DIGITAL ELECTRONICS NOTEBOOK

ALEJANDRO LÓPEZ RODRÍGUEZ

Universidad Politécnica de Valencia
ETSID

MAY 15, 2020

Contents

Part I
Programmable Logic Devices

Laboratory Lecture 1: VHDL

In this lab lecture we will design a logic circuit that will act as an interface between the output of an ADC, a 4-bit value, and an array of LEDs so as to represent said value in a visual manner.

To better understand the circuit that we will be working with, we will start by drawing a simple diagram:

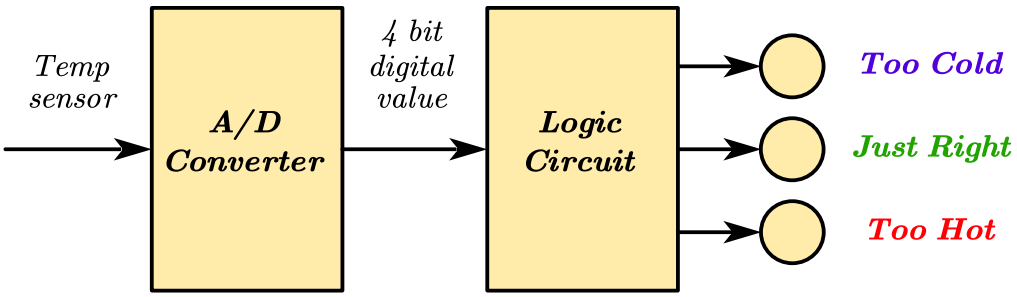


Figure 1: Circuit block representation.

As we can see in the diagram, there is a temperature sensor which output is fed into an analog-to-digital converter. The latter outputs a 4 bit digital value which serves as input to the logic circuit that we have to design. This circuit will interpret the data and, based on it, it will turn on a specific LED to indicate the temperature range that the sensor is measuring, being these states the ones indicated in the diagram

To aid the design process, a table containing the possible output values of the ADC and their corresponding meaning is given:

Digital Values	Category
0000-1000	Too cold
1001-1010	Just right
1011-1111	Too hot

Table 1: Possible states.

Our job will consist in programming the logic circuit in order to obtain the desired behaviour. In this practice, as well as in the rest of the subject, we will make use of the GAL22V10. This IC, commonly found in the DIP package, though old, is still a very good choice for beginners, due to its simplicity and ease of use.

To program the circuit, we will use the proprietary software designed for it, IspLever Classic. This program will allow us to synthesize and implement our VHDL code into the GAL SPLD. The main advantage of PLDs and FPGAs is the ability to program, using code, the HARDWARE part of our circuit, allowing us to obtain higher switching speeds and a faster response compared to what we would obtain if we were to use a microcontroller.

Writing some pseudocode before programming our circuit will surely come in handy later, as it is a nice way of organising the code and its different parts.

For this simple case, we came up with the following:

---

if digital value ≤ 8 **then**
  light only the Too Cold indicator.
else if $8 <$ digital value < 11 **then**
  light only the Just Right indicator.
else
  light only the Too Hot indicator.
end if

Before starting programming our circuit, we will first define the basic parts of any VHDL program:

1.1 Library

One of the most important parts of any VHDL program is the inclusion of several, important libraries.**Libraries** are pre-written chunks of code that allow us to focus on the development of our code without having to worry too much about the technical and laborious parts of the language itself.

The structure of this part is as follows:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
```

Here we can find a library clause, for instance, **library ieee;** , and a use clause, **use ieee.std_logic_1164.all;**, among others. This gives the code access to all the names declared within package **std_logic_1164** in the library **ieee**, and to data type **std_logic** in particular.

We can include other libraries such as **ieee.std_logic_arith** which defines some types and basic arithmetic operations for representing integers in standard ways.

1.2 Entity

An **entity** can be though of as a black box with inputs and outputs. The entity declaration includes the **name** of the entity, and a set of port declarations. A **port** may correspond to a pin on an IC, an edge connector on a board, or any logical channel of communication with a block of hardware.

Each port declaration includes the **name of one or more ports**, the **direction** that information is allowed to flow through the ports **(in, out or inout)**, and the data type of the ports (i.e., **std_logic**).

The structure of this part is as follows:

```
entity NAME_ENTITY is
  port(NAME_OF_PORT_1:  DIRECTION  DATA_TYPE;
                        . . .
       );
end NAME_ENTITY;
```

1.3 Architecture

The **architecture** is no longer a definition of parameters but the code itself. The architecture describes the design and is bounded to the entity.

The syntax for VHDL architecture is as follows:

```
architecture ARCH of NAME_ENTITY is
--begin
--  process(sensitivity list)
  begin
    concurrent/sequential instructions
end ARCH;
```

In VHDL, it is possible to find more than one architecture. Depending on the complexity of the actions that need to be performed by the PLD/FPGA, we may use concurrent statements, sequential ones -with **processes**-, or both.

The architecture has two parts. The declaration part, between the keywords ***architecture*** and ***begin***, in which the interconnection signals, other components referenced by this architecture, or constants are defined and a second part, which starts after the keyword ***begin*** that includes the statements and assignments and structure of the design.

Now that every part of the code has been tacked, we will write the actual code that we will program into the GAL:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity TEMP is
  port(temp: in std_logic_vector (3 downto 0);
       blue: out std_logic;
       green: out std_logic;
       red: out std_logic
      );
end TEMP;

architecture SENSOR of TEMP is
  begin
    compare: process(temp)
      begin

        if(temp <= "1000") then
          blue <= '1';
          green <= '0';
          red <= '0';

          elsif(temp >= "1001" AND temp <= "1010") then
            blue <= '0';
            green <= '1';
            red <= '0';

            else
              blue <= '0';
              green <= '0';
              red <= '1';

        end if;
    end process;
end SENSOR;
```

In order to fully understand the code, we will break it down into small chunks.

```
entity TEMP is
  port(temp: in std_logic_vector (3 downto 0);
       blue: out std_logic;
       green: out std_logic;
       red: out std_logic
      );
end TEMP;
```

As indicated in Figure 1, we will make use of 4 inputs, tied together in a vector, and 3 outputs, which will be connected to our LEDs. This concludes the entity part of our code.

Now, we will analyse its architecture:

```
architecture SENSOR of TEMP is
  begin
    compare: process(temp)
      begin

        if(temp <= "1000") then
          blue <= '1';
          green <= '0';
          red <= '0';

          elsif(temp >= "1001" AND temp <= "1010") then
            blue <= '0';
            green <= '1';
            red <= '0';

            else
              blue <= '0';
              green <= '0';
              red <= '1';

        end if;
    end process;
end SENSOR;
```

As we can see, this architecture is sequential, i.e. the statements don't occur at the same time but one after the other. This is illustrated in line 3, in which the keyword **process** and a sensitivity list containing the vector **temp** are introduced. Every time that the value of **temp** changes, the code linked to that process is run. In our case, this code checks the value of the temperature and turns one of the three LEDs on based on the reading.

After programming the code, it is time to compile it using **IspLeverClassic**. To do this, we will click on **Create Fuse Map** and wait for it to finish. Once compiled, we will flash the JEDEC file, which contains the fuse arrangement, into the GAL. The set up process won't be included into the reports as it is quite long and not really worth the explanation.

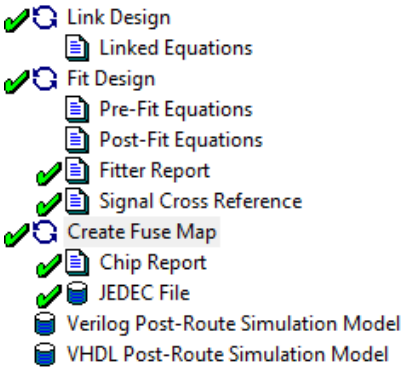


Figure 2: ISPLever.

Due to the simplicity of the GAL, the pin assignments are automatically performed by the software, so we have to check which pins the compiler decided to use. To do this, we will make use of the **Chip Report** pop-up menu.

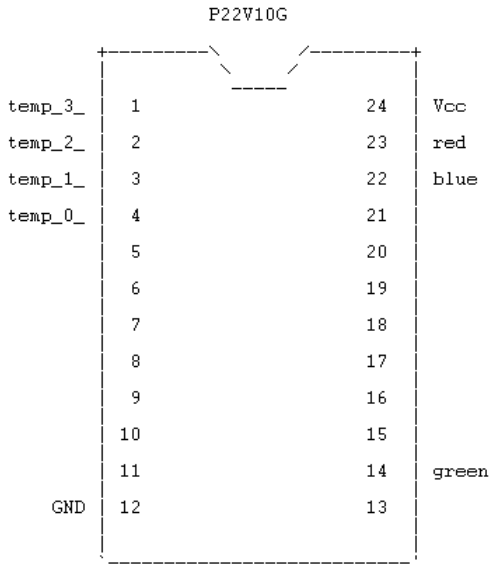


Figure 3: Chip Report Output.

Once everything checks, we will move on to simulating the circuit before finally assembling it. To do this, we will make use of ISIS Proteus, a well-know electronics simulation software.

Knowing the pin assignments makes the task of simulating the circuit very simple, as the only thing required is to follow the **Chip Report's** connections in Proteus.

