Chapter 1: Digital logic

I. Overview

In PHYS 252, you learned the essentials of circuit analysis, including the concepts of impedance, amplification, feedback and frequency analysis. Most of the circuits we used were *linear circuits*, where the output of a circuit component was proportional to the input. For example, a good amplifier might make an input voltage level larger but it should not change the shape of the signal. The best amplifiers (which we did not attempt to build) only amplify the input signal and do not add any noise or distortion to the signal. They also have a large dynamic range, which means that they can amplify small signals or large signals while maintaining a linear proportionality between the output and the input. This is difficult and requires careful attention to the circuit layout. In most cases researchers will simply buy well-engineered commercial amplifiers rather than build their own.

Digital circuits are at the other end of the spectrum from linear circuits. A digital output has only two possible values. Rather than denoting these two output voltages by their actual values (e.g. 0V or 5V), the output voltages, or states, are denoted by a number of conventions:

- High or low

- H or L

- True or False

- T or F

- 0 or 1

Digital circuits are far less sensitive to the circuit components and the component layout than analog designs. They primarily depend on logic, rather than currents or voltages for operation and so one can quickly learn to build rather sophisticated digital circuits. However, the increasing speed of microprocessors has also made digital circuit design less necessary. In this course we will survey some of the most important concepts in digital circuit design: logic gates, timing, A/D and D/A converters, memory, bus design and digital signal processing. As the course progresses, we will move from building simple digital circuits to using computer programming to create sophisticated and flexible interfaces between the digital and analog world.

II. Digital Logic and Boolean Algebra

Variables

Some variables have only a single binary digit, and therefore can take only two possible values, TRUE (1) or FALSE (0). We will call these logical (or Boolean) variables.

This type of variable is extremely useful for controlling equipment and computer programs. For example, if you want a variable that represents whether a piece of equipment is turned on (1) or turned off (0), you might assign a logical variable to it. Similarly, if a shutter is open (1) or closed (0), you might want to assign a logical variable to it. Now, you could use Boolean algebra to make sure that you only try to take data when your equipment was turned on and your shutter was open.

Operators

Since there are only two possible states for a logical variable, there are only two possible logical unary (single operand) operators. The **identity** operator does not change the logical value, so

0 input →0 output

1 input →1 output.

The *inverter* reverses the logical input, so

0 input →1 output

1 input →0 output.

This second operator is usually called *NOT*, and is represented by a bar over a variable, so that *NOT*(A) is written as $\overline{A}$. Note that using an inverter twice produces the identity operator again, so

$$\overline{\overline{A}} = A.$$

The electronic symbol for a device that performs the NOT operations (called a gate) is just an amplifier (a triangle pointing to the output) with a small circle to denote inversion on the output. Figure 1-1 shows the truth table and schematic diagram for a NOT gate. A truth table shows all possible input values with their respective output values. When you are designing logical circuits, you will find that a truth table is usually the best

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Figure 1-1: NOT gate truth table and diagram.

way to summarize your logic.

Of course, truth tables and Boolean algebra are pretty trivial for the unary operators. They become useful when your operators accept more inputs.

First, let's look at two important Boolean operators that take two inputs, AND and OR. AND gives true if *both* inputs are true, while OR gives true if *either* input is true. AND is written as logical multiplication

$$A \ AND \ B \equiv A \times B \equiv AB$$

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
H	H	L
L	X	H
X	L	H

Figure 1-2: NAND gate truth table and diagram.

since multiplying anything by 0 results in 0. Logically, this means that if even one input is false (0), the output of an AND gate is also false (0). The OR gate is written as a logical addition

$$A \ OR \ B \equiv A + B.$$

This makes sense logically if you think that this produces a false (0) only if both inputs are false (0). Note that for both AND and OR, the easy interpretation results from considering what makes it false.

If you invert the output of these gates, you have NAND and NOR gates, which are the fundamental building blocks of digital circuits. For a NAND gate, a 0 on any input produces a 1 on the output. For a NOR gate, all inputs must be 0 to produce a 1 on output. By "fundamental," we mean that *any* logic combination of *any* number of inputs can be constructed from combinations of two-input NOR gates or two-input NAND gates.

A NAND gate is usually pictured as an AND gate with a little circle on the end (just like the NOT gate) to signify inversion. When you draw an AND or a NAND gate, it

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
H	X	L
X	H	L
L	L	H

Figure 1-3: NOR gate truth table and diagram.

