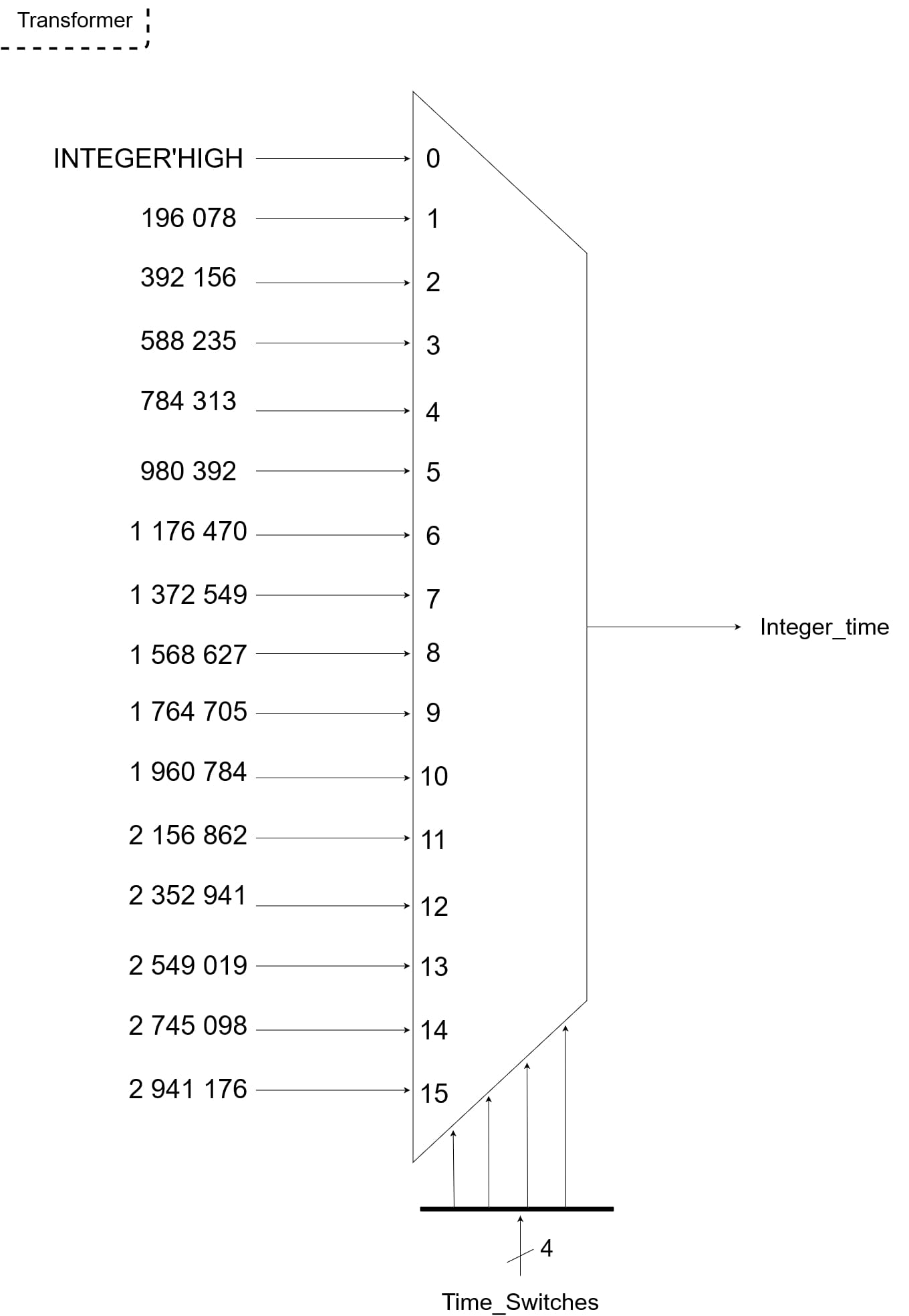
# List of components

1. DMUX 1:4

Firstly, the project is using a DMUX 1:4 in order to choose which mode of operation does the user want to see on the board, according to the diagram above (the 4th mode of operation simply does not do anything meaning that the seven segment display as well as the LEDs are turned off completely).

1. Integer Converter

This component consists of just a predefined function found in the IEEE.NUMERIC\_STD library. It converts both the bitstring resulted from the position of the switches used to give the value of the duty cycle on manual mode and the bitstring resulted from the position of the switches used to select the mode of operation of the board.

1. Transformer

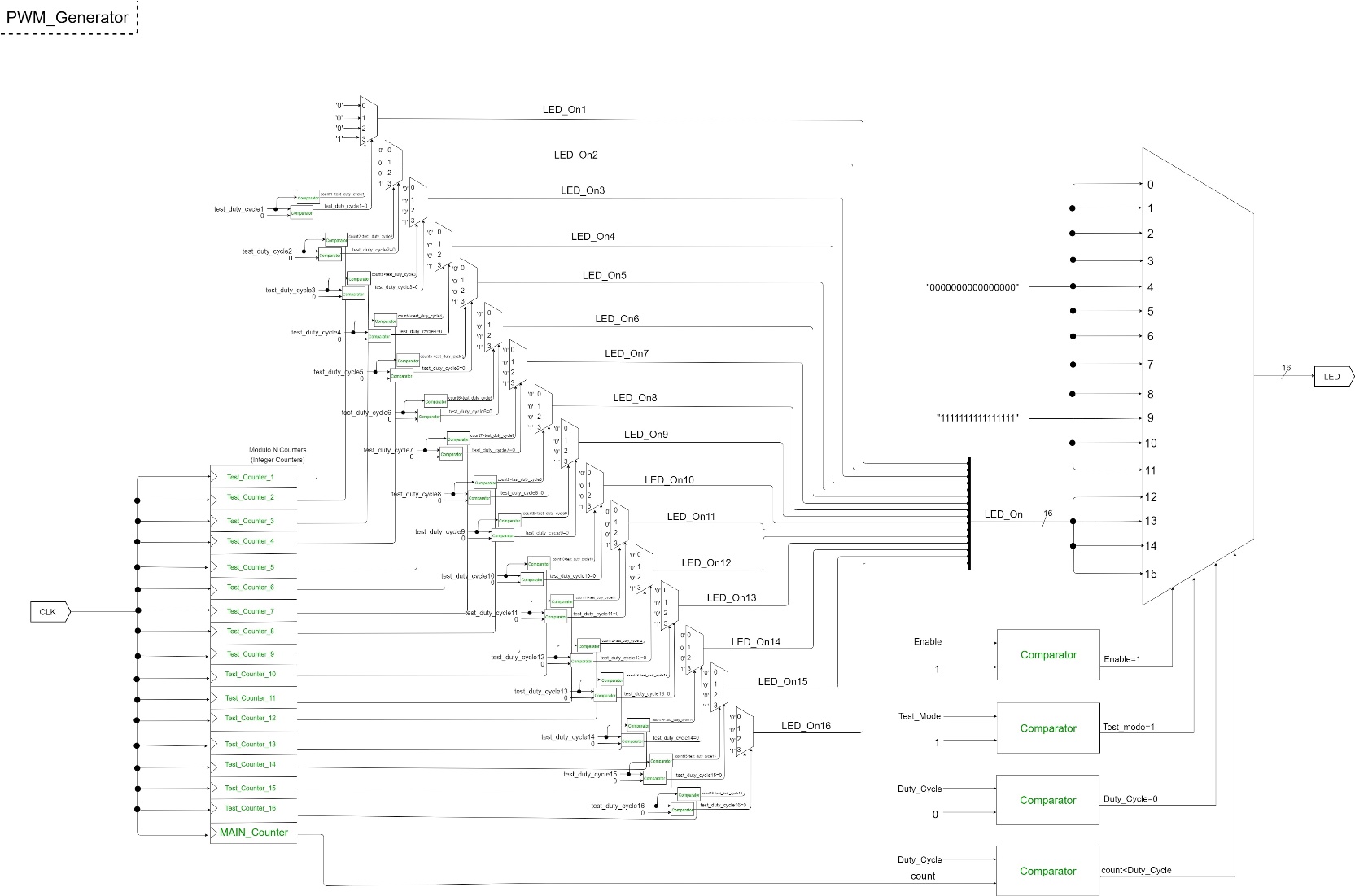
The transformer receives as input the bitstring resulted from the position of the switches used to give the value of the time period used for automatic mode. This bitstring is converted to an integer corresponding to that time period. Essentially, a 16:1 MUX is used and the values are taken automatically similarly to a look-up table. The computation of these values was done in the following way: knowing that the Basys3 board uses a Clock that has a frequency of 100MHz, which is also 108 Hz, the period of a clock cycle would be 1/108 seconds, which is 10 nanoseconds; now, for example, if we want to obtain the value corresponding to 2 seconds, we transform it into nanoseconds, divide this number to the aforementioned 10 ns, then divide it by 255, which represents the limit of counting (given by the 8 switches for the manual mode; 28-1= 255) and then divide it by 2 because the counter of the duty cycle is reversible, and so we have to split the time period in two; so for 2 seconds, we have 2x109 nanoseconds, divided by 10 that is 2x108, divided by 255 is 784 313 and finally, divided by 2 is 392 156. Of course, this number is just a close approximation of the exact number which represents the wanted time period.

