EC312 Lesson 2: Computational Logic

Objectives:

a) Identify the logic circuit gates and reproduce the truth tables for NOT, AND, NAND, OR, and NOR gates.
b) Given a schematic of a logic circuit, determine the corresponding Boolean expression for the circuit output.
c) Identify the functional computing components, such as an adder, that are made of logic gates.
d) Define the term “cyber-physical system (CPS)” and provide two examples.
e) Demonstrate the ability to write simple C programs that perform keyboard input, screen output and simple arithmetic.*
f) Discuss the role of the operating system in bridging the gap between hardware and user applications and services.*
g) Demonstrate the ability to create, edit, compile and execute C programs in a Linux environment.*
h) Demonstrate the ability to analyze C programs that employ if-else statements and for loops.*

[*Note: objectives (e)-(h) are review objectives that cover material which students have already learned in SI110 and ES200/202. Assigned questions in the first two homework sets will assist students in reviewing this material.]

I. Computational Logic and the Five Basic Logic Gates

Computers work with the binary number system; i.e. everything comes down to 1’s and 0’s. The two digits of the binary number system can be represented by the state or condition of electrical or electronic devices. For example, a binary 1 can be represented by a switch that is closed, a lamp that is lit, or a transistor that is conducting, and a binary 0 would be represented by the same devices in the opposite state: the switch open, the lamp off, or the transistor in cut-off.

Of course, a computer doesn’t just store data; it manipulates the data and produces desired outcome based on certain programmed logic statements. (Think about the if-else and for statements in C.) Another way to say this is that computers use the TRUE and FALSE logic conditions of a logical statement to make a programmed decision. Kind of convenient, since TRUE and FALSE can be represented as 1 and 0! But here’s the main question: how do you build a circuit (after all, a CPU is just a really, really complex electrical circuit) that can evaluate logical statements? That’s what we want to explore.

1. Digital concepts of logic The conditions of a statement can be represented by symbols (variables); for instance, the statement “Navy beat Army this year” might be represented by the symbol P. If Navy actually beat Army, then P is TRUE. If Navy did not beat Army, then P is FALSE. As you can see, a statement has two conditions. In computers, these two conditions are represented by electronic circuits operating in two LOGIC STATES. These logic states are 0 (zero) and 1 (one). Respectively, 0 and 1 represent the FALSE and TRUE conditions of a statement.

When the TRUE and FALSE conditions are converted to electrical signals, they are referred to as LOGIC LEVELS called HIGH and LOW. The 1 state might be represented by the presence of an electrical signal (HIGH), while the 0 state might be represented by the absence of an electrical signal (LOW).

If the statement “Navy beat Army this year” is FALSE, then the statement “Navy did NOT beat Army this year” must be TRUE. This is called the COMPLEMENT of the original statement. In the case of computer

---

1 Portions of these lesson notes were adapted from the Navy Electricity and Electronics Training Series (NEETS) course notes, Module 13, September 1998.
math, complement is defined as the opposite or negative form of the original statement or variable. The complement is shown by placing a bar over the statement symbol (in this case, \( \bar{P} \)). This variable is spoken as NOT P. The following table illustrates this concept and the relationship with logic states and logic levels.

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>SYMBOL</th>
<th>CONDITION</th>
<th>LOGIC STATE</th>
<th>LOGIC LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>( P )</td>
<td>TRUE</td>
<td>1</td>
<td>HIGH</td>
</tr>
<tr>
<td>NAVY BEAT ARMY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complement:</td>
<td>( \bar{P} )</td>
<td>FALSE</td>
<td>0</td>
<td>LOW</td>
</tr>
<tr>
<td>NAVY DID NOT BEAT ARMY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In some cases, more than one variable is used in a single expression. For example, the expression \( AB\bar{C}D \) is spoken "A AND B AND NOT C AND D."

2. **The fundamental logic gates** As you study logic circuits, you will see a variety of symbols (variables) used to represent the inputs and outputs. The purpose of these symbols is to let you know what inputs are required for the desired output. A logic function produces the correct output based on the given inputs. In this section we’ll study the fundamental logic functions that form the basis for all computation: AND, OR, NOT, NAND, and NOR. We’ll also study the electronic devices used to implement these logic functions, which are known as logic gates (or, more informally, gates). These gates are the foundation for all digital equipment, and they make up the "decision-making" circuits of computers and other types of digital equipment.

