CMPE 12 Lab Report # 1

John Allard
TUTOR
Lab Section #2

October 6th, 2014

Contents

1 Overview

This lab served as an introduction to circuit design from truth tables using logic gates. All circuits were implemented using the free MultiMedia Logic (MML) program, which allows the user to drag, drop, and connect different circuit components. After the circuit is connected it can be simulated to help check for design errors. This lab consisted of 4 section, with each section varying in difficulty and length.

2 Part A : Using MultiMedia Logic

2.1 Procedure

This section was a combined introduction to MML and DeMorgan's laws. To start we were instructed to navigate to an MML tutorial on YouTube[1] and build the circuit that is shown in the tutorial. After this, we were instructed to show our understanding of MML by implementing DeMorgans laws $A'B' = (A + B)'$. To do this, we needed to build a circuit for each side of the DeMorgans equation and show their equivilence both in practice and by truth table.

2.2 Results

2.2.1 Results - Tutorial

To start, we completed an extremely simple circuit using the MML software. This circuit is about as simple as one can get, it is just a binary switch connected to an LED. In this case, if the switch is high, the LED is on, if the switch is low, the LED is off. Next we simply inserted an NOT between the switch and the LED, causing it to only light up when the switch is in a low state. The two circuits for the tutorial section of part A can be seen in figure one.

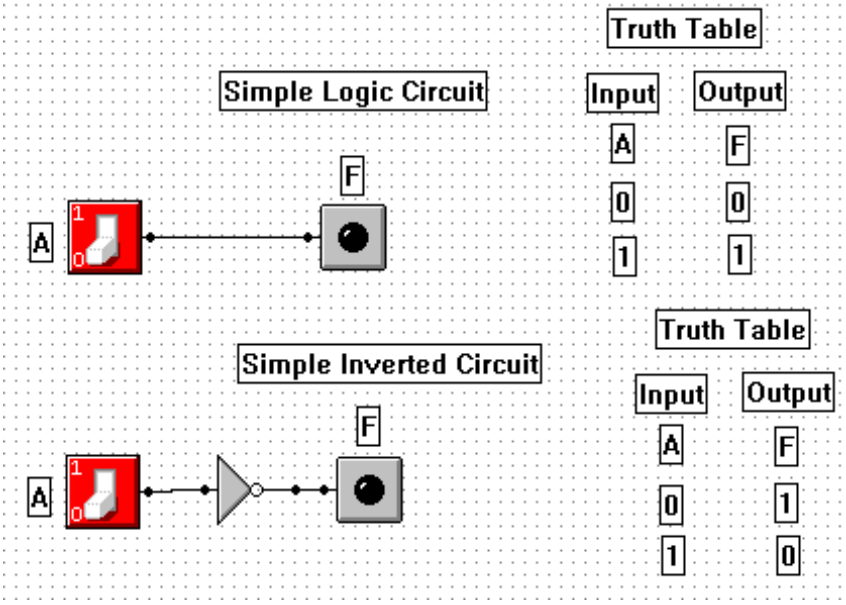

Figure 1: Circuits for Part A Tutorial.

2.2.2 Results - DeMorgans Law

Our next goal was to create two new circuits which demonstrate DeMorgans law, specifically the fact that $A'B' = (A + B)'$. This involved the use of two switches, labled A and B. These switches will be used as the inputs for two circuits, the first one representing $A'B'$ and the second one representing $(A + B)'$. Once completed, a simulation of these circuits can be run and it can be seen that the LED for each of the two circuits are in an identical state for the entirety of the message space, which happens to only be $\{00, 01, 10, 11\}$ for this simple example. The first circuit, $A'B'$, is created by passing the A and