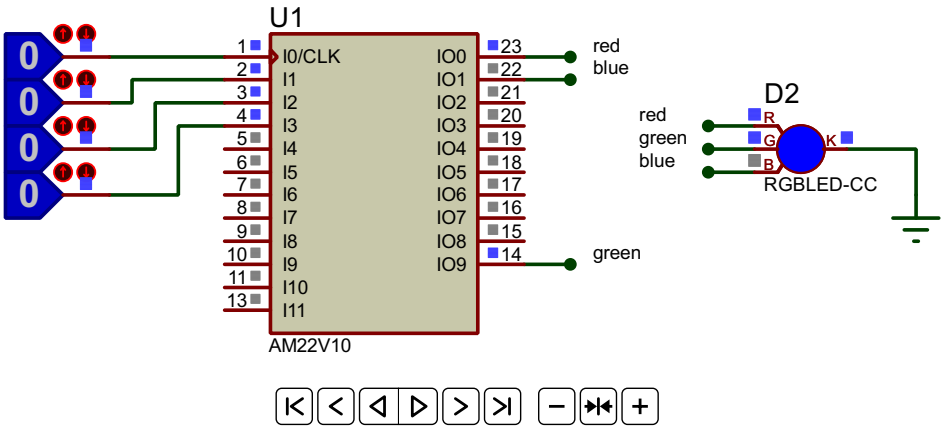


Figure 4: Proteus assembly.

Now that we have checked that the circuit works as intended, we can proceed to the real assembly. Again, this is just a matter of following the previous connections but this time using a protoboard and some hook-up wires.

As expected, everything worked great.

Laboratory Lecture 2: Flip Flops

The main objective of this lab lecture is to learn about the operation of Flip Flops. In particular, we will discuss how JK flip flops work as well as some of their applications.

2.1 Introduction

A flip flop, also known as latch or bistable multivibrator, is a type of circuit that has two states, one of them represents a *one* and the other one a *zero*, i.e. a single bit of data. They are commonly used to store information in digital circuits.

Flip flops can be edge-triggered, that is synchronous/clocked, or level triggered, that is asynchronous. In order to control them, we have to apply specific signals to the inputs, following their truth table.

Some examples of flip flops may include D Flip flops, D Latch Flip flops, SR Flip flops, and JK Flip flops, to name a few. In this practice we are going to work with the latter.

2.1.1 JK Flip Flops

Before diving into what a JK Flip Flop is, we will briefly talk about SR Flip Flops, as they are the old and troublesome early version of a JK one.

SR Flip Flops, also known as SR Latch, are one the most basic sequential logic circuits. They act as a one-bit memory, so a bistable device, that possesses 2 inputs. On the one hand, *SET*, which will output a 1 and is usually labelled *S*, and *RESET*, which will output a 0, and is usually labelled *R*.

Note than we can also find a clocked, or synchronous variation of the SR Flip Flop. The latter will have an extra input, *CLK* and it will only trigger on the positive edge transitions of the clock.

Obviously, *SR* stands for "Set-Reset". As we have previously said, the reset input resets the flip flop, that is, it makes the output, Q, go back to its original state of 0. The different input configurations will define the behaviour of the output following this fashion:

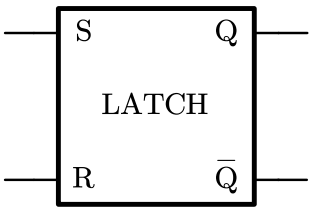


Figure 5: SR Asynchronous Flip Flop.

SET	RST	Q
0	0	Q_0 No change
1	0	Q = 1
0	1	Q = 0
1	1	Invalid

Table 2: SR Flip Flop's Truth Table.

As we can deduce from the table, this type of flip flops pose a problem when both the S and the R inputs are 1, since the output state is invalid and cannot be predicted.

To solve this problem, we will make use of a **JK Flip Flop**. JK Flip flops, in a nutshell, solve this problem by having the output toggle when both inputs J and K, which correspond to the S and R terminals respectively, are 1. The truth table of this type of flip flop is as follows:

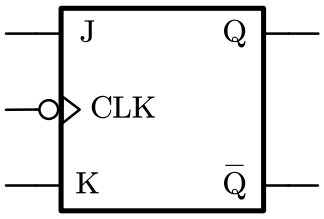


Figure 6: JK Synch. Flip Flop.

J	**K**	**CLK**	**Q**
0	0	$\uparrow$	Q_0 No change
1	0	$\uparrow$	Q = 1
0	1	$\uparrow$	Q = 0
1	1	$\uparrow$	$\overline{Q_0}$ Toggles

Table 3: JK's Synch. Truth Table.

It is possible to find an asynchronous version of the JK Flip Flop as well. This version has two extra pins, *PRESET* and *CLEAR*, which, if active, will turn the output Q to 0, if the *PRESET* is connected to 1 and the *CLEAR* to 0. Alternatively, they will turn the output Q to 1 if the *PRESET* is connected to 0, and the *CLEAR* to 1. We can visualise this in the following truth table:

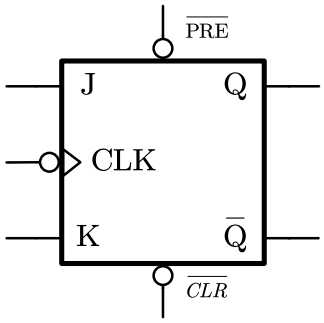


Figure 7: JK Asynch. Flip Flop.

J	**K**	**CLK**	$\overline{\textbf{PRE}}$	$\overline{\textbf{CLR}}$	**Q**
0	0	$\downarrow$	1	1	Q_0
1	0	$\downarrow$	1	1	1
0	1	$\downarrow$	1	1	0
1	1	$\downarrow$	1	1	$\overline{Q_0}$
X	X	X	0	0	Inv
X	X	X	0	1	1
X	X	X	1	0	0

Table 4: JK's Asynch. Truth Table.

Now that we have established the basics, we will move on to completing the laboratory session.

2.2 JK Synchronous Flip Flop

2.2.1 Asynchronous *PRE* and *CLR* effect on Q

In order to complete this part, we will make use of the program Proteus, as per usual. We will simply follow Table 4 in order to check the output when *PRE* and *CLR* change.

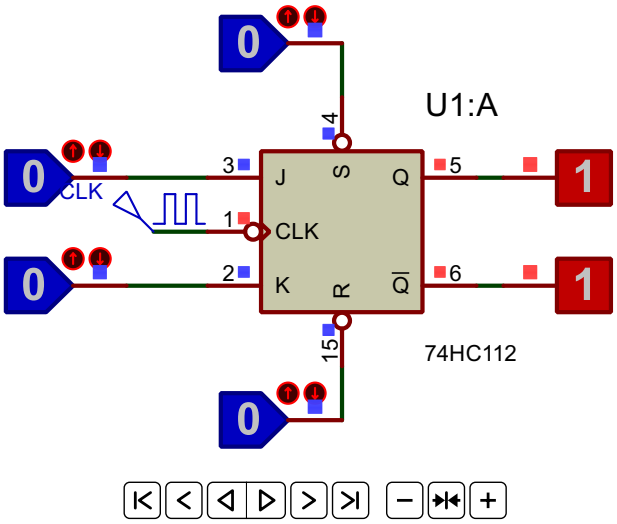


Figure 8: Proteus assembly.

In the simulation, we can not only see that the output, Q, is correct, but that it also does not depend on the values of J, K, and CLK.

2.2.2 Synchronous *J* and *K* effect on Q

For this part, as we are working with a Synchronous circuit, both *PRE* and *CLR* have to be tied to their inactive state, in this case, since they are active low, we will tie them to 1.

Again, we have to check that the circuit behaves as expected, that it, that it follows Table 3

To achieve this, we will use Proteus once more.

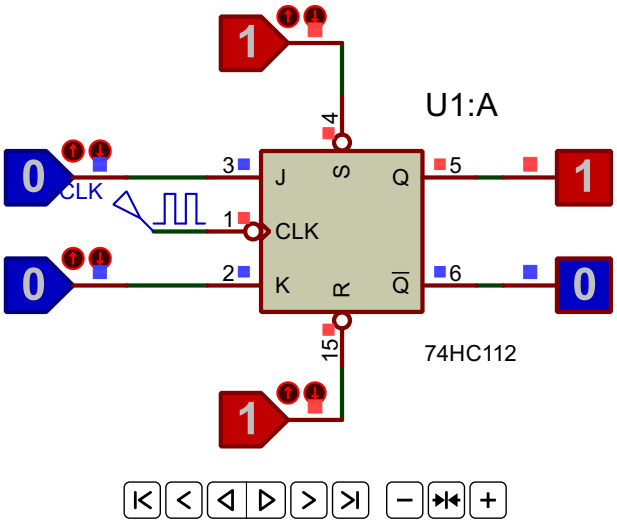


Figure 9: Proteus assembly.

As we can see, the circuit behaves as expected.

2.3 T Flip Flop from JK Flip Flop

In this exercise we are asked to create a T Flip Flop using a JK one. To do this we have to take Table 3 into account, as it clearly states that when both inputs, J and K are tied to 1, and a falling edge of the clock occurs -in the case of a 74HC112-, the output toggles, which is precisely what we aim to obtain.

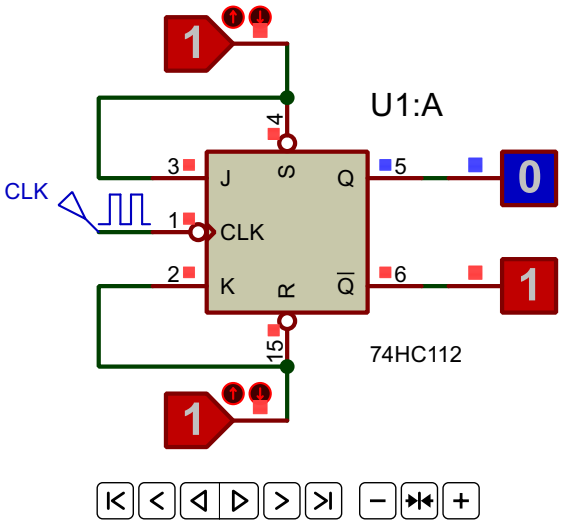


Figure 10: Proteus assembly.