**a. The AND gate** Let’s hearken back to EE331 (yes, a long long time ago, back to last semester…) and consider the circuit in Figure 1. If we think of the state of switch A and switch B as our device inputs (0=open, 1=closed) and the state of the LED as our device output, we can apply our finely tuned analysis tools to determine that the LED will only turn on if both A and B are closed. All other cases will result in the LED remaining off. The corresponding logic gate is known as the AND gate.

Letting \( X \) denote the state of the LED (0=off, 1=on), we can summarize our results in the truth table shown below on the left. A truth table is just a table that depicts the output for every possible combination of inputs. The logic symbol for an AND gate is also shown below, as well as the associated Boolean expression (which is just a fancy name for expressions that result in TRUE/FALSE outputs).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 1: AND Circuit**
Note that an AND gate can have more than two inputs. In this situation, all combinations result in a 0 output except for the case where every input is 1 (i.e. High). (I.e., an AND gate requires a “unanimous vote”.)

b. The OR gate Now let’s consider the circuit shown in Figure 2, again viewing our two switches as inputs and the LED as the output. In this case, the LED will light (i.e. X=1) if either the first switch or the second switch is closed (or if both are closed). In fact, the only case in which the LED won’t light is if both switches are open. The corresponding logic gate is known as the OR gate. The truth table, logic symbol, and Boolean expression for the OR gate are shown below.

![Figure 2: OR Circuit](image)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
X = A + B
\]

c. The NOT gate A NOT gate (also known as an INVERTER), is a logic device that has an output opposite of the input. (We encountered this gate already, at the end of the last lesson.) It may be used alone or in conjunction with other gates. The truth table, logic symbol, and Boolean expression for the NOT gate are shown below.

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
X = \overline{A}
\]

d. The NAND gate The next gate is simply the combination of the NOT gate introduced in Lesson 1 with the AND gate. This is known as a NAND gate, which is the exact opposite of the AND gate, i.e., the output is always 1 (i.e. High) except for the case where both inputs are high. The truth table, logic symbol, and Boolean expression for the NAND gate are shown below. (Note that the logic symbol for the NAND gate looks exactly like the AND symbol, except for the small circle at the output. The small circle denotes a NOT gate, i.e. the NAND is equivalent to an AND with a NOT.)
The NOR gate

By now you’re probably getting the picture of how this works. The NOR gate, which is equivalent to an OR gate connected in series with a NOT gate. The truth table, logic symbol, and Boolean expression for the NOR gate are shown below.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ X = \overline{A+B} \]

Practice Problem

Fill in the truth table for the logic circuit shown below.

\[ \begin{array}{c|c|c}
   A & B & X \\
   \hline
   0 & 0 & 1 \\
   0 & 1 & 0 \\
   1 & 0 & 0 \\
   1 & 1 & 0 \\
\end{array} \]

3. Logic gates in combination

Most logic circuit diagrams are made up of many combinations of gates, with the outputs of some gates serving as inputs for others. At first it may seem confusing and complex, but if you interpret one gate at a time, you can work your way through any network.

Typically, our goal when analyzing a logic circuit is to determine the Boolean expression of the output and its corresponding truth table. This is best taught by example, so let’s consider the logic circuit shown in Figure 3.
Here the outputs of the AND gate and the NOT gate become the inputs for the OR gate, so that the final output X will go High if either C is Low or both A AND B are High. The best way to tackle analyzing such a circuit is to take it one gate at a time from left to right, writing the Boolean expression at each output and working our way to the final output X. In this case, we know that the output of the AND gate is AB, and the output of the NOT gate is Ĉ, and therefore the final output X is given by \( X = AB + Ĉ \), as shown in Figure 4.

