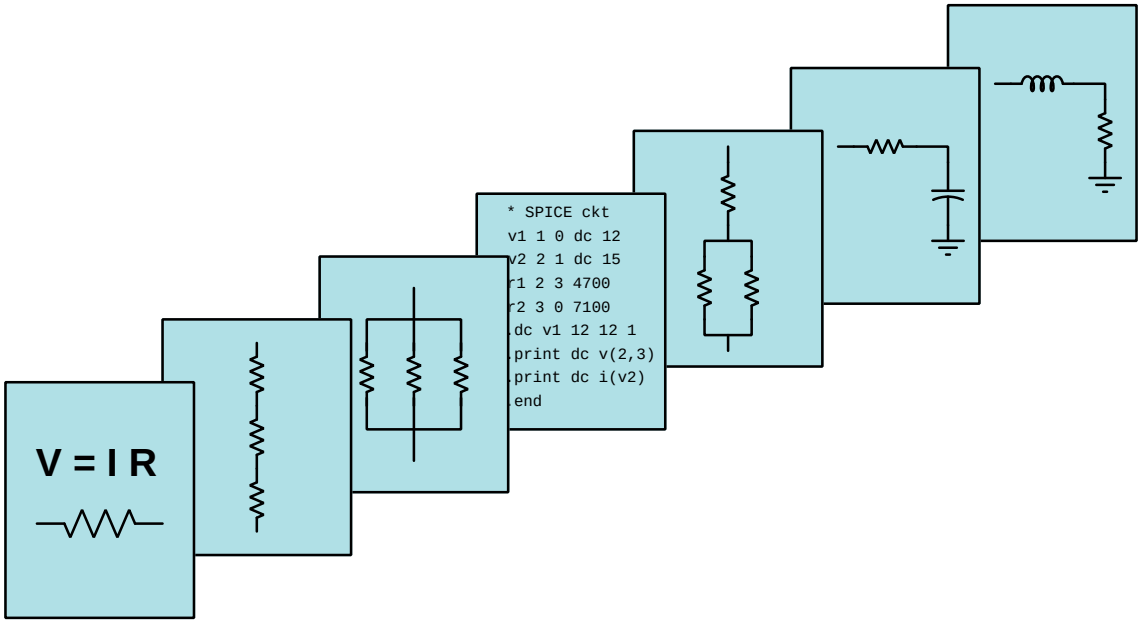
Modular Electronics Learning (ModEL) project



Basic Principles of Digital

© 2018-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 24 April 2026

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to digital circuits . . . 5
- 1.3 Recommendations for instructors . . . 6

2 Case Tutorial **7**
- 2.1 Example: simple logic functions using switches . . . 8
- 2.2 Example: inverter demonstration circuits . . . 9
- 2.3 Example: NAND gate demonstration circuit . . . 11
- 2.4 Example: comparator demonstration circuit . . . 13
- 2.5 Example: simple bargraph display circuit . . . 14
- 2.6 Example: bitwise logical operations . . . 15
  - 2.6.1 Bitwise-AND . . . 15
  - 2.6.2 Bitwise-OR . . . 15
  - 2.6.3 Bitwise-XOR . . . 16
  - 2.6.4 Bitwise-complement . . . 16

3 Tutorial **17**
- 3.1 Analog versus digital . . . 17
- 3.2 Logic states . . . 19
- 3.3 Bits and words . . . 21
- 3.4 Logic functions . . . 22
- 3.5 IC logic gates . . . 25
- 3.6 Boolean expressions . . . 26
- 3.7 Digital numbers and codes . . . 29
- 3.8 Digital storage and communication . . . 30
- 3.9 Comparators . . . 31

4 Historical References **33**
- 4.1 The original telegraph code . . . 34
- 4.2 Punched paper tape . . . 35
- 4.3 Claude Shannon makes the connection . . . 38

5 Derivations and Technical References **41**
5.1 Process switch types and symbols . . . 41

6 Programming References **43**
6.1 Programming in C++ . . . 44
6.2 Programming in Python . . . 48
6.3 Modeling Boolean logic using C++ . . . 53

7 Questions **63**
7.1 Conceptual reasoning . . . 67
7.1.1 Reading outline and reflections . . . 68
7.1.2 Foundational concepts . . . 69
7.1.3 Analog versus digital quantities . . . 71
7.1.4 Switch states . . . 72
7.1.5 Designing switch/lamp circuits . . . 73
7.1.6 Pulse-width modulation . . . 74
7.1.7 Logic levels in a simple transistor circuit . . . 75
7.1.8 Simple logic probe circuit . . . 76
7.1.9 Logic levels in a bipolar logic circuit . . . 77
7.2 Quantitative reasoning . . . 78
7.2.1 Miscellaneous physical constants . . . 79
7.2.2 Introduction to spreadsheets . . . 80
7.2.3 Using Python to evaluate basic logic expressions . . . 83
7.2.4 Logic gate truth tables . . . 85
7.2.5 Boolean expressions for logic functions . . . 86
7.2.6 Boolean expressions for simple combinational networks . . . 87
7.3 Diagnostic reasoning . . . 88
7.3.1 Electrical faults in a simple battery/lamp circuit . . . 89
7.3.2 Strange arithmetic . . . 90
7.3.3 Effects of faults in simple digital circuit . . . 91