One of the most important applications of this type of gate is to build a frequency divider. Since we are using a 74HC112 as a T Flip Flop, if we apply a CLK signal with a duty cycle of 50% to the clock input, the output Q will toggle every time a falling edge occurs, that is, every full cycle, effectively dividing the input frequency by 2. We can visualise this in the following timing diagram:

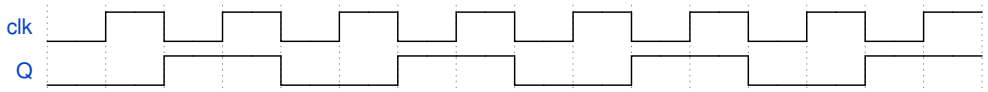


Figure 11: Timing diagram of a 2-bit frequency divider.

2.4 T Flip Flop Asynchronous counter

Now that we have established how to build a frequency divider, we will focus on its applications. In particular, we will build a binary counter, which will be fed to a BCD to 7 segment display decoder, so as to help us visualise the output.

In Figure 11, we can already see the the behaviour that we are aiming for. In the first division, the Q and the clk signals have a logic level of 0, or 00 in binary. This value turns into a 01 in the second division, 10, in the third, and finally 11 in the fourth. Translating these numbers into decimal yields 0, 1, 2, 3, so we can say that Figure 11 represents a 2 bit binary counter.

In order to be able to display numbers up to 7, we will need a 3 bit binary counter. Building it is just a matter of concatenating 2 T Flips Flops following this fashion:

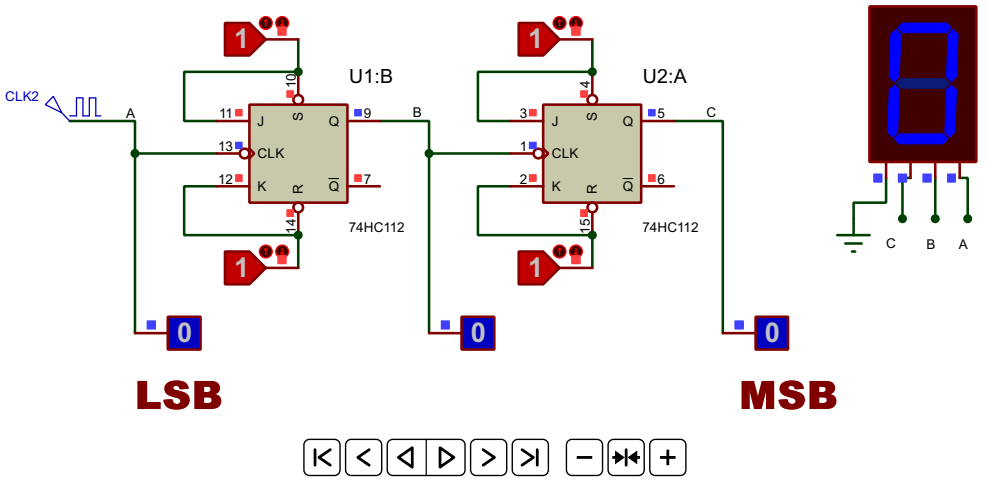


Figure 12: Proteus assembly.

If we look at the output signals with a logic analyser, we will see the following:

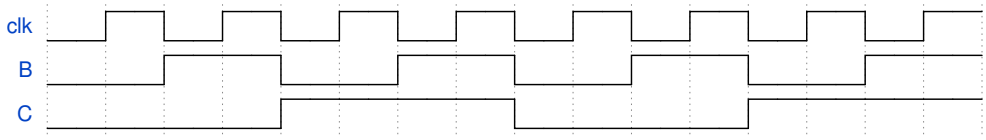


Figure 13: Timing diagram of a 3-bit frequency divider.

The only problem that this configuration poses, is that a small glitch is produced at the output due to the propagation time of the *CLK* signals, as the output of the first T Flip Flop is fed into the input of the next one and so on.

One possible solution would be to add a capacitor connected to ground to each of the 3 outputs, in order to smooth out the peak and make the transition more visually appealing and less prone to error.

Laboratory Lecture 2 BIS: 555 Timer

3.1 Introduction

Up until this point we have seen how to manipulate clock signals by using frequency dividers, but what if we want to obtain a specific type of output response, for instance a single pulse or a clock with a duty cycle different from 50% ? In this case, a multivibrator circuit is needed.

These type of integrated circuits use an RC timer to set the pulse duration and, depending on the manufacturer, they may work in different modes, i.e.:

1. Clock generator circuits (Astable Multivibrator)
2. One-Shot (Monostable Multivibrator)
    - Retriggerable
    - Non-Retriggerable

3.1.1 Astable Multivibrator

The first type of circuit, the **Astable Multivibrator**, is what is commonly referred to as **"Clock"**. These simple circuits basically switch back and forth between two unstable states with a specific duty cycle[1] set by an RC circuit.

In the previous section we have seen how modify a clock signal to obtain a specific output frequency, taking into account the limitations of an even division of course. The problem with this type of circuits is that they cannot provide custom duty cycles, which are needed in real case scenarios. That is when astable circuits come in handy.

3.1.2 Monostable Multivibrator

On the other hand, we can also find **Monostable Multivibrators**, commonly referred to as **"One-Shots"**. These type of circuits, as the name suggests, only output a single pulse when triggered, that is, they have only one stable state. The change from stable to quasi-stable occurs for a fixed time period $\mathbf{t_p}$ or $\mathbf{t_w}$ which is determined by an RC constant as well.

These circuits are readily available and they usually come in two forms, i.e. a retriggerable and a non-retriggerable one. When a retriggerable one-shot is triggered before the end of the pulse, the pulse duration $\mathbf{t_p}$, is restarted. Contrarily, when a non-retriggerable monostable is triggered before the end of the pulse, the output remains the same, in other words, the input is ignored until the output returns to the steady state.

[1] The duty cycle of a digital circuit is the percentage of the ratio of pulse duration, or pulse width to the total period. It can be expressed as:

$$D = \frac{t_{ON}}{t_{ON} + t_{OFF}} \cdot 100$$

We can visualise their behaviour in the following graphs:

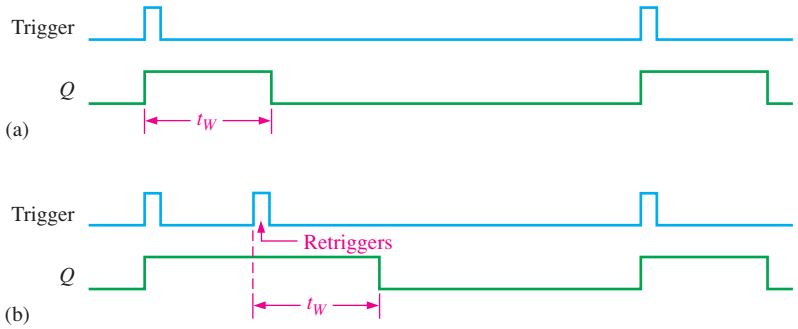


Figure 14: Retriggerable Monostable. [1]

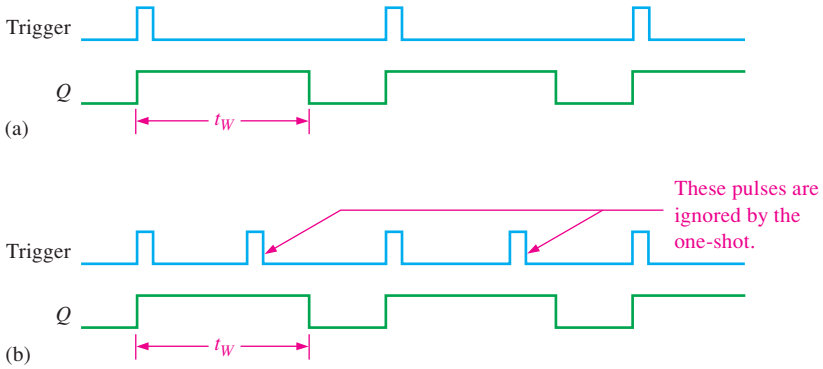


Figure 15: Non-Retriggerable Monostable. [1]

Both monostables, as well as the astable one, can be implemented using VHDL, but we will not dive into this, as it is not required in this practice, though they can be found **here**.

3.2 555 Timer

Now that every type of multivibrator has been introduced, we will discuss the 555 timer. This timer is a TTL-compatible device that can operate in both of the modes described above. The heart of the 555 timer is composed of two voltage comparators and a SR Latch (See 5).

The comparators are devices whose outputs are HIGH when the voltage on the positive (+) input is greater than the voltage on the negative (-) input and LOW when the (-) input voltage is greater than the (+) input voltage.

The voltage divider consisting of three 5 kΩ resistors provides a trigger level of $\frac{1}{3}$ VCC and a threshold level of $\frac{2}{3}$ VCC. The control voltage input (pin 5) can be used to externally adjust the trigger and threshold levels to other values if necessary.

When the normally HIGH trigger input momentarily goes below $\frac{1}{3}$ VCC, the output of comparator B switches from LOW to HIGH and sets the S-R latch, causing the output (pin 3) to go HIGH and turning the discharge transistor Q1 off.