[1] `http://www.youtube.com/watch?v=hJq2gECXYWc&noredirect=1`

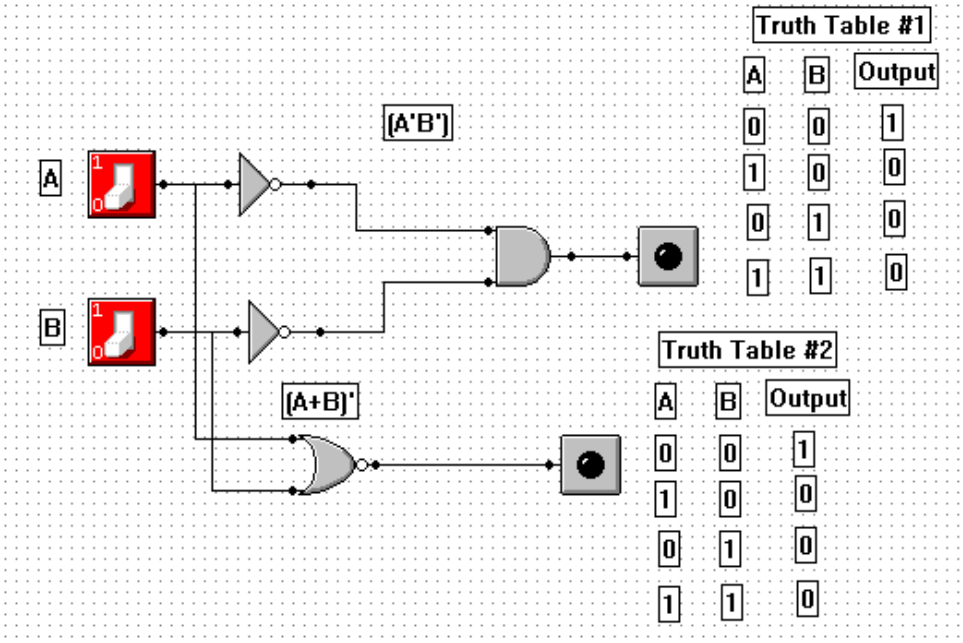

Figure 2: Circuits for Part A2, Showing DeMorgans Law.

B inputs through NOTs before combining them into an AND gate. The output of the AND gate is run directly to an LED. The second circuit is made by running the A and B inputs directly into a NOR gate, then to an LED. The circuit and truth tables can be seen in Figure 2.

3 Part B : Implementing Functions as Sums of Products

3.1 Procedure

IN[2]	IN[1]	IN[0]	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 3: Truth Table for Parts B and C

This part of the lab consisted of two sections, B1 and B2. The goal of both sections was the same, implement the truth table in Figure 3, the differences between the two implementations are the restrictions on the gates that can be used.

In the circuits that we are building, the following names will be used for the inputs listed in Figure 3 :

$$\text{IN}[2] \rightarrow A,\ \text{IN}[1] \rightarrow B,\ \text{IN}[0] \rightarrow C$$

The output of the circuit will be a single LED with a basic 1 = HIGH, 0 = LOW relationship. Part B1 asked us to construct a circuit to represent the truth table in Figure 3 using AND, OR, or NOT gates. Part B2 asked us to reimplement the same circuit using only NAND gates. This requires translation of the AND, OR, and Inversion gates into nested layers of NAND gates. Finally, we were supposed to run the circuits in both sections side by side to ensure their equivalence.

3.2 Results

Before any circuitry could be properly constructed, the truth table had to be broken down into a boolean equation using the sum of products rule. Working from left to right in the truth table

in Figure 3, with IN[2] $\rightarrow A$, IN[1] $\rightarrow B$, IN[0] $\rightarrow C$, we arrive at the following equation.

$$\text{A'BC'} + \text{A'BC} + \text{AB'C} + \text{ABC'} \tag{1}$$

3.2.1 Results - AND, OR, & NOT Gates

Part B1 involved directly translating (1) into a circuit using MML. Because we were allowed to use AND, OR, and NOT gates, the circuit reads all most exactly like (1). The circuit for part B1 is shown in figure 4.