1. Auto-Mode Counter

This counter is a modulo n counter (here, n = *max\_time*, computed previously by the transformer component). It is used, as the name says, for the automatic mode of operation of the board. However, this counter also gives the duty cycle for the LEDs, which also works like a counter, only this time, it is a reversible counter which increments (or decrements) the value of the duty cycle whenever the “main” counter is greater or equal to *max\_time* (or is smaller or equal to 0) (this is done in order to obtain the triangle wave). I used this safety precaution because I noticed that if I just wrote equal (without the greater or the smaller part), whenever I change the mode of operation to automatic mode or even change the time period during automatic mode, the LEDs would get stuck for a variable time period and the user would have to wait for them to get “unstuck” and start working normally again.

1. DMUX 2:1

This component chooses between the duty cycle given by the manual mode and the one given by the automatic mode, its selection being the bit corresponding to the manual mode, which is one of the outputs of the initial MUX.

1. PWM Generator

This is one of the most complex components of the system, because it contains several other sub-components of its own (these two sub-components are for the Test mode operation):

6.1) Test Counter

This counter is almost identical to the Auto-Mode counter mentioned above in the sense that it is also modulo n counter (here, n is an integer value corresponding to the time period in seconds). However, it is different than the Auto-Mode counter by the fact that when it comes to generating the duty cycle it is not reversible but it goes only up when it reaches the maximum value it restarts from zero (this is done in order to obtain the sawtooth wave).

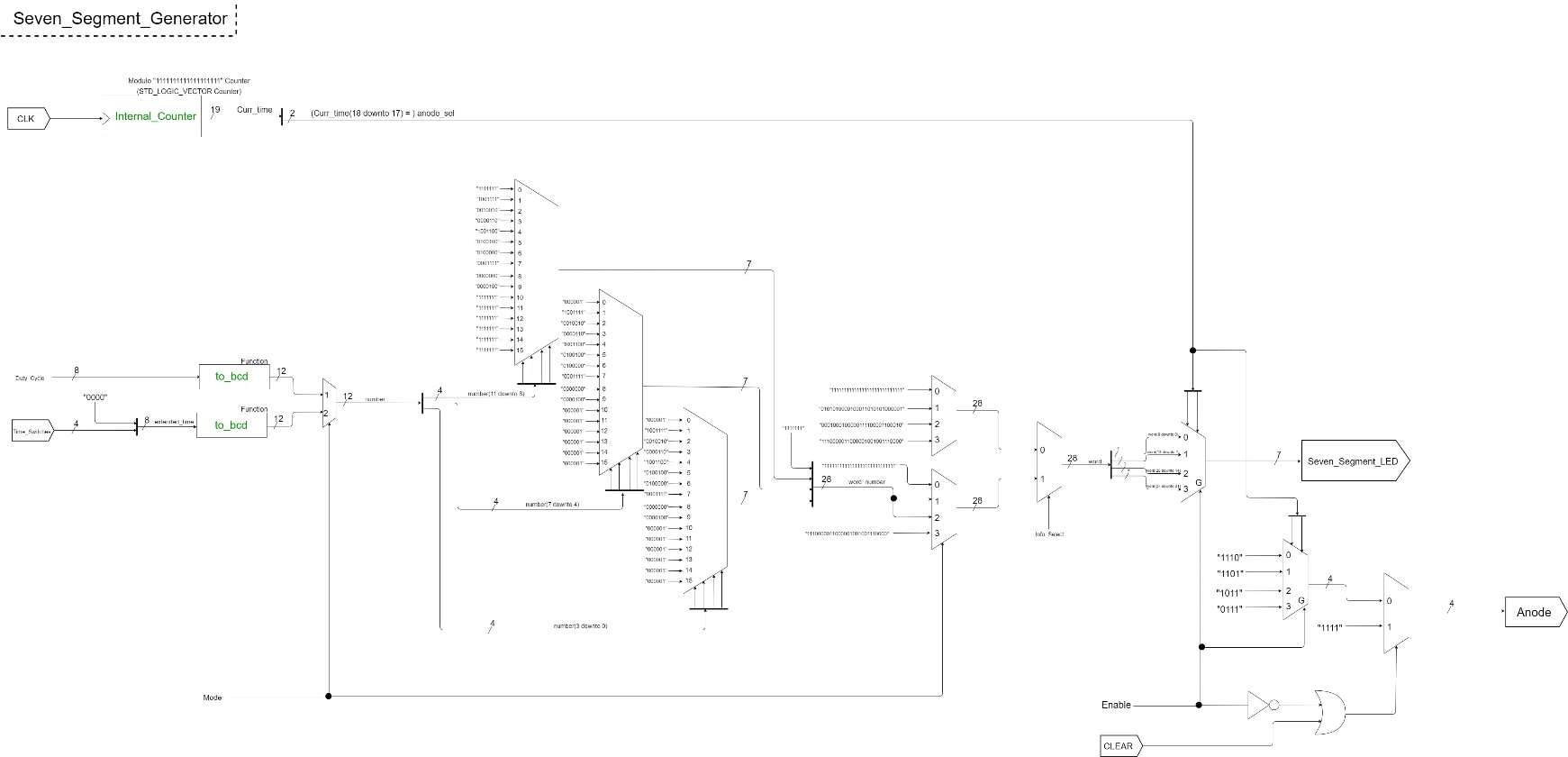
6.2) LED MUX

This MUX basically chooses when to turn on or off the LED based on the relation between the duty cycle obtained by the aforementioned test counter and the current value of the main counter (the counter of the PWM Generator itself). The on and off turning of the LED is done so fast that it gives the illusion of changing the intensity of the LED.

