Lab 1: Building a Combinational Circuit Using Logic Gates

1 Overview

The purpose of this lab is to take a first contact with the lab stand, and learning how to build a combinational circuit.

Each lab in this first term consists of two phases:

1. **Design** of the circuit
2. **Building** and **debugging** this design until it correctly works.

The design phase consists of performing, in paper, the problem design (truth table, switching expressions, etc.). The problem specification is given in the lab script. To perform this design phase we must apply what we have learned about combinational circuits, explained in both practice and theory sessions in this course.

The building phase is performed on the breadboard of the training system located in each lab stand. All the material needed to build the circuit can be found in the lab briefcase. This briefcase must be picked up by the student when he or she is enlisted in laboratories, by November 1.

In this lab, we must design and build a combinational system that performs a conversion from 4-bit Gray code to binary (see Figure 1)

Figure 1. Gray to binary converter

The Gray code (Frank Gray), is a binary numeral system where two successive values differ in only one bit. The reflected binary code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems. Figure 2 shows a 2-bit Gray code.

0	0
0	1
1	1
1	0

Figure 2. 2-bit Gray code

The binary-reflected Gray code list for n bits can be generated recursively from the list for n-1 bits by reflecting the list (i.e. listing the entries in reverse order), concatenating the original list with the reversed list, prefixing the entries in the original list with a binary 0, and then prefixing the entries in the reflected list with a binary 1. For example, generating the n = 3 list from the n = 2 list:

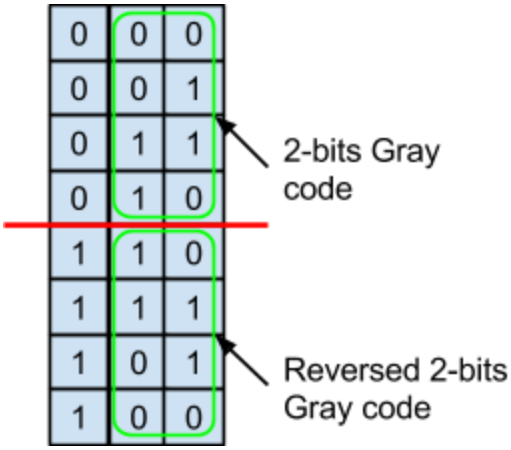


Figure 3. 3-bit Gray code

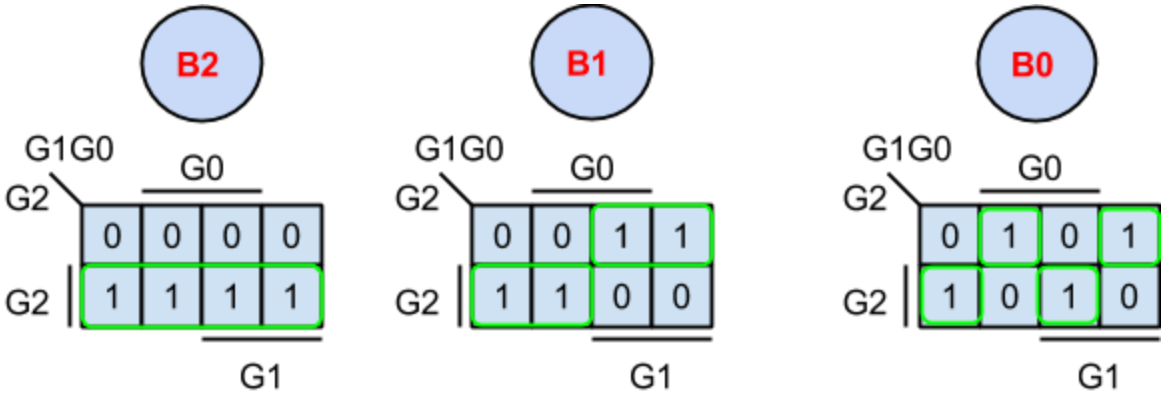
Next, we develop a combinational system: a 3-bit Gray code to binary converter. To this end, we follow the three phases enumerated above.

2 Circuit design

2.1 Truth table

G2	G1	G0	B2	B1	B0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	1	0	1

2.2 Karnaugh maps and simplified expressions



- $B2 = G2$
- $B1 = G2 \oplus G1$
- $B0 = G2 \oplus G1 \oplus G0 = (G2 \oplus G1) \oplus G0$

2.3 Logic implementation

Figure 4 shows the logic implementation of the converter (a graphic representation done through logic components, connection between them, inputs and outputs). In this case, we are using xor gates.