\[
\begin{align*}
A & \quad B & \quad C & \quad AB & \quad Ĉ & \quad X \\
0 & \quad 0 & \quad 0 & \quad 0 & \quad 1 & \quad 1 \\
0 & \quad 0 & \quad 1 & \quad 0 & \quad 0 & \quad 0 \\
0 & \quad 1 & \quad 0 & \quad 0 & \quad 1 & \quad 1 \\
0 & \quad 1 & \quad 1 & \quad 0 & \quad 0 & \quad 0 \\
1 & \quad 0 & \quad 0 & \quad 0 & \quad 1 & \quad 1 \\
1 & \quad 0 & \quad 1 & \quad 0 & \quad 0 & \quad 0 \\
1 & \quad 1 & \quad 0 & \quad 1 & \quad 1 & \quad 1 \\
1 & \quad 1 & \quad 1 & \quad 1 & \quad 0 & \quad 1
\end{align*}
\]

How did we know how many rows to put in our truth table? If we have a total of \( N \) inputs to the system (in this case \( N=3 \) because we have inputs A, B, C), then we need \( 2^N \) rows. You’ll note that we filled in the inputs for those rows by simply counting up from 000 to 111 in binary. Then we can determine the quantity AB by applying the rules of our AND truth table, we get Ĉ by inverting C, and we get the output X by applying our OR truth table rules to the values in the AB column and the Ĉ column.
Practice Problem

For the logic circuit given below, determine the Boolean expression for the output X, and construct the corresponding truth table.

![Logic Circuit Diagram]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that there are other (easier) ways to simplify Boolean expressions based on certain laws and identities, but that’s beyond the scope of this course. However, if you’re interested, you should search online for “Simplifying Boolean Expressions” and see the amazing methods that have been developed!

4. Computing Components

Combinational logic can produce many useful circuits. We will examine a basic adder circuit (half adder). A half adder takes two single digit binary numbers and adds them together. See the circuit in Figure 5 where A and B are the inputs and S is sum and C is the carryover.

![Half Adder Circuit Diagram]

Figure 5: Half Adder Circuit

Practice Problem

For the half adder circuit shown in Figure 5, determine the Boolean expression for the output C and S.

\[
C = \]

\[
S = \]
Practice Problem

For the half adder circuit shown in Figure 5, determine the construct the truth table to show that the half adder works properly.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Cout</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The half adder can be used to add two one bit numbers but when adding multiple bit numbers, you need to be able to carry in a 1 from a previous addition. This is effectively adding three one bit numbers. For example, adding 101 and 11 results in:

\[
\begin{array}{c}
111 \\
101 \\
+ 011 \\
\hline
1000
\end{array}
\]

In this example there are carry ins in the second and third bit additions. To be a fully functional adder, we need to be able to handle a carry in the previous digit. The circuit below adds this feature.

Figure 6: Full Adder Circuit
Practice Problem

For the full adder circuit shown in Figure 6, determine the Boolean expression for the output Cout and S.

\[\text{Cout} = \]  
\[\text{S} = \]

Combining many of these circuits you can make a 4-bit adder:

Similarly, circuits can be made to subtract, multiply, divide, etc. In a modern computer these devices are contained within an arithmetic logic unit (ALU) and are used as the building blocks of a computer, which we will be working on programming in the following lessons.

II. Cyber-physical Systems (CPS)

When you think of computing, you probably think of your issued laptop (which we make you lug to EC312 class) or a desktop computer in one of your classrooms. Most of this course will focus on those types of systems, but it’s important that we consider computing (and its vulnerabilities to cyber attack) in other contexts. For instance, consider your car. (Yes, you wish you could park it on the Yard, but at least you can have one!) Did you know that a typical modern automobile may have up to 100 million lines of computer code?2 As Bruce Emaus, the chairman of SAE International’s embedded software standards committee, puts it in the previously cited article, “It would be easy to say the modern car is a computer on wheels, but it’s more like 30 or more computers on wheels.” Computers and physical systems are increasingly being connected together.

The National Science Foundation describes Cyber-physical Systems (CPS) as systems which “integrate sensing, computation, control and networking into physical objects and infrastructure, connecting them to the Internet and to each other” (http://www.nsf.gov/news/special_reports/cyber-physical/). In other words, this is the tight intertwining of “cyber” (i.e. software, networks, digital data) with the “physical” world (e.g. motors, sensors, actuators). Sounds like something that Systems Engineers would be interested in! Examples include bionic limbs, automated damage control systems, self-driving vehicles, etc. Check out http://cyberphysicalsystems.org/ for a big picture “Concept Map” of CPS and their various applications and research areas.

--