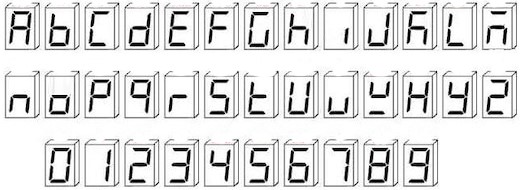
Coming back to the PWM Generator, it also contains a counter as well as a series of multiplexers which will help us choose which duty cycle to use when we are turning the LEDs on and off. We will first use the inverse of the output corresponding to branch number 0 of the initial MUX as an enable signal for the entire component because we first need to determine if we should use it or not (in case it needs to be deactivated). Then, we will use the output of branch number 3 of that same initial MUX as a signal to determine if the system should choose the duty cycles given by the LED multiplexers for test mode operation or the duty cycle resulted from the DMUX if the manual mode or automatic mode is activated. Lastly, similarly to the LED multiplexers’ way of functioning, the relation between the duty cycle and the current value of the counter is tested and so based on its result, the system will choose the rate at which it will turn on the LEDs. It is worth mentioning that manual mode and automatic mode give the same duty cycle to all of the 16 LEDs on the board, whereas test mode gives a different duty cycle to each of them.

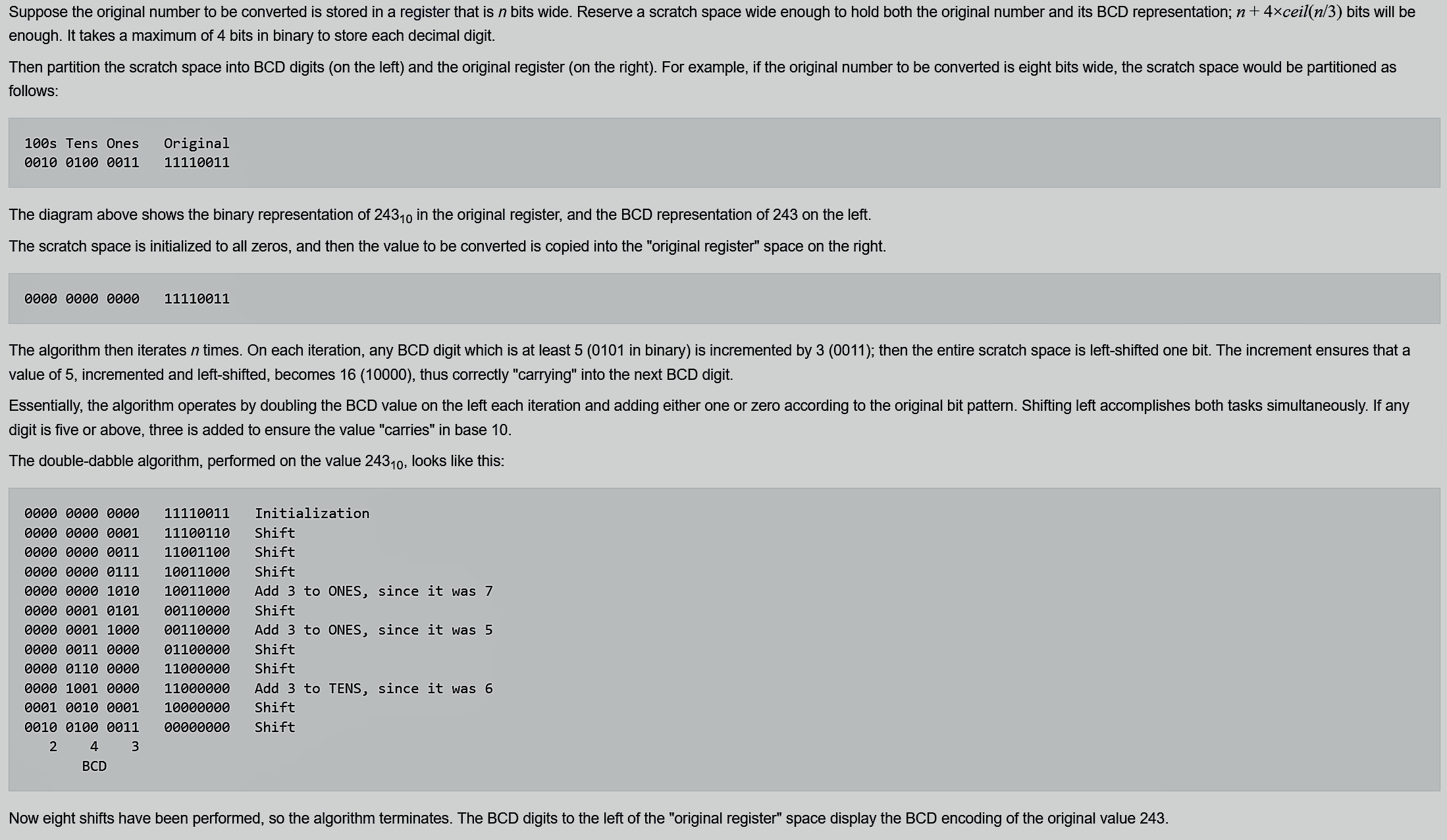
1. Two-Second Counter

For this counter, the situation is a bit different given the fact that we work with the seven segment display. Therefore, we can take the already computed integer value from the auto mode operation, i.e., 784 313, multiply it back by 255 and obtain 199 999 815. Additionally, it is a reversible counter, so it counts two seconds up and two seconds down. It is implemented in this manner because for manual mode and automatic mode, the seven segment display needs to display two different pieces of information, namely the name of the operation mode and the duty cycle (for manual mode) or the time period (for auto mode), and so I made the decision that the system will display for a duration of two seconds each piece of information (for test mode, only the name of the operation will be displayed).

1. Seven Segment Generator

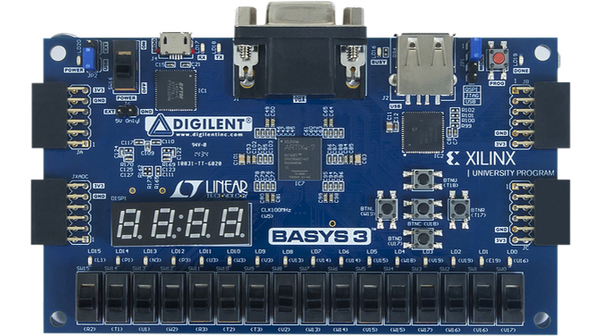
This is also one of the most complex components of the system. It also has an Enable signal (which is resulted from the negation of the same branch number 0 of the initial MUX) that dictates whether the component should be activated or not. Firstly, it contains a clock divider, which is a binary counter as opposed to all of the other counters so far, which were integer counters. This counter is a modulo “1111111111111111111” counter. The length of the bitstring is 19. This value was computed in the following way: instead of the 100MHz clock cycle, we would want a 250Hz clock cycle, so we take the 100MHz, we convert them into Hz by multiplying with 106, we divide the result by 250 and obtain 4x105; then, to this result we apply a logarithm in base 2 and obtain the number 18, which represents the length of the desired bitstring for a frequency divider which would have 250Hz clock cycle (~4ms). However, 19 can also be chosen since such a small difference in length does not affect the way we want it to function. I opted for a binary counter because the system will use the two most significant bits and use them as the selection bits for the anode selection as well as the selection of each character for each position of the seven segment display. In order to obtain the digits of the numbers that we need to display, they go through many transformations: firstly, for auto mode, we need to extend the time which is on 4 bits to 8 bits by concatenating the bitstring “0000” at the beginning of the bitstring, but for manual mode we don’t need to do that since the duty cycle is already expressed on 8 bits; now, that we have the 8-bit number, we will apply the “Double Dabble” algorithm (also known as the “Shift Add Three” algorithm), which is described below; the resulted bitstring has 12 bits because we now have each digit of the decimal number in BCD representation; afterwards, three groups of 4 bits are created, one for each digit (we don’t need a longer bitstring since we don’t need to display a decimal number which has more than three digits) and a 28-bit number is created so that it can be displayed properly on the seven segment display (7 bits for each of the four positions). The alphabet used for expressing letters and digits is the following:



“Double Dabble” algorithm:

# Pinout of the Inputs

* *SW1* and *SW2* are used to choose the working mode: Deactivated (“00”), Manual (“01”), Automatic (“10”), Test (“11”)
* *LED****\_****Switches* (8)are used to select the value of the duty cycle for manual mode
* *Time\_Switches* (4) are used to select the time period for automatic mode
* *CLK* represents the clock of the system and it’s represented by the CLOCK found on the board
* *CLEAR* (button) is used to clear whatever is displayed on the seven segment display (it clears everything as long as it is pressed; when the button is released the information reappears)
* *LED* (16) are the on-board LEDs and represent the core of this project
* *Seven\_Segment\_Display* (28+4) is represented by the 28 segments of the display (7 for each of the 4 available characters; the dots are not used) as well as the 4 anodes, one for each character of the display



***CLK***

***SW1* and *SW2***

***CLEAR***

***Time\_Switches***

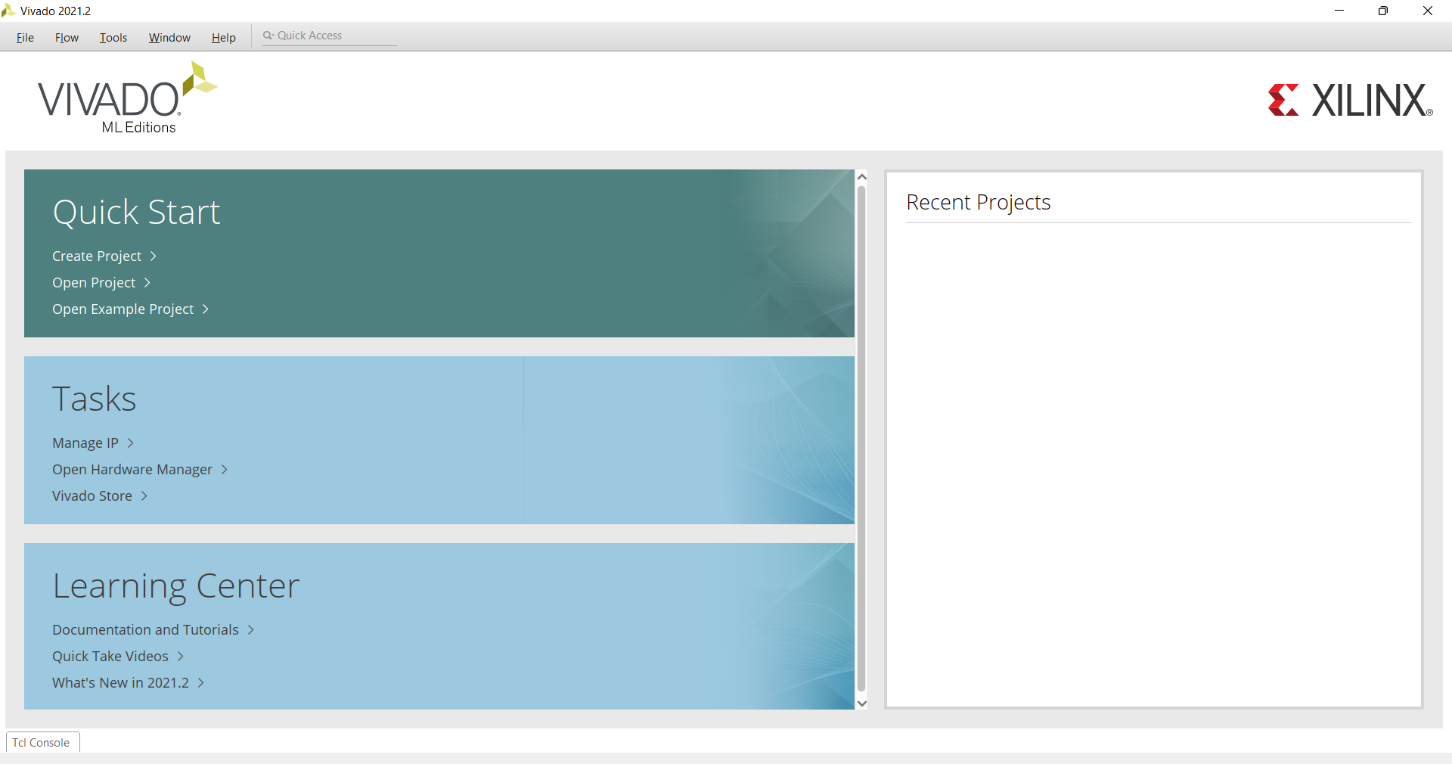
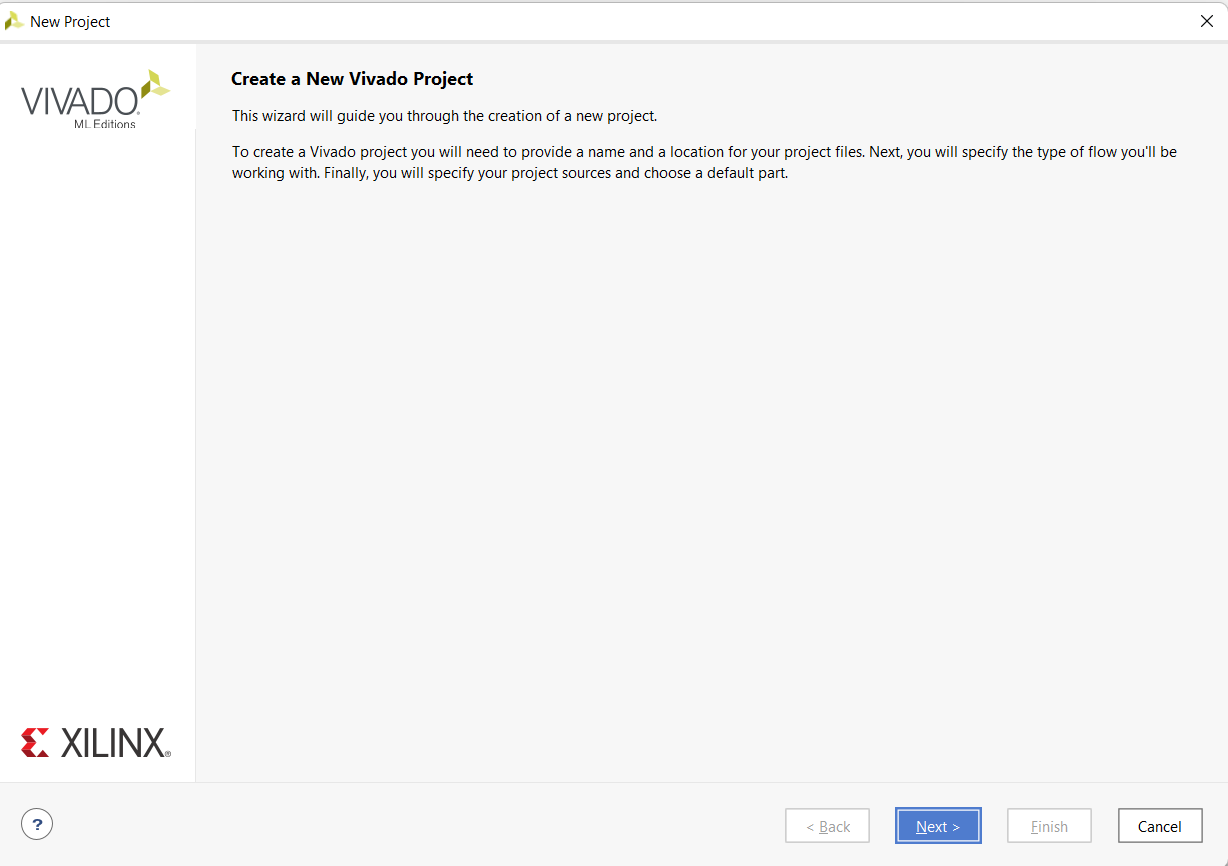
***LED\_Switches***