The output will stay HIGH until the normally LOW threshold input goes above $\frac{2}{3}$ VCC and causes the output of comparator A to switch from LOW to HIGH. This resets the latch, causing the output to go back LOW and turning the discharge transistor on. The external reset input can be used to reset the latch independent of the threshold circuit. The trigger and threshold inputs (pins 2 and 6) are controlled by external components connected to produce either monostable or astable action. [1]

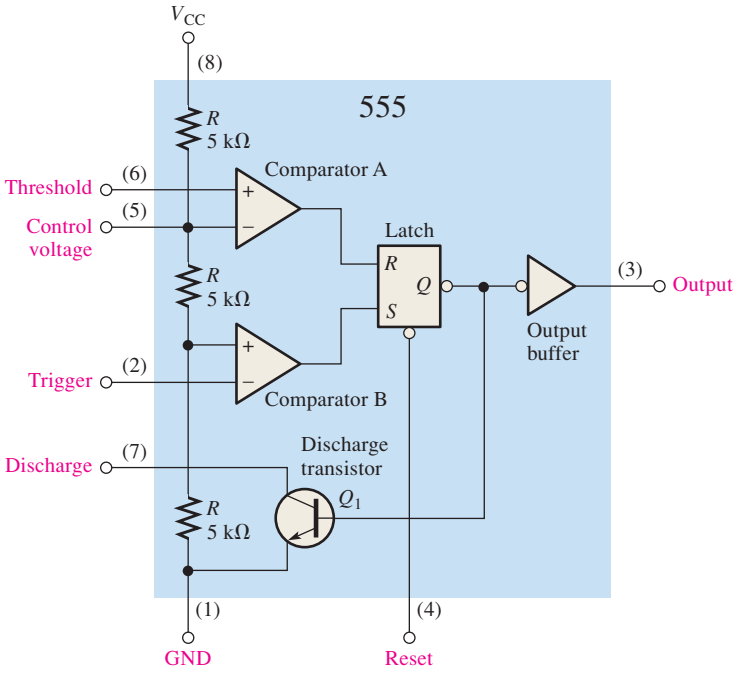


Figure 16: Internal functional diagram of a 555 timer. [1]

3.2.1 Astable 555

As we have mentioned before, the 555 timer can be configured as a basic **Astable Multivibrator** following the circuit down below:

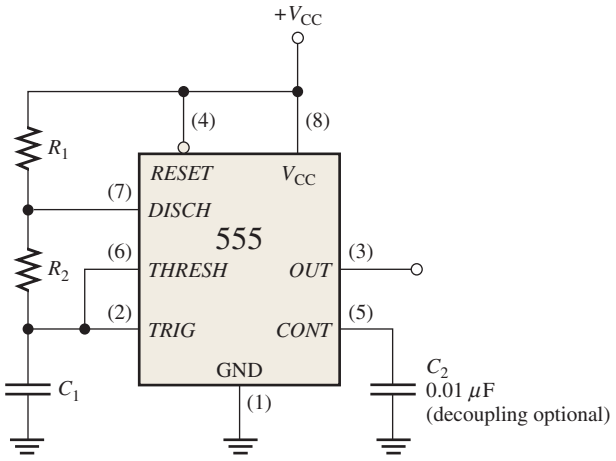
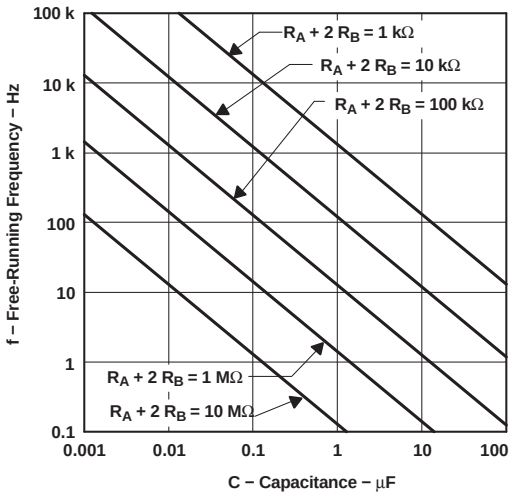


Figure 17: 555 timer connected as an astable multivibrator (oscillator). [1]

In this circuit C1 charges through R1 and R2 and discharges through only R2. The output frequency is given by:

$$f = \frac{1.44}{(R_1 + 2 \cdot R_2) \cdot C_1}$$

In order to help us find a set of suitable component values, the manufacturer provides a chart that shows sets of compatible and valid configurations. Besides, some useful equations are provided:



$$t_H = 0.693 \cdot (R_1 + R_2) \cdot C_1$$
$$t_L = 0.693 \cdot (R_2) \cdot C_1$$
$$T = 0.693 \cdot (R_1 + 2 \cdot R_2) \cdot C_1$$

Figure 18: 555 Astable Freq. Chart. [2]

3.2.2 Monostable 555

The 555 timer can also be used as a **Monostable Multivibrator** following the circuit down below.

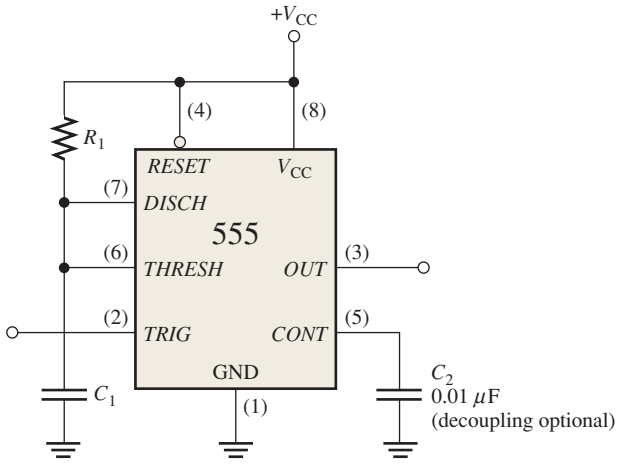


Figure 19: 555 timer connected as an monostable multivibrator (One-shot). [1]

As per the other configuration, the duration of the pulse $\mathbf{t_p}$ or $\mathbf{t_w}$ can be determined by the next equation:

$$t_p = 1.1 \cdot R_1 \cdot C_1$$

For this configuration, the trigger is a NGP (Negative-going pulse). In the manufacturer's datasheet we can find a chart similar to the last one:

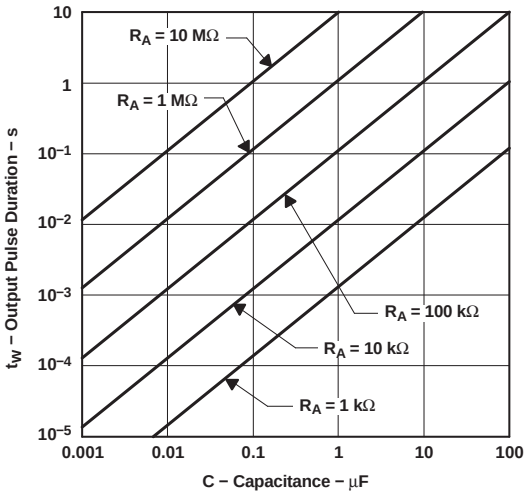


Figure 20: 555 timer Monostable Frequency Chart. [2]

3.3 Exercise 1: 555 as Astable Multivibrator

Design an astable multivibrator by using 555 timer with $C = 10$ nF, $R_1 = 10$ kΩ y $R_2 = 10$ kΩ. Calculate theoretically the value of t_H (high level semi period), T (period) and DC% (duty cycle).

Draw and simulate the circuit. Obtain the graphics of the outputs and between the pins of the capacitor. Measure the values of t_H (high level semi period), T (period) and DC% (duty cycle).

Answer to Exercise 1:

Using the equations listed in 18 and 1, obtaining what we are asked is rather simple:

$$\mathbf{t_H} = 0.693 \cdot (R_1 + R_2) \cdot C_1 = 0.693 \cdot (10k\Omega + 10k\Omega) \cdot 10\text{nF} = \mathbf{138.6}\,\mu\text{s}$$

$$\mathbf{t_L} = 0.693 \cdot (R_2) \cdot C_1 = 0.693 \cdot (10k\Omega) \cdot 10\text{nF} = \mathbf{69.3}\,\mu\text{s}$$

$$\mathbf{T} = t_L + t_H = \mathbf{207.9}\,\mu\text{s}$$

$$\mathbf{D} = \frac{t_H}{t_H + t_L} \cdot 100 = \frac{\mathbf{138.6}\,\mu\text{s}}{\mathbf{138.6}\,\mu\text{s} + \mathbf{69.3}\,\mu\text{s}} \cdot 100 = \mathbf{66.6}\,\%$$

We are also asked to simulate the circuit. To do this we will make use of ISIS Proteus, as we have done in the past.

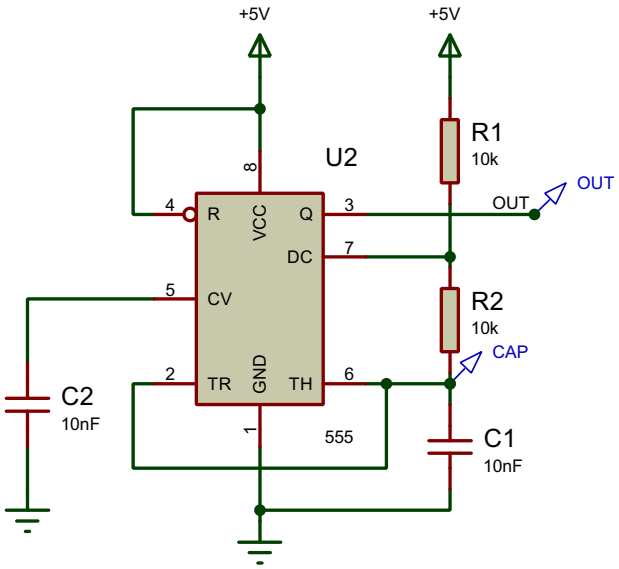


Figure 21: Proteus assembly of the first subsection with $R_2 = 10k\Omega$.