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

Figure 1-4: XOR gate truth table and diagram.

is crucial to make the gate face perpendicular to the input lines. Figure 1-2 shows the

truth table and diagram for a two-input NAND gate. A NOR gate is pictured as an OR gate with a little circle on the end to signify inversion. The OR gate looks similar to the AND gate, but its inputs enter into a curved face. Figure 1-3 shows the truth table and diagram for a two input NOR gate.

In addition to the OR and AND gates there is a final common two input gate called an exclusive OR (or XOR). This gate gives a true if either of the inputs are true but not it both inputs are true. It is drawn as an OR gate with an extra curved line on the input lines. Figure 1-4 shows the truth table and diagram for a two-input NAND gate.

Operator Synthesis

You can build any other gate from them the two-input NOR and the two-input NAND gates. For example, if you want to build an inverter, you simply tie the two inputs together on either a two-input NOR or a two-input NAND, since they both then have the same truth table:

A	A	$\overline{A+A}$	$\overline{AA}$
0	0	1	1
1	1	0	0

If you want to use more than two inputs, you can always build them from AND or OR gates since both multiplication and addition are associative. These properties are shown in both ways of writing the formulas as:

$$A \text{ AND } B \text{ AND } C = A \text{ AND } (B \text{ AND } C)$$
$$ABC = A(BC)$$
$$A \text{ OR } B \text{ OR } C = A \text{ OR } (B \text{ OR } C))$$
$$A + B + C = A + (B + C)$$

However, if you want to use our basic NAND and NOR gates, then we must first note that inversion distributes itself in a funny way. Inverting the output of a gate is equivalent to inverting both inputs and simultaneously changing the gate from an AND to an OR or from an OR to an AND.

$$\overline{AB} = \overline{A} + \overline{B}$$
$$\overline{A+B} = \overline{A}\,\overline{B}$$

From this property, you can write a three-input AND in terms of our fundamental two-input NAND gates:

$$ABC = \overline{\overline{A} + \overline{B} + \overline{C}} = \overline{\overline{A} + \overline{BC}} = \overline{A\left(\overline{BC}\right)}$$

- 4 -

with an inverter on one of the inputs. Note that since a double bar on top is the identity operator, you could have written the third expression directly, using the associative property, as:

$$\overline{ABC} = \overline{A(BC)} = \overline{A\left(\overline{\overline{BC}}\right)}.$$

This notation does have the problem that it can be very confusing to keep track of all the over-bars.

TTL Gates

TTL (Transistor-Transistor Logic) gates were the first robust, fast, commonly available digital circuit elements. TTL and CMOS (Complementary Metal Oxide Semiconductor) remain the two most common types of discrete logic chips. Although they have different input and output impedances, they both have the same input and output voltage levels. The standard voltage levels for TTL logic are Low = ground, and High = 5 Volts. All integrated circuits (ICs) must have a ground (GND) and a power supply (V_{cc}) in addition to its connections for input and output.

Many TTL inputs will "float high" if you do not connect them to anything. This will change a NAND gate into an inverter, if only one input is connected, but it will render a NOR gate useless. Sometimes an unconnected input oscillates between high and low due to stray capacitive loads. These oscillations can be initiated by a radio station signal or as you move your hand over a circuit due to static charges. In general, unconnected inputs produce suspect outputs and make for circuit-debugging headaches. It is a good general rule to terminate any unused inputs by connecting them to another input, to GND, or to V_{CC}, according to your gate type and usage.

III. Binary Numbers & Math

The common and relatively simple operation in computing is integer arithmetic (addition and subtraction). This can be used to update a counter, compute a location in memory, or perform a requested mathematical operation.

The essence of digital circuitry is that a digital signal is forced to be one of two values, which are symbolically represented as 0 or 1. Given the digital nature of computers and other electronic devices, it is most convenient to represent numbers using only 0s and 1s. This is called a *binary* representation (or base two).

A. Binary Numbers

In our normal math we use ten digits for a *decimal* (base ten) representation of numbers. We make up a multi-digit number using the following scheme:

$$2\times10^3 + 9\times10^2 + 8\times10^1 + 4\times10^0 = 2000 + 900 + 80 + 4 = 2984$$

The base for a number is often indicated by a subscript such as "2984_{10}", where the subscript is always in base ten. Decimal is assumed if no subscript is shown.