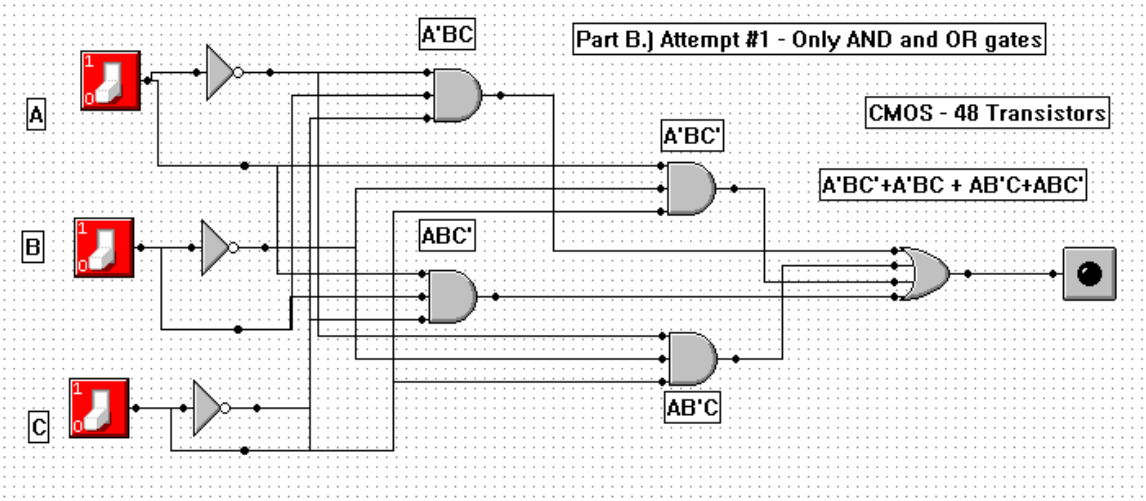

Figure 4: Circuit for Part B1

The circuit is best read from left to right. It starts off with the three inputs, A, B, and C. These three switches can be toggled to create a total of $2^3 = 8$ possible combinations. Right of these switches is a layer of four 3-input AND gates. These AND gates represent the individual products in (1). The output from each of these four AND gates are routed to a single 4-input OR gate. This gate represents the sum of the individual products shown in (1). The output of this OR gate is connected to a single LED to display the circuit output to the user. As is shown in figure 4, this circuit consists of a total of 48 transistors using CMOS transistor logic. How I found this number is documented below.

$$\#\text{ of* Transistors per Gate}$$

$$\text{NOT} = 2,\ \text{AND} = 2n_1 + 2,\ \text{OR} = 2n_2 + 2$$

Where n denotes the number of inputs for that specific gate. With 3 NOT gates, four 3-input AND gates, and one 4-input OR gate, the linear combination of transistors for this circuit becomes ...

$$3 * (\text{NOT}) + 4 * (\text{AND}) + 1 * (\text{OR}) =$$

$$3 * 2 + 4 * (2 * n_1 + 2) + 1 * (2 * n_2 + 2) = 6 + 32 + 10 = 48$$

Of course in the real world we would not use 48 transistors for this circuit, (1) can be simplified significantly, as we will see in Part C.

3.2.2 Results - NAND Gates

Next we were instructed to take (1), implement it in a circuit using MML, but this time we were only permitted to use NAND gates. I started with the circuit in the previous section (Figure 4), and used the following substitutions to achieve the same output with only NAND gates.

$$\text{NOT}(A) \rightarrow \text{NAND}(A,A)$$

$$\text{AND}(A,B) \rightarrow \text{NAND}(x,\, x), \text{ where } x = \text{NAND}(A,B)$$

$$\text{OR}(A,B) \rightarrow \text{NAND}(x_A,\, x_B) \text{ where } x_n = \text{NAND}(n,n)$$