To check the output, we will employ the Analogue Analysis tool:

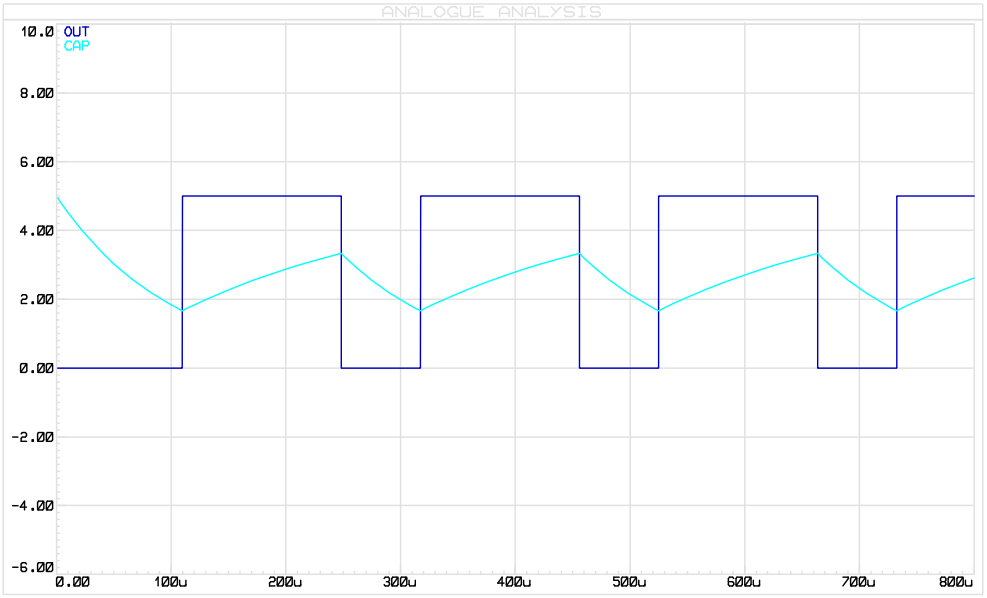


Figure 22: Analogue Analysis of circuit with $R_2 = 10k\Omega$.

Using the cursors, we can measure the required parameters:

$$\mathbf{t_H = 207\,\mu s} \quad \mathbf{t_L = 69\,\mu s} \quad \mathbf{T = 207\,\mu s} \quad \mathbf{D = 66.6\,\%}$$

As we can see, they match the calculations perfectly since proteus doesn't take into account the tolerances of the different components. The same simulation in NI Multisim yields vastly different results, due to its mathematical models of components being more accurate.

The exercise now asks us to change the value of R_2 to $100k\Omega$ and measure the same parameters. After following the same procedure, we obtain these results:

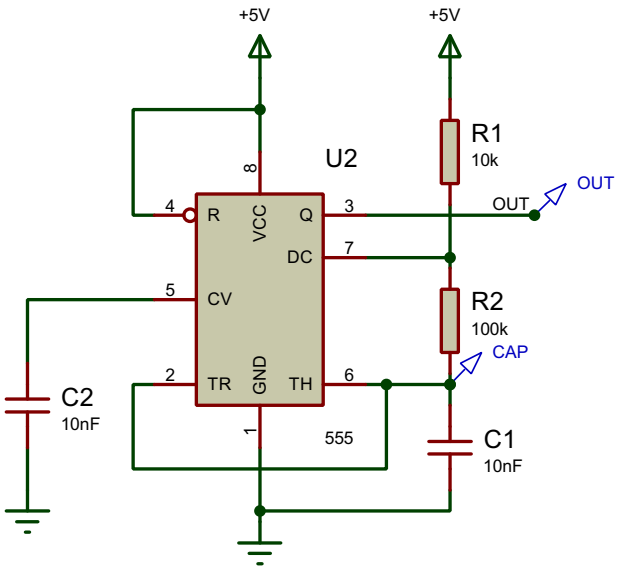


Figure 23: Proteus assembly of the second subsection with $R_2 = 100k\Omega$.

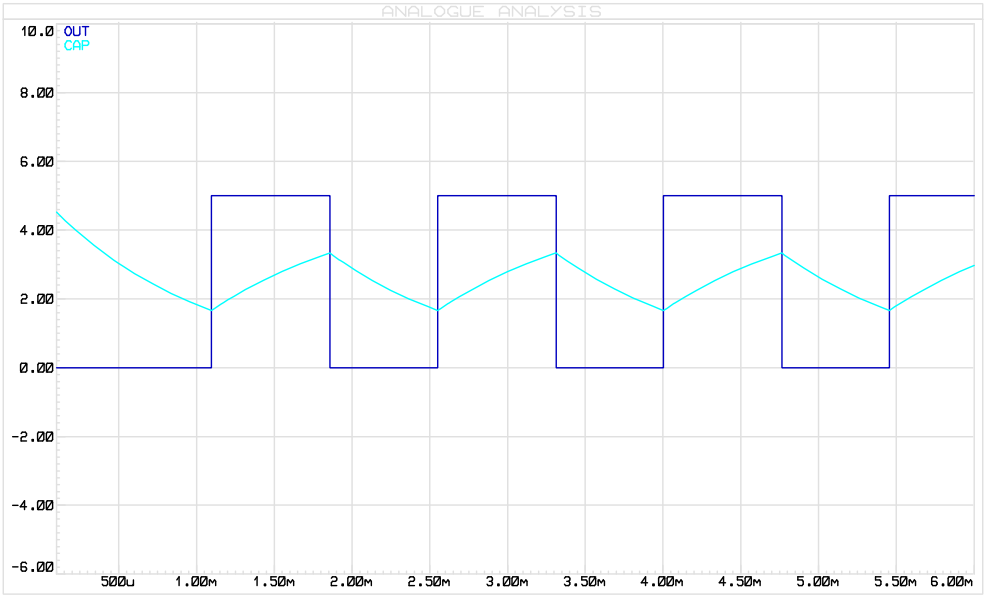


Figure 24: Analogue Analysis of circuit with $R_2 = 100k\Omega$.

$$\mathbf{t_H = 762\,\mu s} \quad \mathbf{t_L = 693\,\mu s} \quad \mathbf{T = 1455\,\mu s} \quad \mathbf{D = 52.37\,\%}$$

The exercise finally asks us to add a diode in parallel with R_2 and measure the same parameters. After following the same procedure, we obtain these results:

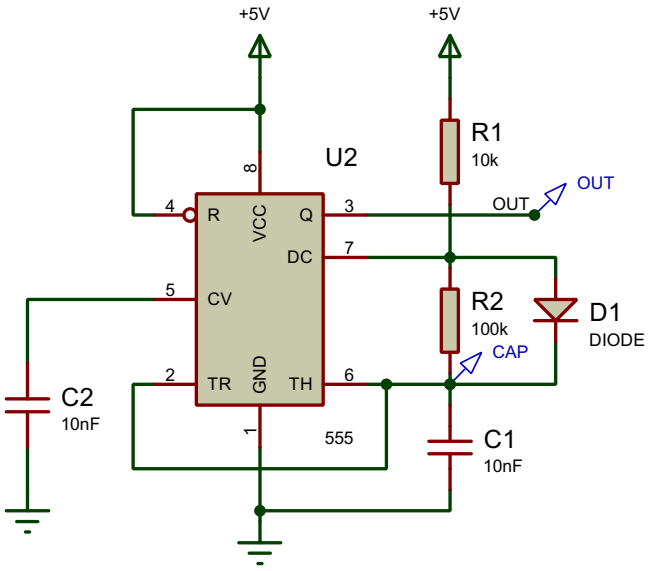


Figure 25: Proteus assembly of the third subsection with $R_2 = 100k\Omega$ and a diode in parallel.

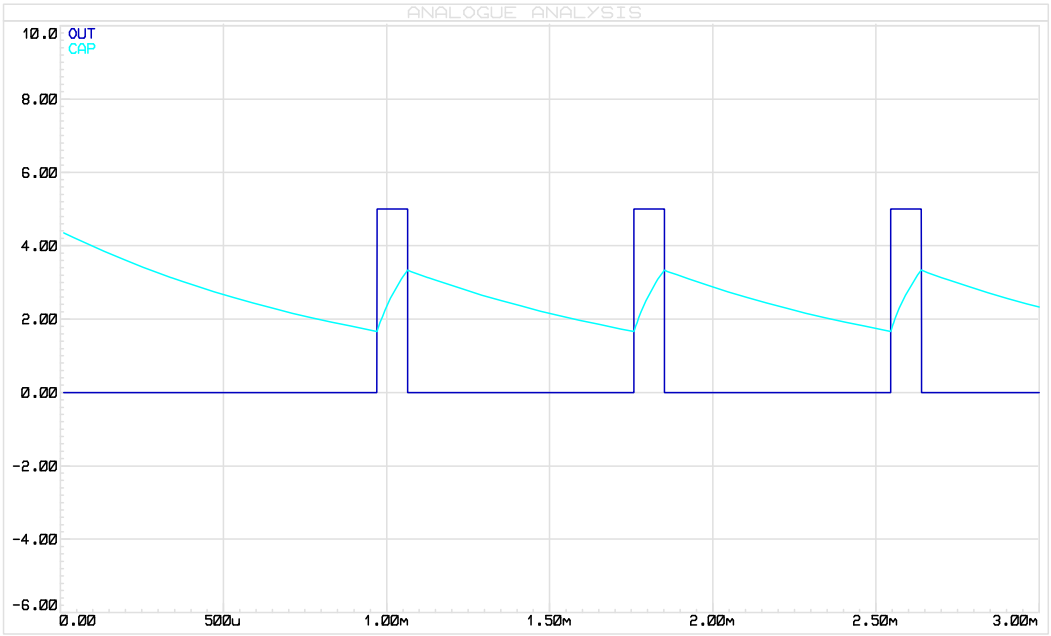


Figure 26: Analogue Analysis with $R_2 = 100k\Omega$ and Diode in parallel.

$$\mathbf{t_H = 98\,\mu s} \quad \mathbf{t_L = 690\,\mu s} \quad \mathbf{T = 788\,\mu s} \quad \mathbf{D = 12.44\,\%}$$