We can use a similar algorithm to convert back from a binary representation to decimal representation.

$$10110_2 = 1{\times}2^4 + 0{\times}2^3 + 1{\times}2^2 + 1{\times}2^1 + 0{\times}2^0 = 16 + 0 + 4 + 2 + 0 = 22$$

To convert binary to decimal representation one divides by two repeatedly and writes down the remainders. To convert 13_{10} to binary

$13/2 = 6$ remainder 1

$6/2 = 3$ remainder 0

$3/2 = 1$ remainder 1

$1/2 = 0$ remainder 1

From which we can conclude the $13_{10} = 1011_2$. Note that the digits come out in the order right to left.

A single binary digit is known at a *bit*. With n bits one can represent 2^n different numbers. The highest binary number one can represent in n bits is given by 2^n-1 (since 0 is one of the numbers). In a binary number, the leftmost digit is the *most significant bit* (MSB) and the rightmost digit is the *least significant bit* (LSB).

Hexadecimal numbers

Note that it took 5 bits to represent a relatively small number like 22 in binary. Binary numbers have too many digits to write out for anything but the smallest numbers, so they are usually abbreviated in *hexadecimal* notation (base sixteen) or occasionally in *octal* (base eight). In hexadecimal notation, four binary digits are grouped together and represented by a single "digit" between 0 and 15. To force the digit to a single place, we will use *a*, *b*, *c*, *d*, *e*, and *f* for the digits 10 through 15. A standard notation in computer programming is to denote a hexadecimal number with a 0x preceding it. Thus,

$$0x16 = 1{\times}16^1 + 6{\times}16^0 = 16 + 6 = 22$$

and

$$0x2f = 2{\times}16^1 + 15{\times}16^0 = 32 + 15 = 47$$

Two hexadecimal digits can be combined to form an eight-bit *byte*. A byte can represent numbers from 0 and 255 (0xff).

B. Binary Addition

As the semester progresses we will start to use binary and hexadecimal numbers extensively. We will be doing conversions between bases and doing arithmetic in these alternate bases. To start with, let us consider addition of multi-digit binary numbers. Our goal is to try to understand the logic of the process in enough detail that we can generalize it into a digital circuit.

The general process looks like normal addition. Here are some examples that encompass all portions of the operation.

$$0101_2 + 0010_2 = 0111_2$$
$$0101_2 + 0001_2 = 0110_2$$
$$0111_2 + 0001_2 = 1000_2$$

The main differences between decimal and binary addition are that the binary one, on average, carries half the time and there are only a limited number of possible operands and resultants for each place in the operation (i.e. only 1s or 0s).

If we just look at a particular digit, there are up to three bits that are inputs to addition. These are the ith digit of the first number, the ith digit of the second number and an additional bit that tells us if there was a "carry" from adding the previous digits. The resulting answer can have values of 00_2, 01_2, 10_2, or 11_2. For the first two cases there would not be a carry to the next digit. In the last two cases, there would be a carry, which is represented by the MSB.

C. Karnaugh Maps

A Karnaugh map is useful for trying to define the logic for circuits with up to four inputs and a single output. An example of a three-input map is shown in Figure 2-1. A Karnaugh map is made with the following steps:

1. Make a truth table. Put in L or H or X ("don't care") for the inputs and the resulting outputs one expects.

2. Set up the map. Put up to two inputs on each of the two axis. As you fill in the values make sure that no more than on input bit changes from one row to the next and from one column to the next.

3. Fill in the map based on the truth table

To use a map to make up a logical expression:

1. An entire row or column indicates a simple AND of a single variable.

2. An adjacent pair of cells with "H" indicates an AND relationship with a pair of

the row and column variables.

3. An adjacent pair of cells with "L" indicates an OR.

4. Once all the of either the "H" or "L" cells are encompassed in simple logical expressions they are put together with ORs or ANDs to make a final logical expression.

This simple methods of maping a sets of ANDs and Ors will work and make a valid expression. It will likely not be the most efficient way of making the expression (i.e. will use extra gates). All of the logical operators make a different pattern on the map. You can play with these patterns to come up with mode efficient solutions.

D. Negative True Inputs

As time goes by this semester we will see a number of reasons for inputs coming into a circuit to have their "true" value as a ground. For example, switches which are monitoring equipment often are closed to ground in their normal state and are open when off (e.g. a window switch for a security system). Another reason is that a ground state usually uses less power so it is a better "resting" state for outputs that rarely are used to "assert" a 0. These are called *negative true* inputs.