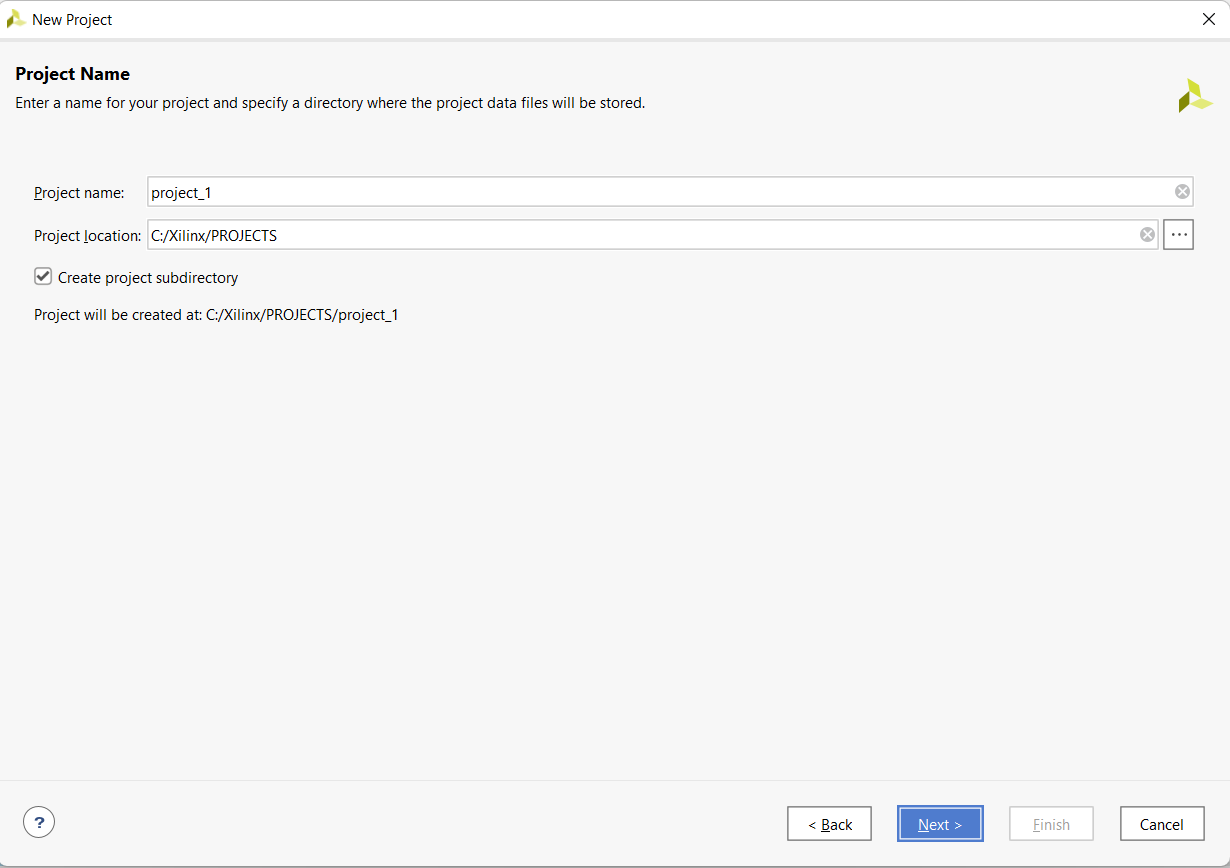
***LED***

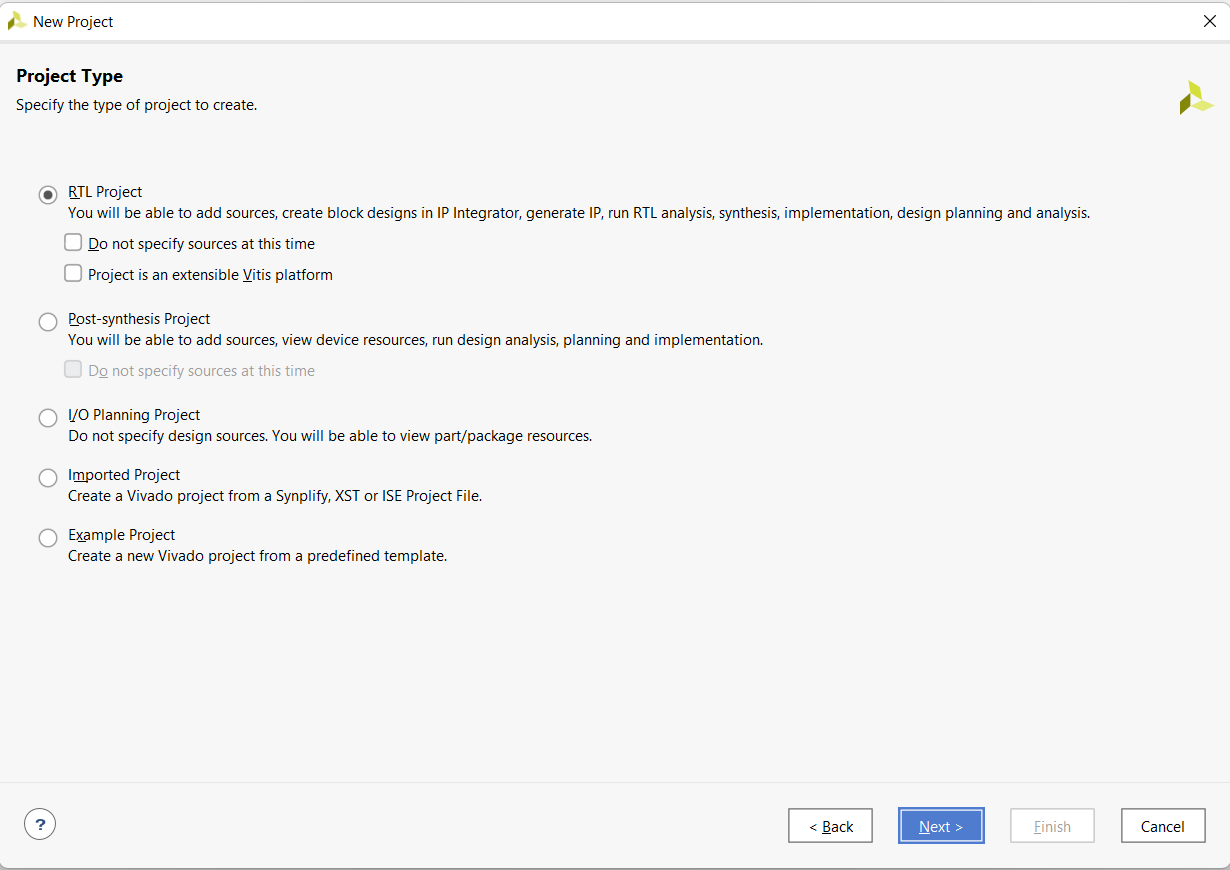
***Seven\_Segment\_Display***