3.4 Exercise 2: 555 as Monostable Multivibrator

In this exercise we are asked to design a one-shot multivibrator (Figure 19)using a 555 timer, a 10 µF capacitor and a $100k\Omega$ resistor. As a reminder, the duration of the pulse of a 555 monostable can be obtained using this equation:

$$t_p = 1.1 \cdot R_1 \cdot C_1 = 1.1 \cdot 100k\Omega \cdot 10\mu F = 1.10s$$

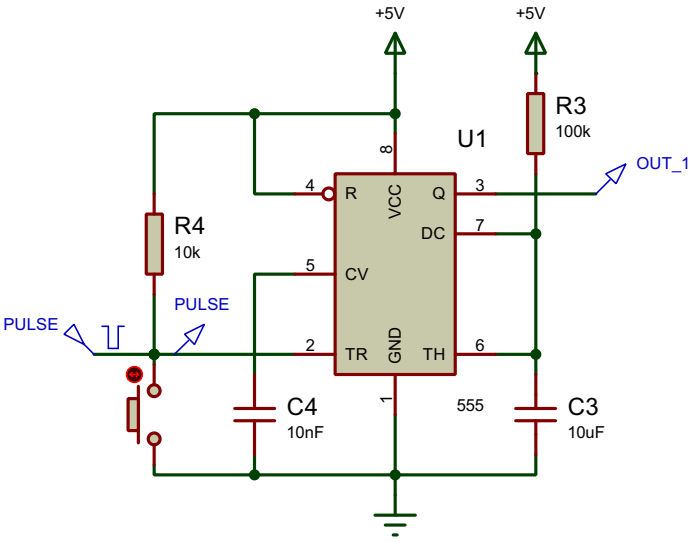


Figure 27: Proteus assembly of a Monostable Multivibrator

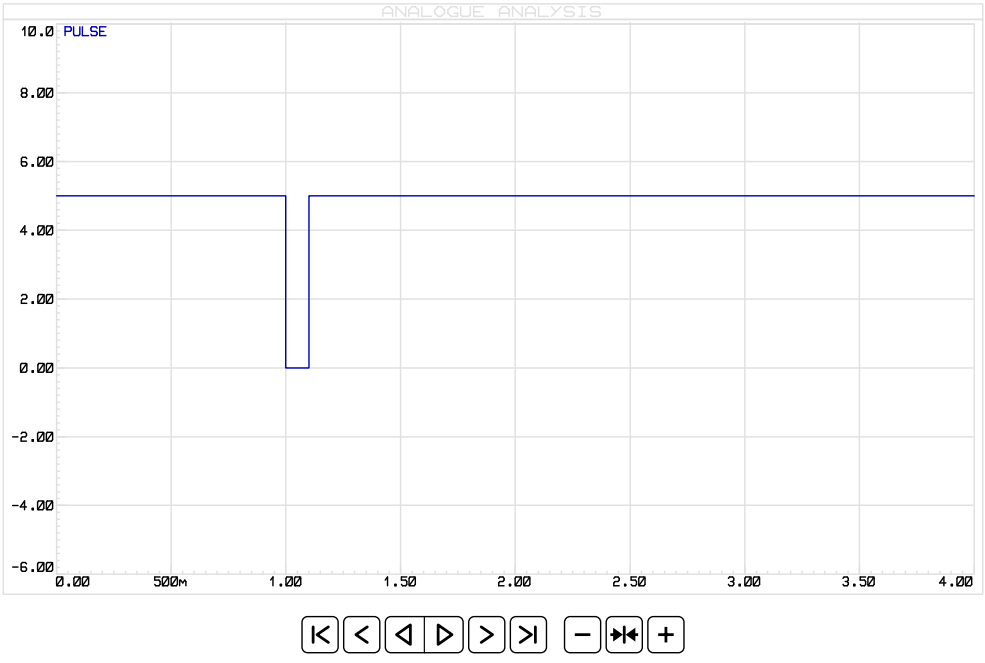


Figure 28: Analog Analysis of a Monostable Multivibrator.

Laboratory Lecture 3: Stepper Motor Controller

4.1 Introduction

The aim of this practice is to go over the basics of how to control a stepper motor. A stepper motor is a brushless DC electric motor that divides a full rotation in several small increments or *steps*. By providing power to its coils in a specific way, we can control the position of the shaft.

Stepper motors can be unipolar or bipolar. Unipolar Stepper motors are very similar to Bipolar Stepper Motors, but are manufactured with a central tap that connects back to the power source, essentially splitting each coil into two smaller coils that can be powered independently. If required, the central tap can also be left disconnected, allowing the Unipolar Stepper Motor to be converted into a Bipolar configuration.

Bipolar Stepper Motors do not feature a central tap for dividing their solenoid coils. This makes their internal wiring slightly less complex than that of a Unipolar Motor.

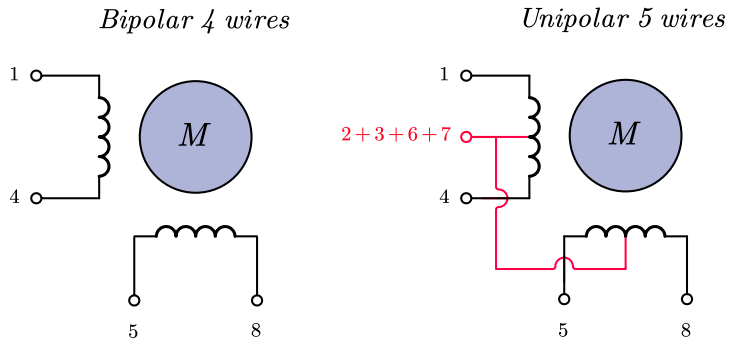


Figure 29: Bipolar vs. Unipolar configurations.

4.2 Stepper Motor Controller

In this practice we will use the Bipolar type, in particular a NEMA 17 motor. To control it we will use a special driver, the L293D, which is basically an array or darlington transistors, in half-H configuration, whose main objective is to take care of switching the high currents that the motors require.

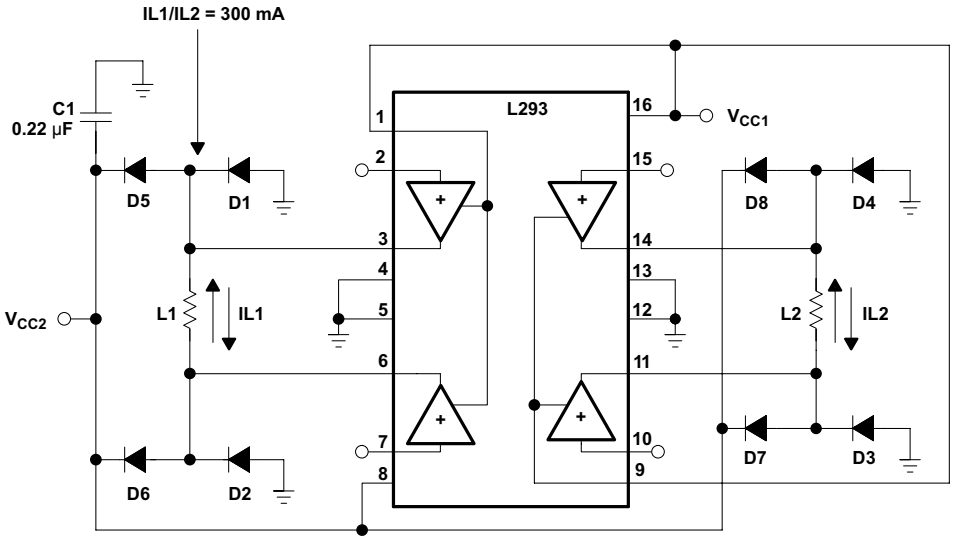


Figure 30: L293D Bipolar Stepper Controller. [3]

To control said driver, we will make us of the SPLD that we have been using since the beginning, the GAL22V10C and some basic VHDL code. In order to make the shaft spin in a controlled manner, we have to take into account the phase current waveforms, i.e. the pulses that we have to send to the driver to cause a step change and the order in which they must be sent. This information can be seen below:

Step	Full-Step [1.8°]	Wave-Step [1.8°]	Half-Step [0.9°]
0	1010	1000	1010
1	1001	0001	1000
2	0101	0100	1001
3	0110	0010	0001
4			0101
5			0100
6			0110
7			0010

Table 5: Phase Current Waveforms. [4]

4.2.1 VHDL Code

As we have seen in Table 5, there are 3 different ways in which we can control the rotation of the stepper motor, namely, full-step, wave-step and half-step. Each of them present advantages and disadvantages. For instance, *Half-Steps* can deliver a higher precision, since every step only spins the shaft 0.9°as opposed to *full-step* and *wave-step*, which spin it 1.8°. This increase in performance reduces the output torque significantly, so there is a clear trade-off between both variables that one must consider. In addition to this, *Wave-step* mode is more visually appealing, as the transitions between states are not as choppy as with the other ones.

Due to the limitations that the GAL22V10C possesses, it is not possible to fit the 3 control modes in the same SPLD. That's why we have developed 3 different codes, one for each, that can be seen below:

4.2.1.1 Full Step

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity STEPPER_FS is
5    port(CLK: in std_logic;
6         DIR: in std_logic;
7         MOTOR: out std_logic_vector(3 downto 0)
8         );
9  end STEPPER_FS;
10
11 architecture ARCH of STEPPER_FS is
12   begin
13     process(CLK)
14       variable COUNT: std_logic_vector(3 downto 0);
15         begin
16           if(CLK'EVENT and CLK='1') then
17
18             if(DIR='0') then
19               case COUNT is
20                 when "0000" => COUNT := "1010";
21                 when "1010" => COUNT := "1001";
22                 when "1001" => COUNT := "0101";
23                 when "0101" => COUNT := "0110";
24                 when "0110" => COUNT := "1010";
25                 when others => COUNT := "0000";
26               end case;
27
28
29
```

```
                else
                  case COUNT is
                    when "0000" => COUNT := "0110";
                    when "0110" => COUNT := "0101";
                    when "0101" => COUNT := "1001";
                    when "1001" => COUNT := "1010";
                    when "1010" => COUNT := "0110";
                    when others => COUNT := "0000";
                  end case;
              end if;
            end if;

            MOTOR <= COUNT;

    end process;
end ARCH;
```

As we have previously mentioned, *Full Step* phase control provides the highest torque of all, though the transitions tend to look quite choppy. This mode and the *Wave Step* one provide a maximum accuracy of 1.8° per step, which means that a full 360° rotation would require 200 steps.

The code that we have included moves the shaft one step (forwards or backwards) whenever a *PGT* occurs. This indicates that the process is synchronous. Besides, we can also find a direction pin which changes the direction of the spin when a positive signal is applied to it.

4.2.1.2 Wave Step