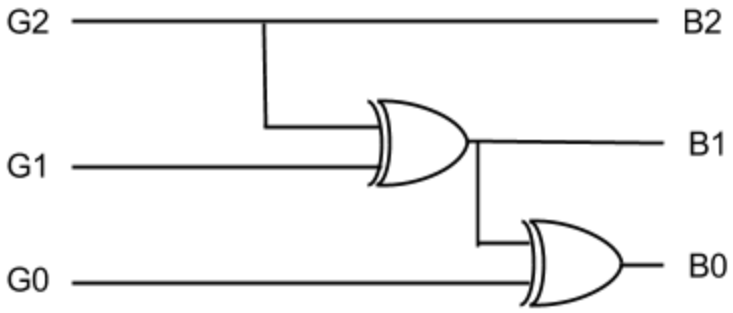


Figure 4. Converter - Logic implementation

In this work, a **4-bit Gray code to binary converter must be designed by the student**: (1) obtaining the truth table, and (2) drawing the corresponding implementation using 2-inputs xor gates. **These two tasks must be done at home and correctly documented in the lab notebook, which must be presented to the assistant professor, at the beginning of the corresponding lab session.**

3 Building and debugging phase

3.1 Training system, breadboard and integrated circuits

To build a circuit we use the training system (see Figure 5), located at the lab stand. This training system is composed by several areas, each one with a specific functionality.

Figure 5. Training system

1. DC Power Supply
    a. Fixed DC output: 5V, 1A
    b. Fixed DC output: -5V, 300mA
    c. Variable DC output: 0V to +15V, 500 mA
    d. Variable DC output: 0V to -15V, 500 mA
    e. GND is the ground output terminal (0 V)
2. 8 bits LED display:
    a. Eight red LEDs separate input terminals. The LED will be lighted up when input is at “HI level”, and it will be turned off when it is at no input or at “LO level”.
3. Eight bits data switches
    a. 8 toggle switches and corresponding output point. When switch is set at “down” position, the output is “LO level”; contrarily, it is to be HI level while setting at “up” position.
4. Center zone
    a. This zone is where the breadboard is placed (see Figure 6). In this breadboard the circuit is build.

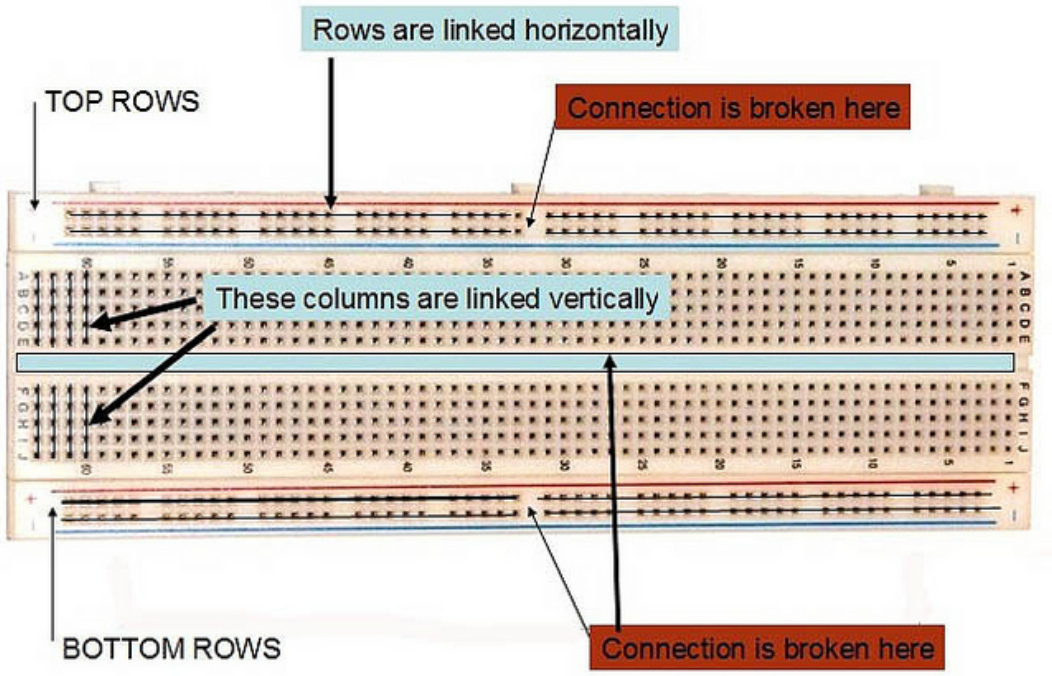
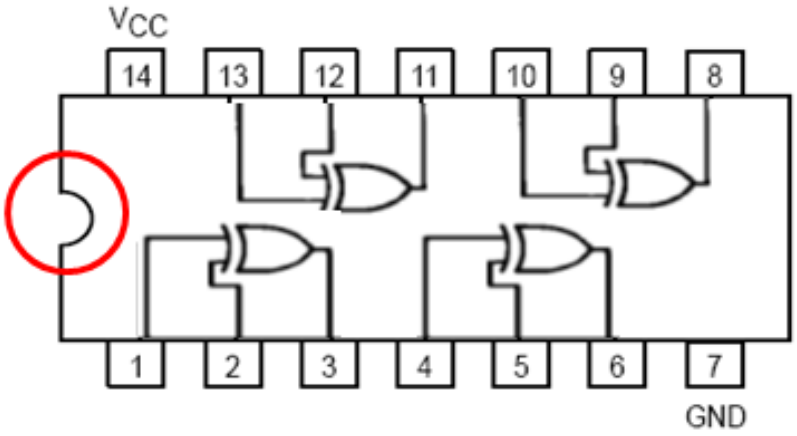