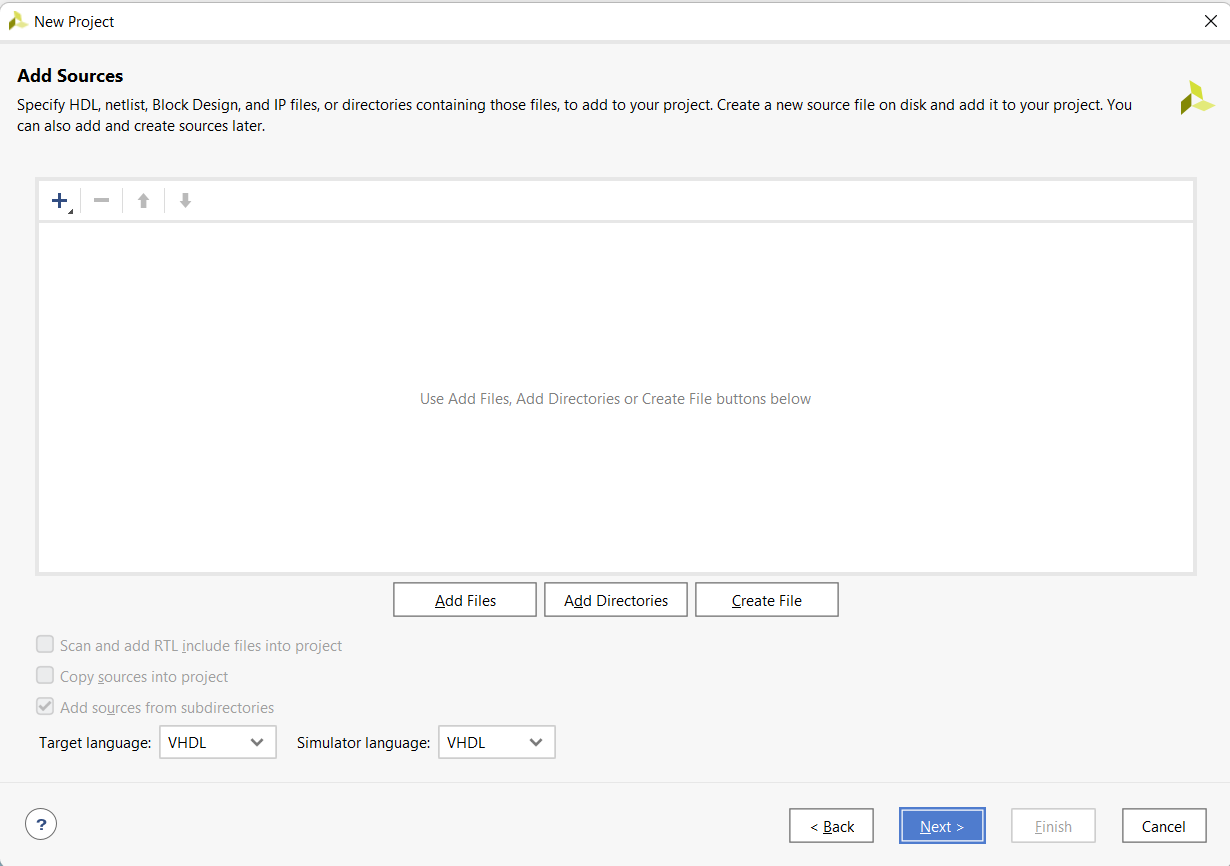
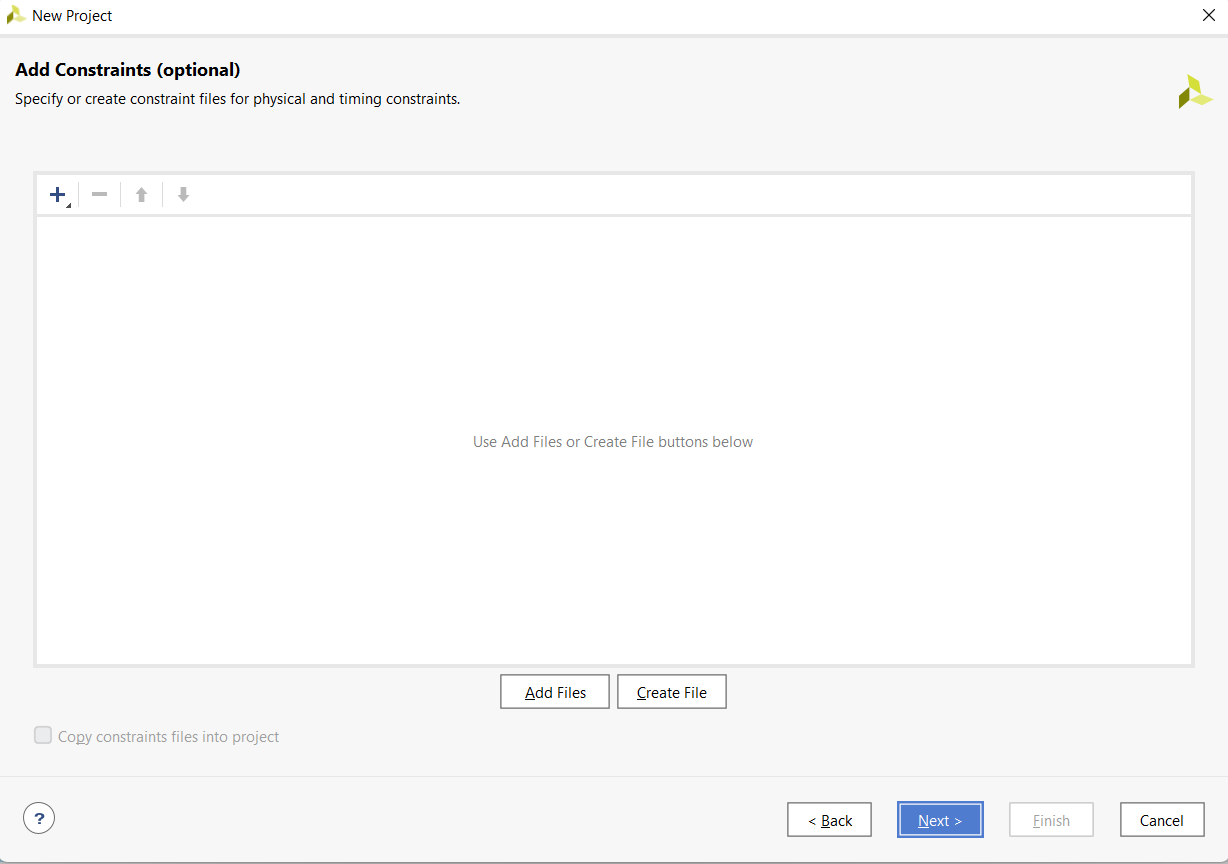
# Instructions for Use and Maintenance