```
library ieee;
use ieee.std_logic_1164.all;

entity STEPPER_WS is
  port(CLK: in std_logic;
       DIR: in std_logic;
       MOTOR: out std_logic_vector(3 downto 0)
       );
end STEPPER_WS;

architecture ARCH of STEPPER_WS is
  begin
    process(CLK)
      variable COUNT: std_logic_vector(3 downto 0);
        begin
          if(CLK'EVENT and CLK='1') then

```

```
            if(DIR='0') then
              case COUNT is
                when "0000" => COUNT := "1000";
                when "1000" => COUNT := "0001";
                when "0001" => COUNT := "0100";
                when "0100" => COUNT := "0010";
                when "0010" => COUNT := "1000";
                when others => COUNT := "0000";
              end case;

                else
                  case COUNT is
                    when "0000" => COUNT := "0010";
                    when "0010" => COUNT := "0100";
                    when "0100" => COUNT := "0001";
                    when "0001" => COUNT := "1000";
                    when "1000" => COUNT := "0010";
                    when others => COUNT := "0000";
                  end case;
            end if;
          end if;

          MOTOR <= COUNT;

    end process;
end ARCH;
```

This code describes the *Wave Step* phase control sequence. Wave Stepping is usually used when the smoothness of the output rotation is important. As per the last mode, this mode has a high torque but its trade-off is a reduction in its accuracy.

The rest of the code remains the same as the last one.

4.2.1.3 Half Step

```
library ieee;
use ieee.std_logic_1164.all;

entity STEPPER_HS is
  port(CLK: in std_logic;
       DIR: in std_logic;
       MOTOR: out std_logic_vector(3 downto 0)
       );
end STEPPER_HS;

```

```
architecture ARCH of STEPPER_HS is
  begin
    process(CLK)
      variable COUNT: std_logic_vector(3 downto 0);
        begin
          if(CLK'EVENT and CLK='1') then

            if(DIR='0') then
              case COUNT is
                when "0000" => COUNT := "1010";
                when "1010" => COUNT := "1000";
                when "1000" => COUNT := "1001";
                when "1001" => COUNT := "0001";
                when "0001" => COUNT := "0101";
                when "0101" => COUNT := "0100";
                when "0100" => COUNT := "0110";
                when "0110" => COUNT := "0010";
                when "0010" => COUNT := "1010";
                when others => COUNT := "0000";
            end case;

                else
                  case COUNT is
                    when "0000" => COUNT := "0010";
                    when "0010" => COUNT := "0110";
                    when "0110" => COUNT := "0100";
                    when "0100" => COUNT := "0101";
                    when "0101" => COUNT := "0001";
                    when "0001" => COUNT := "1001";
                    when "1001" => COUNT := "1000";
                    when "1000" => COUNT := "1010";
                    when "1010" => COUNT := "0010";
                    when others => COUNT := "0000";
                  end case;
            end if;
          end if;

          MOTOR <= COUNT;

    end process;
end ARCH;
```

In addition, we can find the *Half Step* phase control sequence. As we can see, this code is longer than the last 2 codes. This is due to the fact that it has some intermediate states which help smooth out the rotation by getting rid of most of the chopping that characterizes the latter.

4.2.2 Proteus Simulation and Assembly

After discussing the code, we will move on to the simulation of the circuit. For this we will use ISIS Proteus once again. Even though Figure 30 may look cumbersome, in actuality, we only have to connect the 2 coils to the driver's output and the 4 direction pins to its inputs following this fashion:

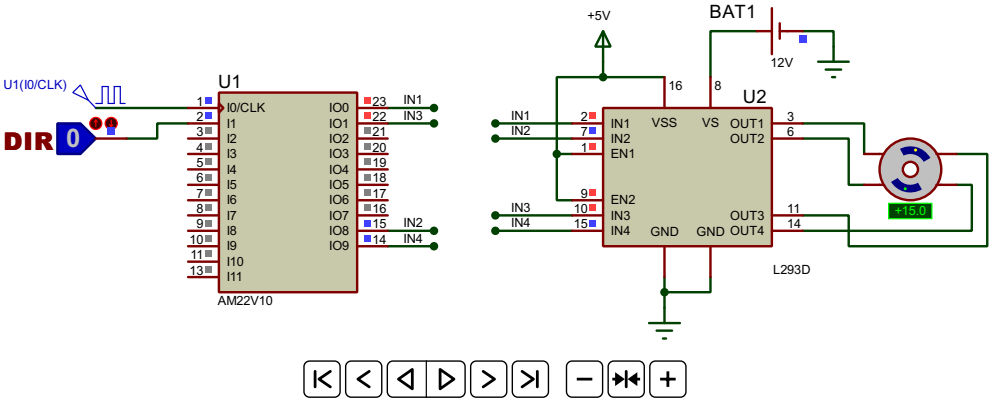


Figure 31: Stepper with rotation direction control.

The I/O pins that we used are the ones that the the compiler assigned by default. They are displayed in the Chip Report (Figure 3).

Assembling the circuit is just a matter of manually connecting everything following the schematic of Figure 31. To provide a clock signal we have used a signal generator which output has been set to a TTL level, 0 to 5V, and its frequency to 1 Hz.

Laboratory Lecture 4: Stepper Motor Controller

The main objective of this practice is to understand the fundamentals of Finite State Machines and their main applications. We will then use the knowledge that we have acquired to build a stepper motor controller which will be controlled by a rotary encoder.

5.1 Finite State Machines

Before diving into the practice, we must firstly define what a Finite State Machine. In essence, a FSM is a machine that makes predictable transitions through a sequence of states, based on external inputs and the current state of the machine. In digital electronics, the timing of the transition of the machine from one state to another is controlled by a register and a system clock, while the next state of the machine is determined by a combination of logic gates and embedded memory. The number of states, as the name suggests can be assumed to be finite. [5]

Two basic types of state machines are the Moore and the Mealy. The Moore state machine is one where the outputs depend only on the internal present state. The Mealy state machine is one where the outputs depend on both the internal present state and on the inputs. [1]

We can see and example of these type of machines here:

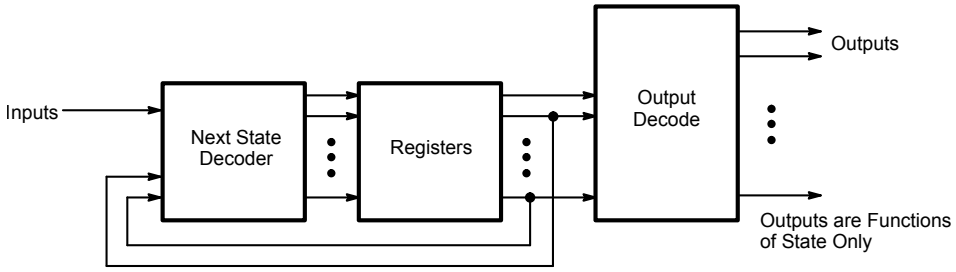


Figure 32: Moore State Machine Operation Diagram [6]

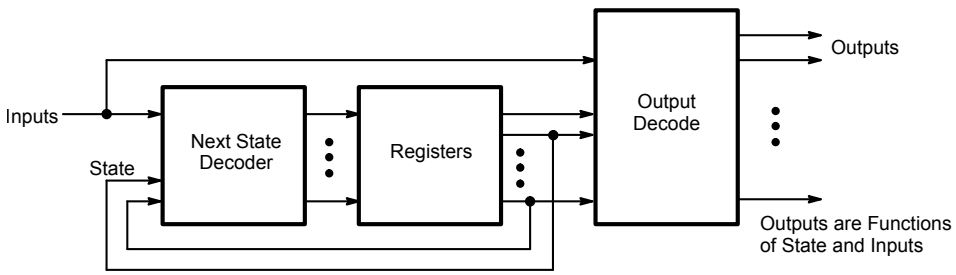


Figure 33: Mealy State Machine Operation Diagram [6]

To represent both types of machines, a State Diagram is used. Normally, state diagrams are composed of bubbles, which indicate the different states, and arrows, that indicate the transitions.

In the following example we will implement a FSM that recognizes the sequence "10" using both type of machines.

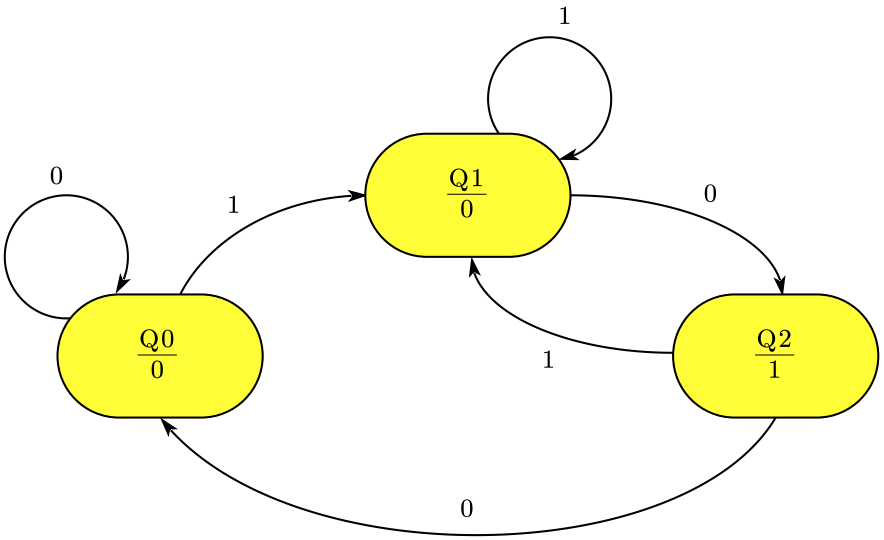


Figure 34: Moore State Machine [7]

For this particular case, $\boldsymbol{Q_0}$ means *No elements in the sequence observed*, $\boldsymbol{Q_1}$ means *"1" observed* and $\boldsymbol{Q_2}$ means *"10" observed*

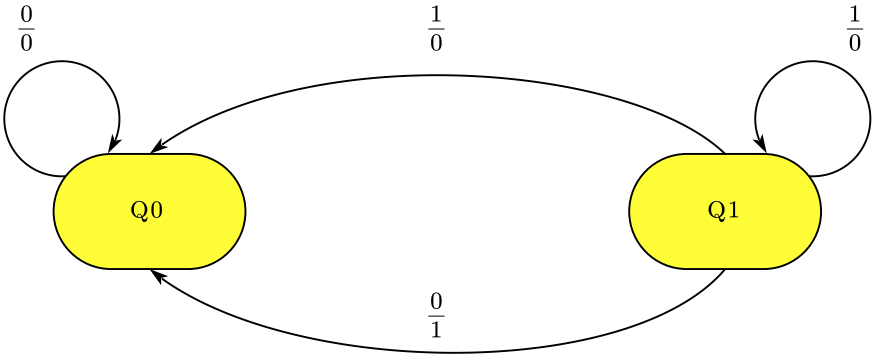


Figure 35: Mealy State Machine [7]

In the latter case, the states mean the same, but as we can seem there is one less state than in the previous one.

This brings us to the advantages and disadvantages that both of them have:

1. Mealy machines have **fewer states** and react faster to inputs.
    - Outputs are defined in the transitions (n^2) rather than in the state itself (n).
    - They react in the same cycle, i.e., they do not need to wait for the clock.
    - Their outputs may be considerably shorter than the clock cycle, which can pose problems.
    - Since fewer states than the **Moore** type are used, the propagation delay time is also shorter.
2. Moore machines are **safer** to use.
    - Outputs change at clock edge.
    - In **Mealy machines**, an input change will cause an output change as soon as the logic is done, which a big problem when two machines are interconnected since asynchronous feedback may occur if one is not careful.

5.2 Incremental encoder

An incremental encoder is a type of rotary encoder that indicates both the ocurrence and the direction of movement. An encoder of such type does not indicate the absolute position or angle; it only reports changes in position and for each position, the direction of movement.

They usually have 2 outputs, A and B, which value changes as the encoder is swept through its range, incrementing the count every time that its shaft changes position.

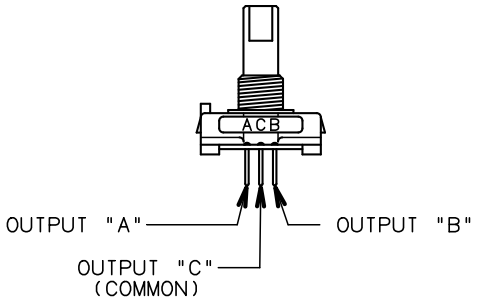


Figure 36: Encoder pinout [8]

Incremental encoders output a defined number of pulses per revolution. We can use these pulses to obtain both the angle and the direction of the rotarion since the outputs A and B are shifted 90°. This behaviour can be clearly seen in the following graph:

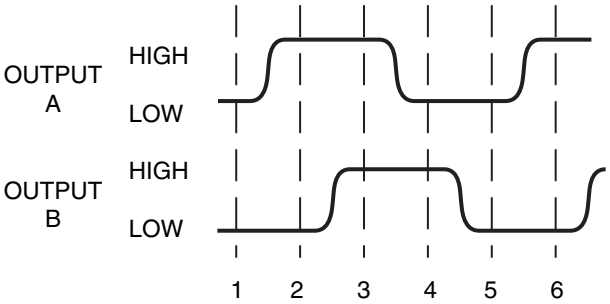


Figure 37: Encoder Output States [8]

5.3 Exercise 1: Motor Control

Now that we have defined how every part of the system works, we will move onto solving the laboratory lecture itself.

The first and only exercse states the following:

Design a state machine (FSM) that supplies the proper output to a stepper motor according to the position of the incremental encoder. Draw the FSM.

Answer to Exercise 1:

As per usual, we will implement the code for the FSM in VHDL and we will program it into a GAL22V10C.

The VHDL code can be found below:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity ENCODER is
5    port(CLK, RST: in std_logic;
6         AB: in std_logic_vector(1 downto 0);
7         STEPPER: out std_logic_vector(3 downto 0));
8  end ENCODER;
9
10 architecture ARCH of ENCODER is
11   type STATE_TYPE is (E0, E1, E2, E3);
12   signal STATE: STATE_TYPE;
13
14     begin
15       process(CLK, RST)
16         begin
17           if(RST = '0') then STATE <= E0;
18             elsif(rising_edge(CLK)) then
19
20               case STATE is
21                 when E0 => STEPPER <= "0110";
22                            if(AB = "01") then STATE <= E1;
23                              elsif(AB = "10") then STATE <= E3;
24                                else STATE <= E0;
25                            end if;
26                 when E1 => STEPPER <= "1010";
27                            if(AB = "11") then STATE <= E2;
28                              elsif(AB = "00") then STATE <= E0;
29                                else STATE <= E1;
30                             end if;
31                 when E2 => STEPPER <= "1001";
32                            if(AB = "10") then STATE <= E3;
33                              elsif(AB = "01") then STATE <= E1;
```

```
34                                  else STATE <= E2;
35                              end if;
36                  when E3 => STEPPER <= "0101";
37                              if(AB = "00") then STATE <= E0;
38                                elsif(AB = "11") then STATE <= E2;
39                                  else STATE <= E3;
40                              end if;
41              end case;
42          end if;
43      end process;
44  end ARCH;
```