We simulate an inverter by routing a single signal to both inputs of a NAND gate. If the signal is high, both inputs to the NAND are high, which outputs a 0. If the signal is low, both inputs are low and thus the NAND outputs a 1. I simulate an AND gate by first routing two inputs through a single NAND gate. I then route this output to both inputs of another NAND gate. This effectively creates a 'double negative' effect, canceling out the 'N' in NAND. Finally, I simulate an OR gate by routing each of the inputs through a NOT gate (which is just a NAND gate with one signal tied to both inputs), then finally routing the output from each NOT gate through a single NAND gate. This works because NAND only returns true if any input is 0. Because we inverted every input before the final NAND gate, this means that our NAND gate will only return true if any of the original signals were equal to 1, which is exactly the definition of an OR gate.

After I figured how to how to convert each individual component from the first circuit to their NAND gate equivalent, all that was left to do was to tie all of the NAND gates together and hope for the best. My final circuit is shown in Figure 5.

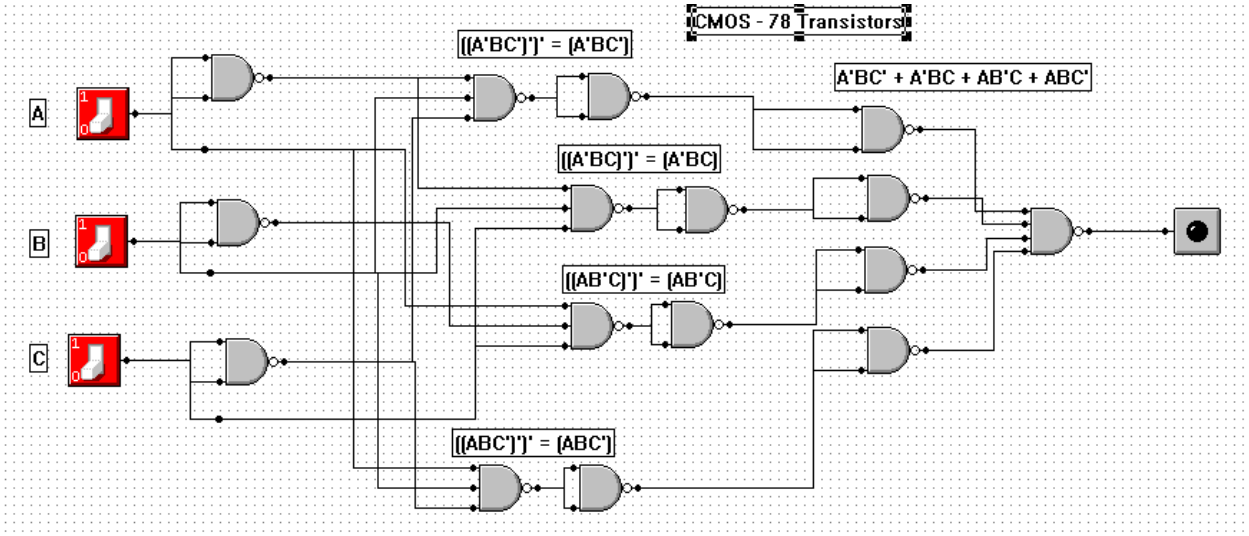

Figure 5: Circuit for Part B2, using only NAND gates.

As you can see it consists only of NAND gates, yet structurely resembles the circuit from Figure 4. It still has a layer of 3 NOT gates, followed by four 3-input AND gates, and lastly all 4 signals are tied together through an OR gate, the only difference was this time everything has been implemented using only NAND gates. The transistor count for this circuit was 78. I used the following math to find this number.

$$\#\text{ of Transistors per Gate}$$

$$\text{NAND} = 2n$$

Where n denotes the number of inputs for the NAND gate. With eleven 2-input NAND gates, four 3-input NAND gates, and one 4-input NAND gate, the linear combination of transistors for this circuit is ...

$$11 * (\text{NOT}) + 4 * (\text{AND}) + 1 * (\text{OR}) =$$

$$11 * (2 * 2) + 4 * (2 * 3) + 1 * (2 * 4) = 44 + 24 + 10 = 78$$

You can simulate these two circuits by running the `lab1B.lgi` file included with this paper.