Figure 6. Basic breadboard structure[1]

A breadboard has holes all over it designed to fit the leads of electrical components. These holes are connected in columns by conductive metal strips. We also may find two sets of two rows on top and bottom. These are connected horizontally, usually used for the Fixed DC output and GND contacts of the training system. Figure 6 shows how this pattern is arranged by outlining connected holes using black lines. Not everything is outlined, just enough to show the pattern.

We first need logic gates to build circuits. These gates are encapsulated into an integrated circuit colloquially called “IC” or "chip". These chips are found in the lab kit.

The single "chip" that is required in this first lab is the 7486 containing four 2-input XOR gates (see Figure 7).



[1] http://www.buildcircuit.com/building-a-circuit-on-breadboard/

Figure 7. Content of the 7486 IC

To know how these "chips" work and how they are interconnected we must use the datasheets provided by the manufacturer, which contain a complete specification of the "chip". These datasheets may be found in the lab PC, within the folder named Electrónica. As an example, Figure 7 shows the interconnection of the "chip" 7486, provided in the datasheet of this IC.

As Figure 7 depicts, this IC has four 2-input XOR gates inside. On the outside, we may find 14 pin, each one with its corresponding function:

- Pins 14 and 7 are responsible for providing energy to the circuit connecting VCC to +5V and GND to GND in the training system.
- The remaining pins are inputs and outputs of the four XOR gates as indicated in Figure 7.

As Figure 7 shows, leads are numbered consecutively from Pin 1. When the identifying notch in the package is at the left, Pin 1 is the bottom left corner of the device. Then, the remaining pins are numbered in counter-clockwise order. Sometimes Pin 1 is identified with an indent or paint dot mark.

3.2 Physical implementation

Once we know what chips are needed to implement the circuit, the logical design must be completed. To this end, we must specify one target “chip” for each logic gate (this is mandatory if we have more than one IC), and what pin of the IC will be used for each input or output of the logic gate. We must also define the set of switches and leds we need in the training system. Switches are used as inputs, whereas leds are used to “see” the output. Figure 8 shows the resultant design, which can be called “physical implementation”.

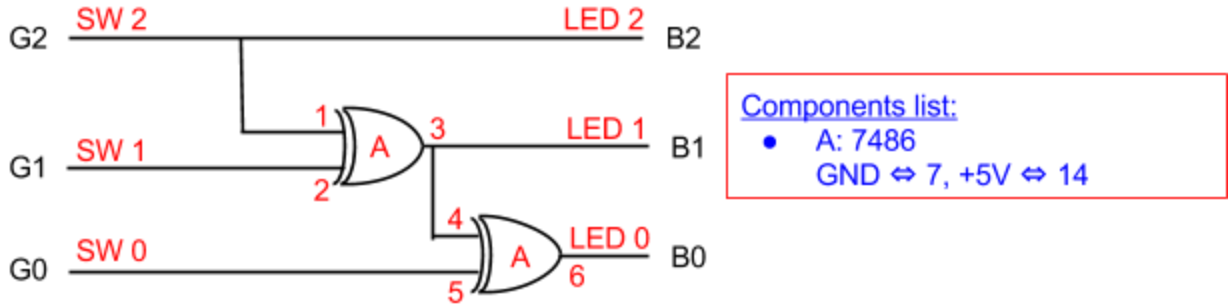


Figure 8. Converter - Physical implementation

The student must do the same, but with the circuit obtained for the 4-bit Gray-to-Binary converter. **Physical implementation must be developed at home and drawn in the lab notebook.**