As we can clearly see in the code, this FSM is of the type Moore, as the outputs update in the states themselves rather than in the transitions. In addition, since the capacity of the GALs is very limited, we have not added an asynchronous nor synchronous reset pin, which could cause problems in other, more complex FSMs.

The output signal **STEPPER** will be connected to the stepper motor driver that we saw in the last practice so as to simulate a similar version of the full-step control mode with fewer states (**See Subsubsection 4.2.1.1 for more on this**)

The State Diagram that describes this FSM can be found below:

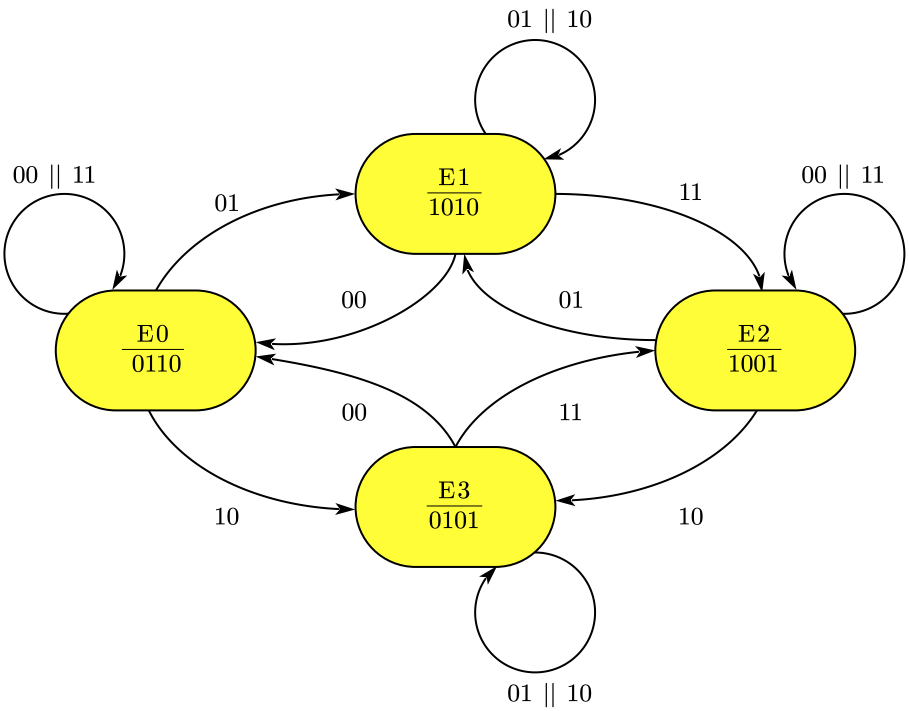


Figure 38: Stepper Controller with Incremental Encoder FSM

References

[1] T. L. Floyd, *Digital Fundamentals, Global Edition*, 11th ed. Harlow, United Kingdom: Pearson Education Canada, 2015.

[2] *xx555 Precision Timers*, NE555, Rev. 1, Texas Instruments, Sep. 2014. [Online]. Available: https://www.ti.com/lit/ds/symlink/ne555.pdf.

[3] *L293x quadruple half-h drivers*, L293D, Rev. D, Texas Instruments, Jan. 2016. [Online]. Available: https://www.ti.com/lit/ds/symlink/l293.pdf.

[4] E. G. Breijo, *Lecture 3: Counters*, ser. Digital Electronics. Valencia, Spain: Universidad Politécnica de Valencia, 2020.

[5] S. Aubi, *Chapter 6: State Machines*, ser. Physics 351: Digital Electronics Lectures. Williamsburg, VA, USA: College of William and Mary, 2007. [Online]. Available: https://bit.ly/State_FSMs.

[6] Advanced Micro Devices, AMD, *PAL Device Data Book and Design Guide*. Sunnyvale, CA, USA, Jun. 1993, Rev. A.

[7] E. G. Breijo, *Lecture 4: Finite State Machines (FSM)*, ser. Digital Electronics. Valencia, Spain: Universidad Politécnica de Valencia, 2020.

[8] *Series 25l: Hex, gray and quadrature code*, 25LB10-Q, Grayhill. [Online]. Available: http://www.grayhill.com/assets/1/7/Mech_Encoder_25L.pdf.