4 Part C : Logic Minimization

4.1 Procedure

The main goal of this section is to use the rules of Boolean algebra to simplify (1) down to a more managable equation, then implement it again as a more simple circuit. For reference, here again is the equation we are simplifying.

$$\text{A'BC'} + \text{A'BC} + \text{AB'C} + \text{ABC'}$$

4.2 Results

4.2.1 Simplification

I started by reducing down (1) using Boolean algebra. My steps are listed below.

$$\text{A'BC'} + \text{A'BC} + \text{AB'C} + \text{ABC'} \qquad \text{(original)}$$

$$\text{A'B(C'} + \text{C)} + \text{A(B'C} + \text{BC')} \qquad \text{(factoring)}$$

$$\text{A'B(1)} + \text{A(B'C} + \text{BC')} \qquad ((X + X') \text{ is a tautology})$$

$$\text{A'B(1)} + A(B \oplus C) \qquad ((X'Y + XY') = \text{XOR} = \oplus)$$

$$\text{A'B} + A(B \oplus C) \tag{2}$$

The final simplified equation that I ended up with is (2). Beyond this point simplification is no longer possible, but the equation has still been reduced from its original form by quite a bit.

4.2.2 Building the Circuit

Now we were instructed to build a circuit corresponding to (2), which is equivilent to (1), which is derived from the truth table in Figure 3. So (2) is just another way of representing the original truth table (Figure 3). This can be seen in Figure 6. If you visually compare Figures 5 and 6,

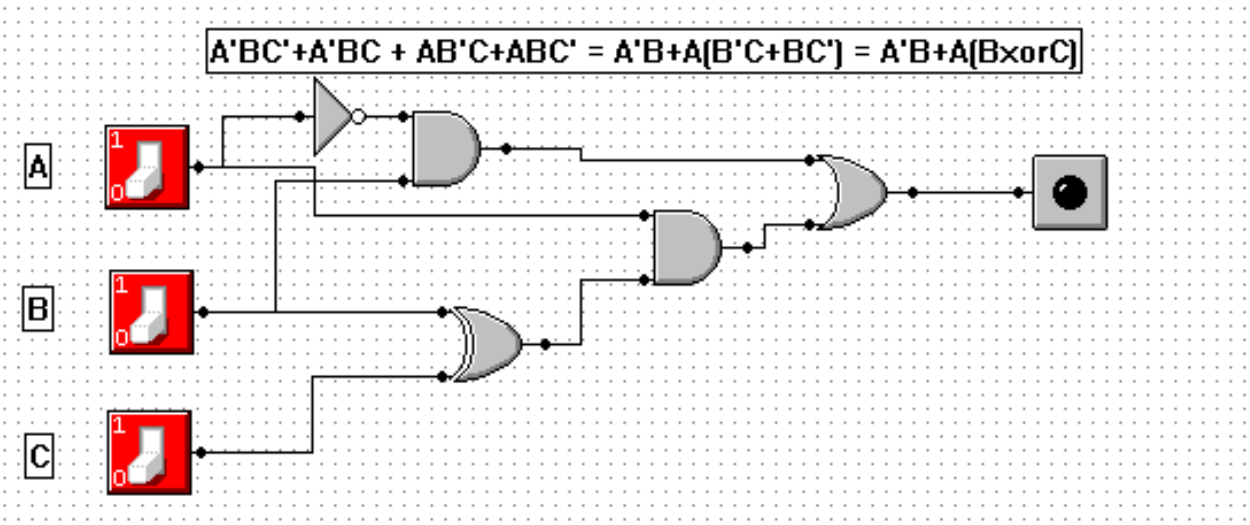

Figure 6: Part C, Simplified Circuit.

you can see how much simpler the latter is.This circuit performed identically when tested side by side with both circuits from Part B.