These sorts of inputs seem to fly in the face of our definitions. To make it more intuitive, we simply pretend that they have been inverted before they reach our circuits. We use the standard inverted notation for these negative true inputs (e.g. $\overline{C}$). This situation is so common in practice that that we will see that most chips actually expect negative true inputs on certain parts of the circuit.

E. Assertion-Level Logic Notation

The routine use of negative true inputs can make the purpose of a circuit harder to interpret. We saw previously that we can make a number of transformations using inversion to, for example, see how an OR and be use to replace an AND in a circuit. We can use these inversion properties to make the logic easier to interpret when negative true inputs are present by putting inversions at the inputs.
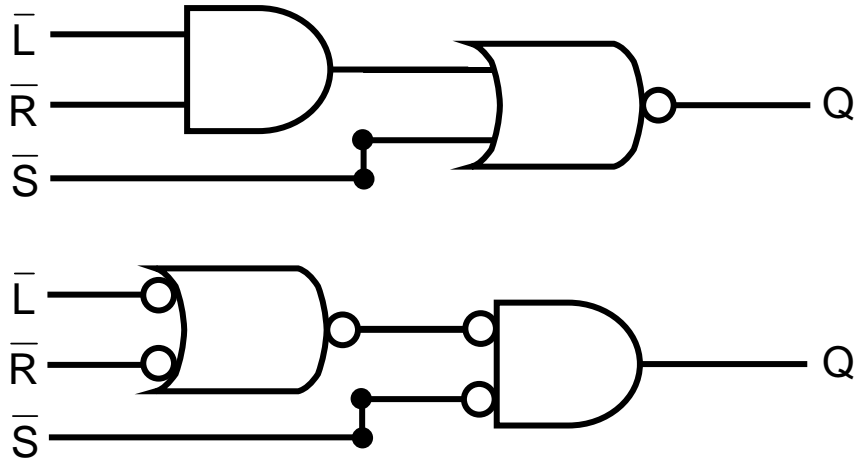
This is illustrated in the example shown in Figure 2-1. Both circuits implement a circuit that performs the operations $Q = (R+L)S$ with negative true inputs. The upper circuit uses only standard gates. Note how the use of negative true inputs makes the implementation hard to understand by simple inspection because the $R+L$ operation is replaced by $\overline{\overline{R}\,\overline{L}}$.

The figure below clarifies the situation by using inversions at the input to make them look like normal positive-true inputs. Then the gates actually look like the operations we intend to perform. It makes it a lot easier for someone looking the schematic to decipher how it functions. This method of trying to emphasize the logic instead of using standard gate symbols is known as *assertion level logic*.

The price one pays with assertion level logic is that the gates are not the standard

gates that one finds on chips. To build a circuit from such a schematic involves using an extra step (application of DeMorgan's Theorem) to get the final gates needed to actually build the device. Therefore, assertion level logic is usually used to show people what you have built rather than for construction diagrams.

Figure 1-5: Here are two circuits that perform the operations Q = (L+R)S with negative true inputs. The upper figure shows an example of a circuit that performs using standard gates.

Design Exercise 1-1: Construct the truth table for a three input NAND.

Design Exercise 1-2: Construct a circuit for a three input NAND, using three two-input NAND gates.

Design Exercise 1-3: Construct a truth table for a two input inverted XOR (called XNOR).

Design Exercise 1-4: Construct a circuit for a two input inverted XOR, using two-input NAND gates.

Design Exercise 1-5: Convert 11 and 5 to binary, add them together, and convert the resultant back to decimal. Did you get 16?

Design Exercise 1-6: Design a truth table for a circuit to add two binary digits and a carry bit. It will have three digital inputs and produce two digital outputs (the resulting digit and a carry bit). This is the truth table for a generalized 1-bit 2-input adder that can be implemented within a multi-bit 2-input adder.

Design Exercise 1-7: Design a Karnaugh map for the resulting digit.

Design Exercise 1-8: Design a Karnaugh map for the carry bit.

Design Exercise 1-9: Design a circuit to make one of the outputs from adding two binary digits. The circuit should produce a single output that contains the resulting digital sum (without carry). You can use any of the standard one and two-input logical gates in your design (e.g. AND, OR, NAND, NOR, XOR, and NOT).

Design Exercise 1-10: Design a circuit to make the other output from adding two binary digits and a carry bit. The circuit should produce the resulting carry bit. You can use any of the standard one and two-input logical gates in your design (e.g. AND, OR, NAND, NOR, XOR, and NOT).

Design Exercise 1-11: Install Quartus II FPGA circuit design software on your computer and request a license from Altera Inc.