3.3 Circuit Building

Now the circuit must be built. Converting a circuit diagram to a breadboard layout is not straightforward because the arrangement of components on breadboard looks quite different from the circuit diagram. When putting parts on breadboard we must concentrate on their connections, not their positions on the circuit diagram. Thus, with the training system off, each

needed IC will be pushed straight in the breadboard. Next, wires must be inserted, connecting components, power supply and inputs/outputs to the training system. To this end, the drawing of the physical implementation must be carefully followed. Figure 9 shows an schema of how the 3-bits converter would be mounted in the training system.

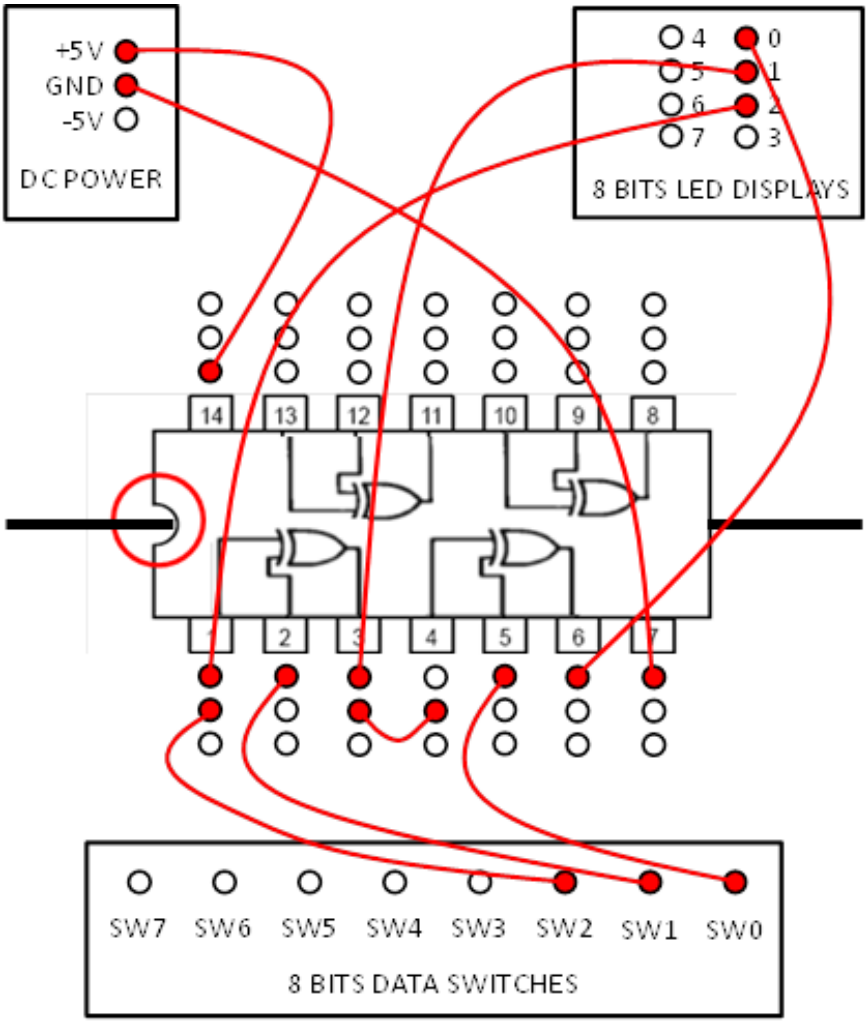


Figure 9. Scheme of the 3-bit Gray-to-Binary converter built in the training system.

3.4 Test

Now the circuit must be checked out. First, we turn the training system on. Next, we use the switches to introduce all the inputs in the truth table. At the same time, we check if the current output (LEDS) and the predicted one (truth table) are the same.

At the end, if the behavior of the circuit has matched exactly the specification given by the truth table, the circuit is correct. Otherwise it is incorrect and must be debugged. Next, we show the steps that must be followed to find errors in combinational circuits.