5 Part D : Guessing Game

5.1 Procedure

This section instructed us to design a 2-bit guessing game. The game works as follows :

- User can set the value of two bits to either 0 or 1, giving them 4 combinations to make.
- Once the user's guess is set, they press a push button to start the game.
- The button causes the a circuit component to generate a 2-bit random number.
- This random number is compared to the users guess for equality.
- If they are equal, an LED goes HIGH. If they are not, the LED stays low.

This involved comparing the low bits for equality, then comparing the high bits for equality, then finally checking that both of those comparisons returned true. If everything works right, the user should have a $\frac{1}{4}$ chance of winning.

5.2 Results

The trickiest problem in completing this part of the lab was figuring out how to compare two bits for equality. I know that AND returns true if they are both 1, which is partly correct. But AND returns 0 if they are both 0, which is not true. Then I realized that XOR is close to what we want, it only returns true if the two inputs are different, and returns false if they are the same. This is exactly the opposite of the desired functionality, so I simply decided to add a NOT gate at the end of an XOR gate to create an XNOR gate

$$\text{XNOR} = (A \oplus B)'$$

So to start, we route the high bit from the user and the high bit from the random number to an XNOR gate, which returns 1 if they are equal and 0 otherwise. We then do the same thing with the low bits from the user's guess and random number. Finally, we route the output from both of these XNOR gates to a final AND gate, which will only output true if both inputs are true. The output from this AND gate is routed to a simple LED to let the user know if they have won or not.

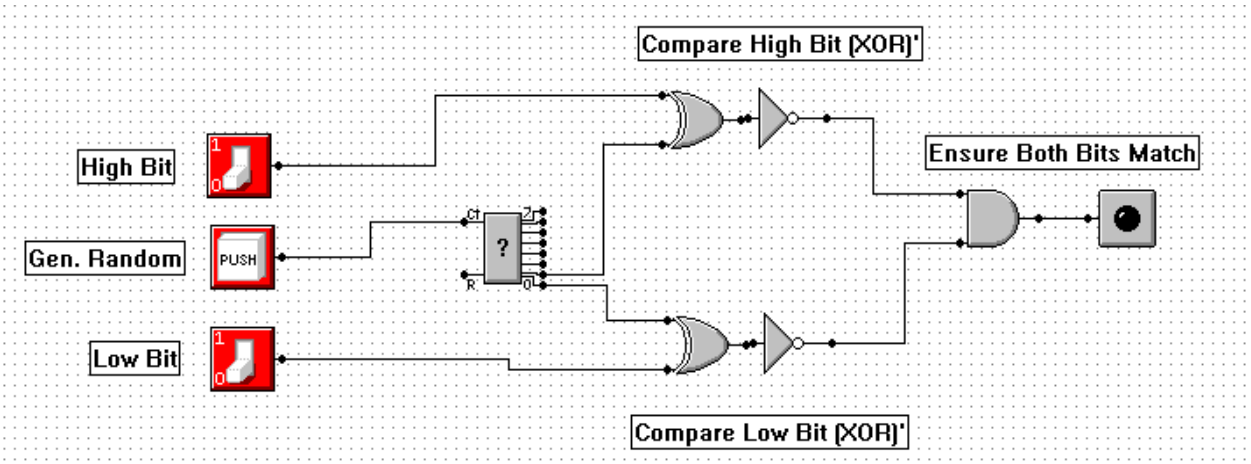

Figure 7: Part D Guessing Game Circuit

6 Conclusion

This lab served as an introduction to building virtual circuits using the MultiMedia Logic program. In learning to use this program, we also learned to use truth tables to design circuits. After this we saw how the tools of Boolean algebra can allow us to significantly simplify the number of components in a given circuit by removing redundancies. Finally, we learned how to compare bits and use a random number generator in our circuit design.