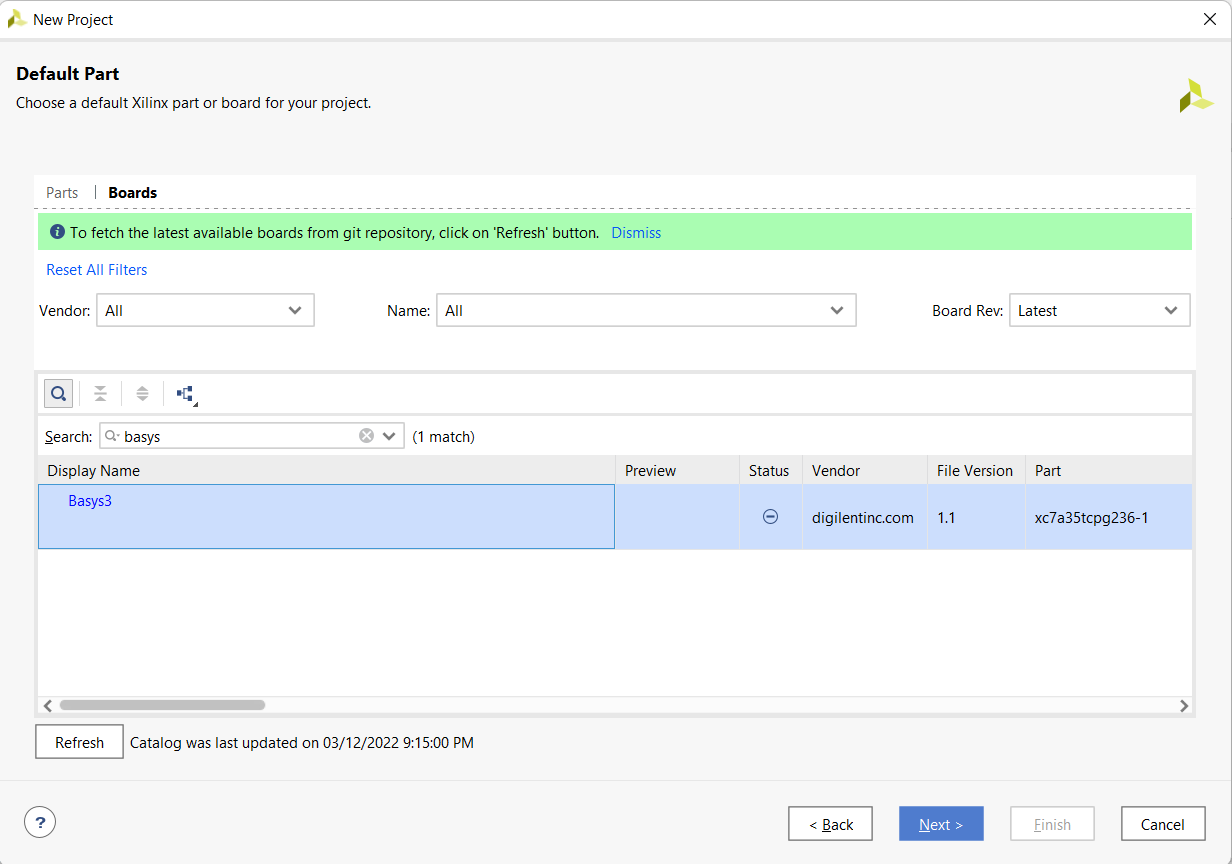
For the project I personally used Vivado ML Editions 2021.2, but the process of creating the project in Xilinx ISE is somewhat similar. To create the project, the following steps must be taken:

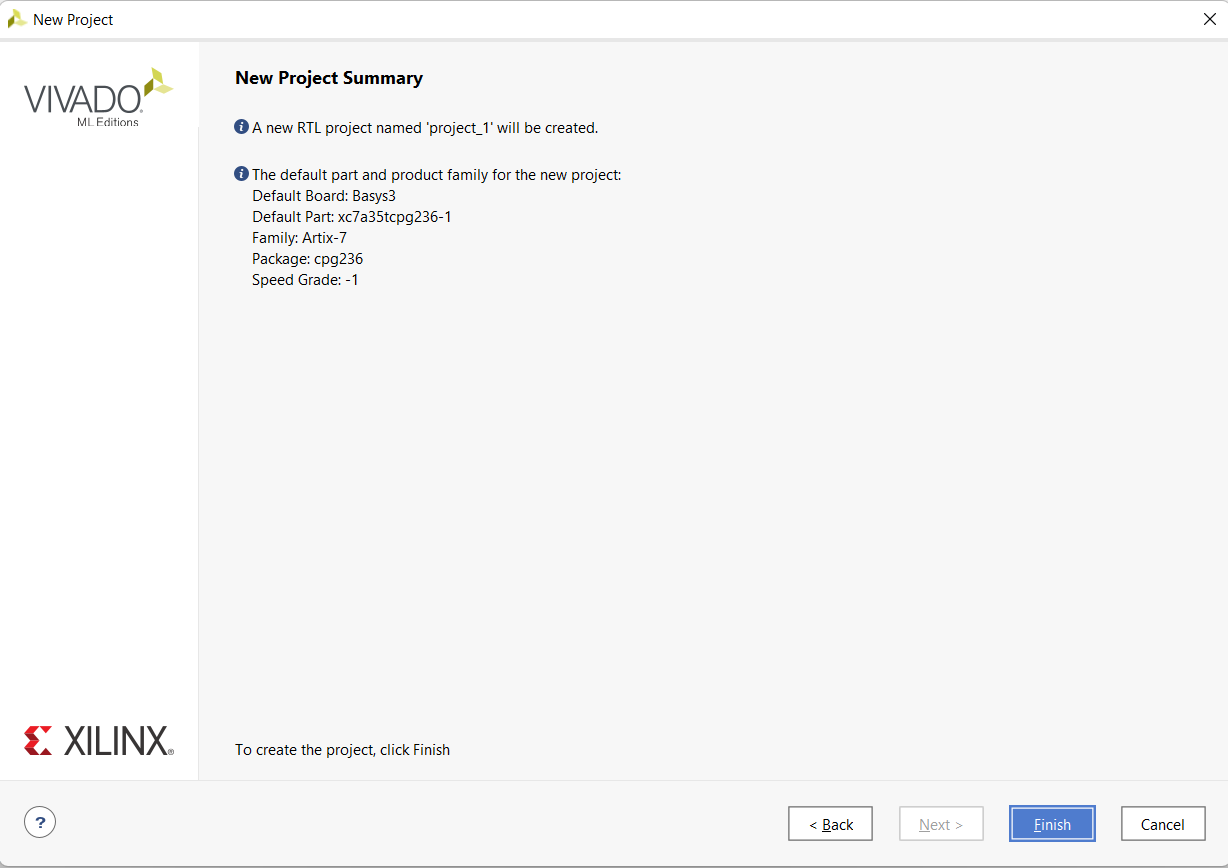
1. We open the Vivado application on a computer and click the button “Create Project”
2. Then we click the “Next” button when a new window opens up. On the next page, we need to choose a name and a location for the project, and click “Next” again