3.5 Combinational circuit debugging

If for a given input, the output of the real circuit is not the same as the output of the truth table, we must find the cause of the error. To this end, we proceed as follows:

1. We must review the design to see if there is an error in the design phase. Otherwise, we

must go to point number 2.

2. Once we are sure that the error is in the hardware or physical implementation of the circuit, we must first check if all the ICs are correctly polarized. Next, we configure the input in the training system that produces the incorrect output. This wrong output (let's call it *S*) is isolated. To this end, the output is completely disconnected, i.e., we extract all the wires connected to this output. Two things may occur:
    a. *S* changes giving the correct value. Thus, this output is connected to other output, which is "forcing" the incorrect value ⇒ go to point 3.
    b. *S* is still giving an incorrect value. Thus, there is an IC that produces an incorrect output ⇒ go to point 4.
3. *S* is being "forced" by the output of another IC, is grounded or connected to +5V. To locate the error, we may follow all the wires connected to S, or we may isolate *S* disconnecting all the wires to *S*, and then reconnecting them to *S* one by one (when *S* gives an incorrect value, we have identified the wrong connection).
4. An IC (named *M*) is giving an output that is wrong with respect to our truth table, but is correct with respect to the *M* inputs. Thus, one of the inputs must be incorrect. Since each input is the output of another IC (named *M'*), we must repeat from point 2 but using *M'*. On the other hand, if the output of M are incorrect with respect to its inputs, then the IC is not correctly polarized or it is broken.

Figure 10 depicts an organigram of this debugging process.

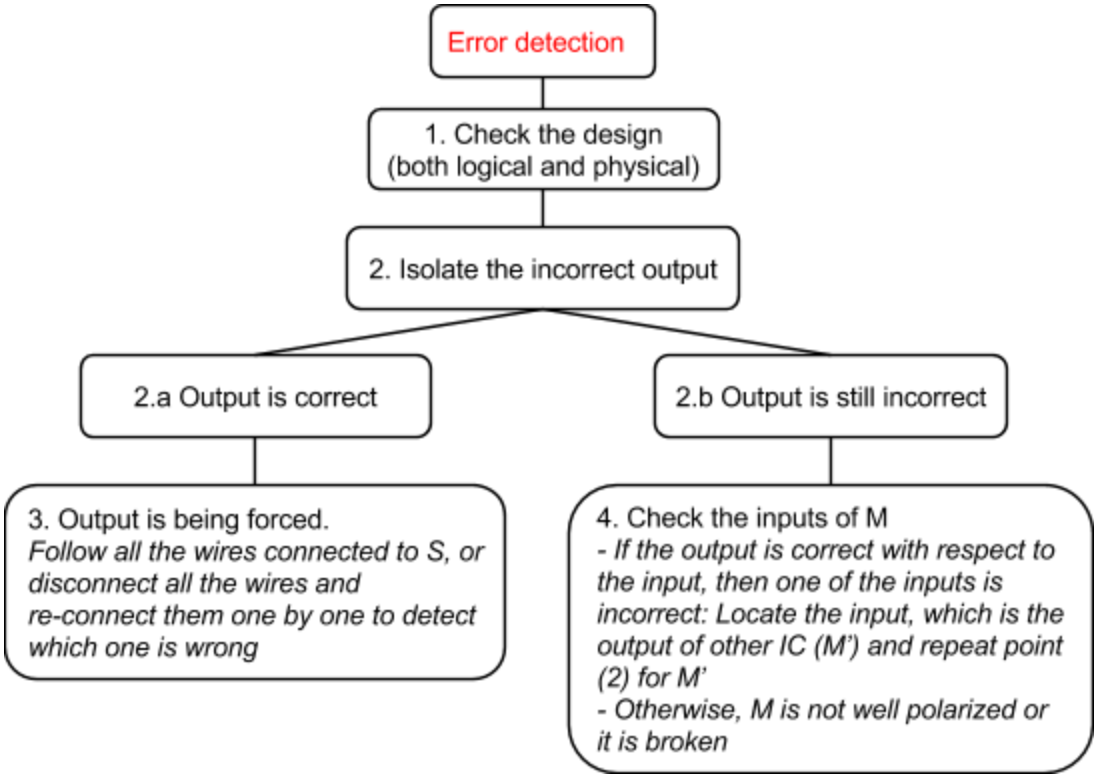


Figure 10. Debugging organigram