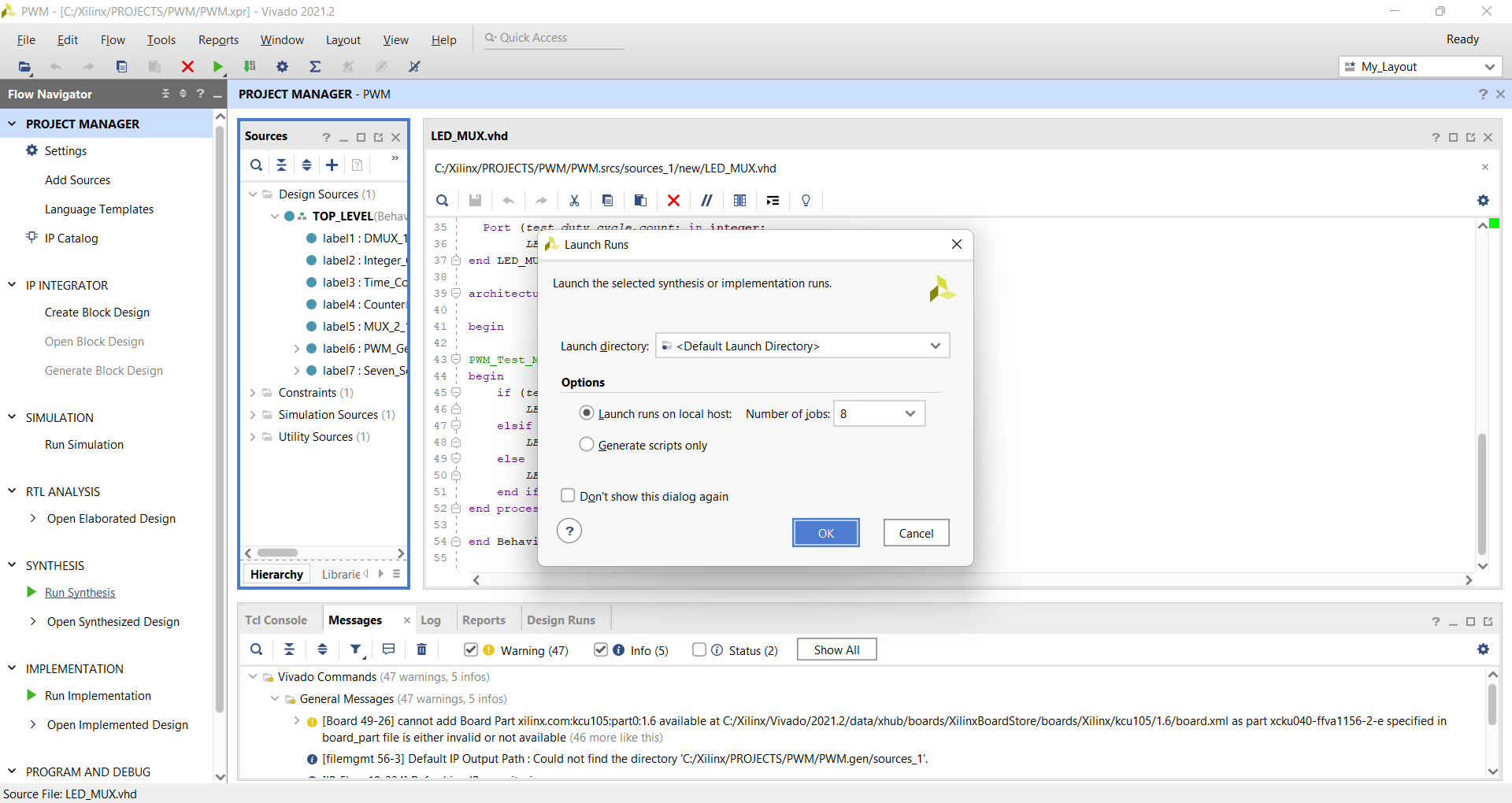
1. On this page, we need to choose the “RTL Project” option and untick both checkboxes. Click “Next” again and you will be prompted with the option to add all of the source files to the project (only the .vhd files). The target language should be VHDL. Click “Next” again and on this page we can now also add the constraints file (.xdc file). Click “Next” again

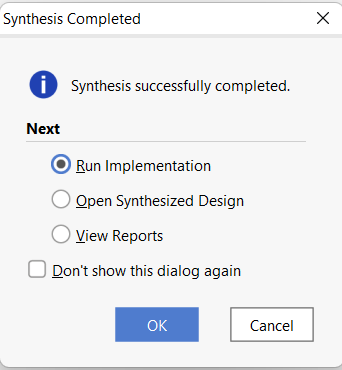
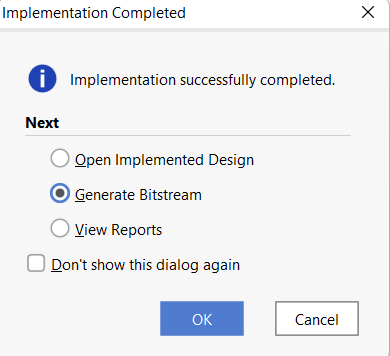
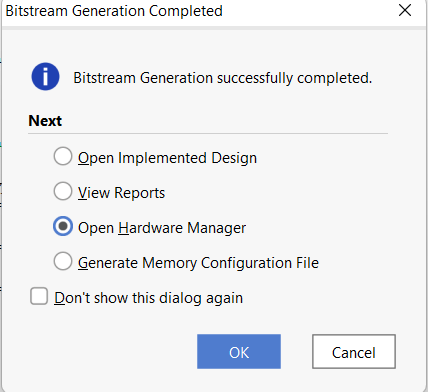


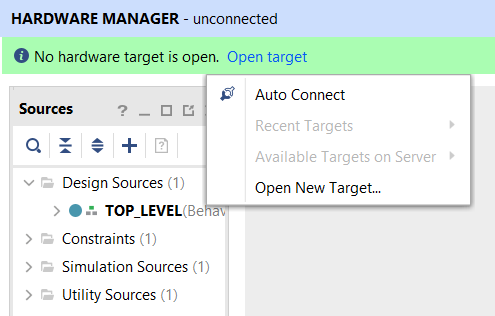
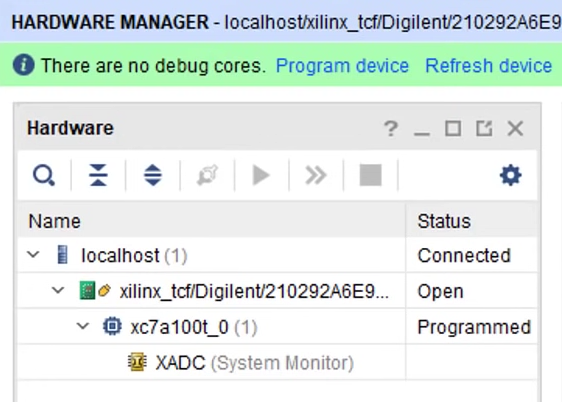
1. On this page, we need to choose the board that the project will be programmed on. I used a Basys 3 board (make sure you are on the “Boards” page) so that is what we will search for using the search bar (if the board does not appear, then clicking the “Refresh” button should update the catalogue and solve the problem). After clicking “Next” one last time, a summary of the project will be presented to you. Click “Finish” and, within a few seconds, the project will be created and ready for further actions.



1. Now that the project is ready to go, we can hit the “Run Synthesis” button found on the left side of the application and click the “OK” button on the pop-up. After a while, another pop-up will appear informing us of the fact that the synthesis was successfully completed. Here, we can choose what happens next. For our project we need to choose “Run Implementation” and click “OK” again. A similar pop-up will appear after the implementation is complete. Next will be “Generate Bitstream” (this is the file that will be uploaded on the board) and click “OK”. Afterwards, choose “Open Hardware Manager” and click “OK” one last time. This will help us establish a connection with the board so that we can program it.





1. The last step is to open the target, i.e. the Basys 3 board and program it. To do that, make sure that the board is connected to the computer and is powered on. Then, click the ”Open target” button as well as the “Auto Connect” button. After you see that the localhost is connected and that the board is open, click the “Program device” button and within a few seconds, the program will be uploaded and ready to be tested on the board

# Possible Improvements

The project can be further developed by adding new functionalities, such as:

* The addition of two buttons: “Increase” and “Decrease”, which can manually raise or lower the Duty Cycle by one unit for the manual mode / raise or lower the period by one second for the automatic mode (an alternative for switches)
* Adding another switch that can let the user choose the shape of the wave (triangle/seesaw)
* The addition of another working mode: **Blinking mode**

Justification for the Chosen Solution

When designing the project, I opted for a (heavily) structural description of the components, especially the TOP LEVEL one, that contains all of the other components presented so far. While the code may look long and complicated in certain parts, it actually just consists of multiple instantiations of the same component (I am referring especially to the test mode of operation). However, the code is very thorough and I was willing to sacrifice simplicity for it in order for the entire project to work as intended. Additional comments here and there may help the user understand the code better. Moreover, the information displayed on the seven segment display as well as the suggestive names of the inputs (and the outputs) further contribute to this matter.

Bibliography

* <https://en.wikipedia.org/wiki/>
* <https://www.fpga4student.com/>
* <https://steamcommunity.com/>
* <https://www.youtube.com/>
* <https://digilent.com/>