A Problem-Solving Strategies **93**

B Instructional philosophy **95**
B.1 First principles of learning . . . 96
B.2 Proven strategies for instructors . . . 97
B.3 Proven strategies for students . . . 99
B.4 Design of these learning modules . . . 100

C Tools used **103**

D Creative Commons License **107**

E References **115**

F Version history **117**

Index **119**

Chapter 1

Introduction

1.1 Recommendations for students

The purpose of this module is to introduce the fundamental concepts of digital signals, in preparation for further explorations into this very broad and very deep subject. The concept of discrete electrical signals will be introduced in two forms: switch-based circuits and transistor-based circuits. Then, the concept of *logical functions* will be explored, where multiple discrete signals come together to generate a resultant discrete signal. Next, we will explore a special form of mathematical representation for discrete signals called *Boolean algebra*. Lastly, we will see how collections of discrete states are used to represent different kinds of information.

Important concepts related to digital logic and digital signals include **analog** versus **digital** quantities, **logic states**, **logic levels**, **high** and **low** logic states, **logic functions**, **truth tables**, **Boolean algebra**, **binary** numeration, and **encoding**.

Here are some good questions to ask of yourself while studying this subject:

- How do analog and digital quantities differ from one another?
- What are some common examples of analog and digital quantities we experience in life?
- How does the number of *bits* in a digital *word* affect the number of possible values for that word?
- What sorts of information may we represent using digital words?
- How are digital logic states represented as electrical voltage signals?
- What does a *truth table* represent?
- What is a *logic function*?
- What are some basic forms of logic functions?
- How do logic functions relate to arithmetic functions?

- How may we use lettered variables to represent logic states and logic functions?
- What is a "binary" number?
- How can we convert a binary number into a decimal number?
- What are some different ways we can physically represent digital bits?

1.2 Challenging concepts related to digital circuits

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Non-decimal numeration** – decimal numeration is so commonplace that many people assume this is the *only* possible way to represent numbers. So, when faced with non-decimal numeration schemes such as binary, octal, or hexadecimal the first impression may be unnerving. Fortunately, though, these are all *place-weighted* systems which means a close examination of how decimal actually works will shed light on how the others work too.

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

  Assessment – Students present their "Conceptual Toolbox" Theory Journal entries listing important concepts applied in the Tutorial text(s). A "Conceptual Toolbox" serves as inspiration when solving problems, each concept suggesting a potential problem-solving action. Challenge students to collaboratively generate a Conceptual Toolbox for the day's topic during the class session and then use it when solving problems together, thus fostering the habit of reasoning from principles rather than recalling procedures!

- **Outcome** – Apply the concept of digital logic levels to a transistor circuit

  Assessment – Predict high/low voltage logic levels within a discrete multi-transistor circuit with various input conditions; e.g. pose problems in the form of the "Logic levels in a bipolar logic circuit" Conceptual Reasoning question.

  Assessment – Complete truth tables for elementary logic function types; e.g. pose problems in the form of the "Logic gate truth tables" Quantitative Reasoning question.

- **Outcome** – Apply the concept of Boolean expressions to simple logic circuits

  Assessment – Write appropriate Boolean expressions for signal lines within simple combinational logic circuits; e.g. pose problems in the form of the "Boolean expressions for simple combinational networks" Quantitative Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate logic gate datasheets and properly interpret some of the information contained in those documents including supply voltage range, gate type(s), output current limitations, acceptable input signal voltage ranges for high and low states, etc.

Chapter 2

Case Tutorial

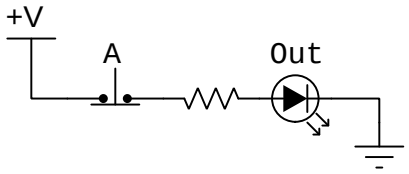
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: simple logic functions using switches

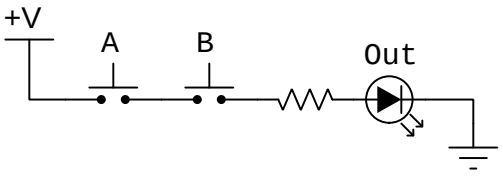
In each of the following circuits, the resistor serves to limit LED current to a safe value. A "1" state refers to a *pressed* pushbutton switch in the context of the logic function's input, and to an *energized* LED in the context of a logic function's output:

NOT function



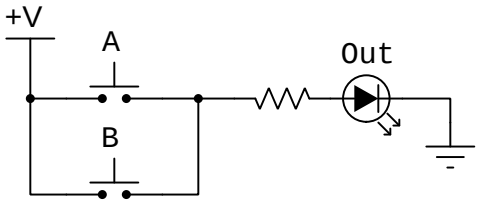
A	Out
0	1
1	0

AND function



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

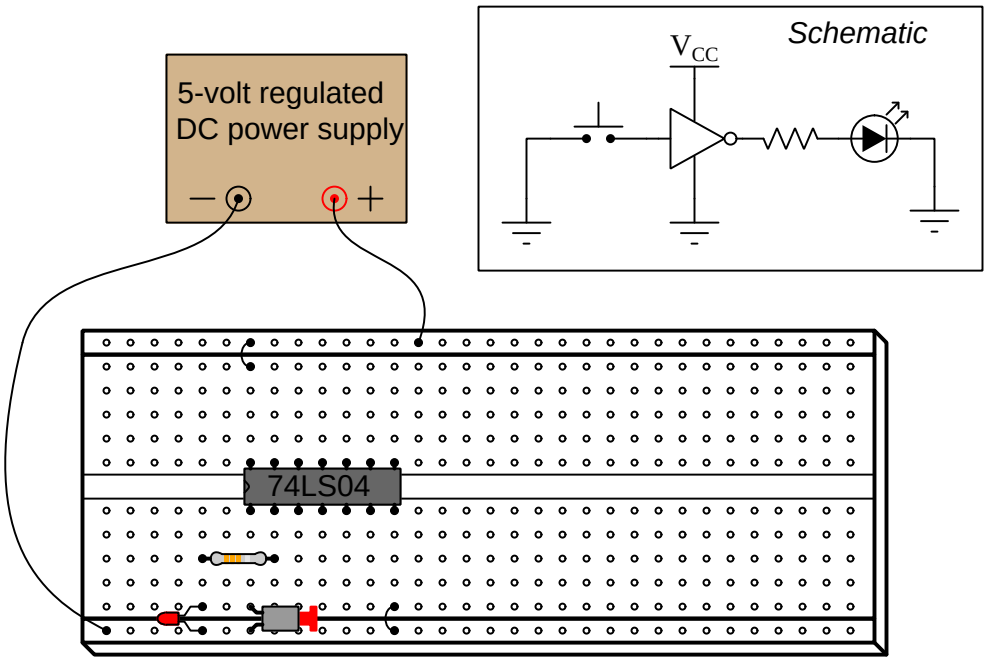
OR function



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

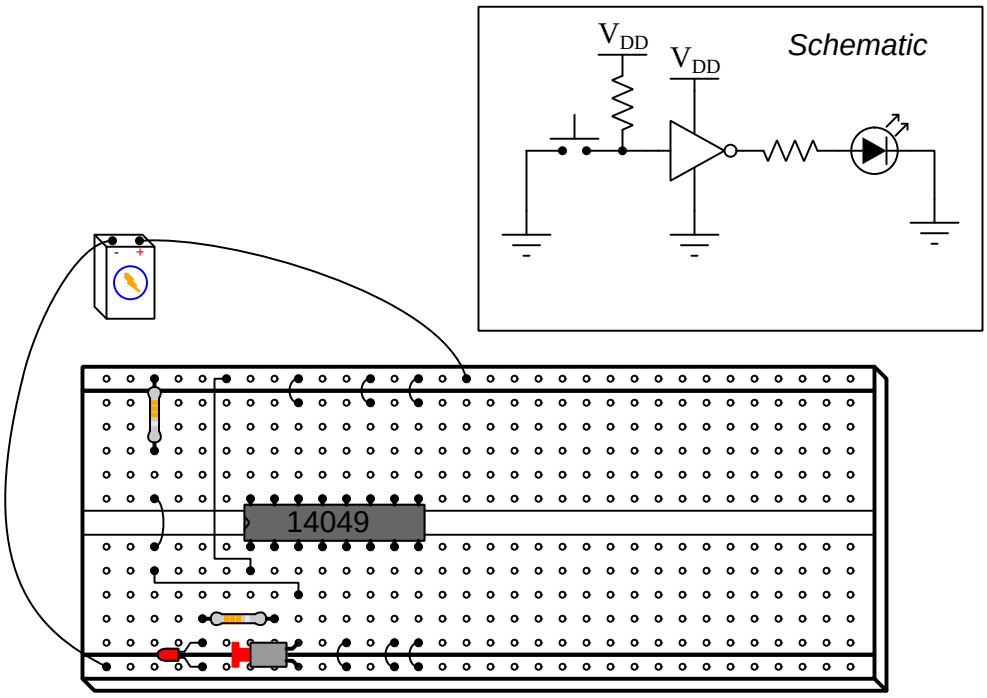
2.2 Example: inverter demonstration circuits

The DIP circuit is a TTL hex inverter (it contains *six* "inverter" or "NOT" logic gates), but only one of these gates is being used in this circuit. Being TTL, it requires a regulated power supply voltage of 5 Volts:



Research a manufacturer's *datasheet* for this logic gate IC to see the "pinout" diagram showing which pins on the IC package connect to which inverter gate terminals inside.

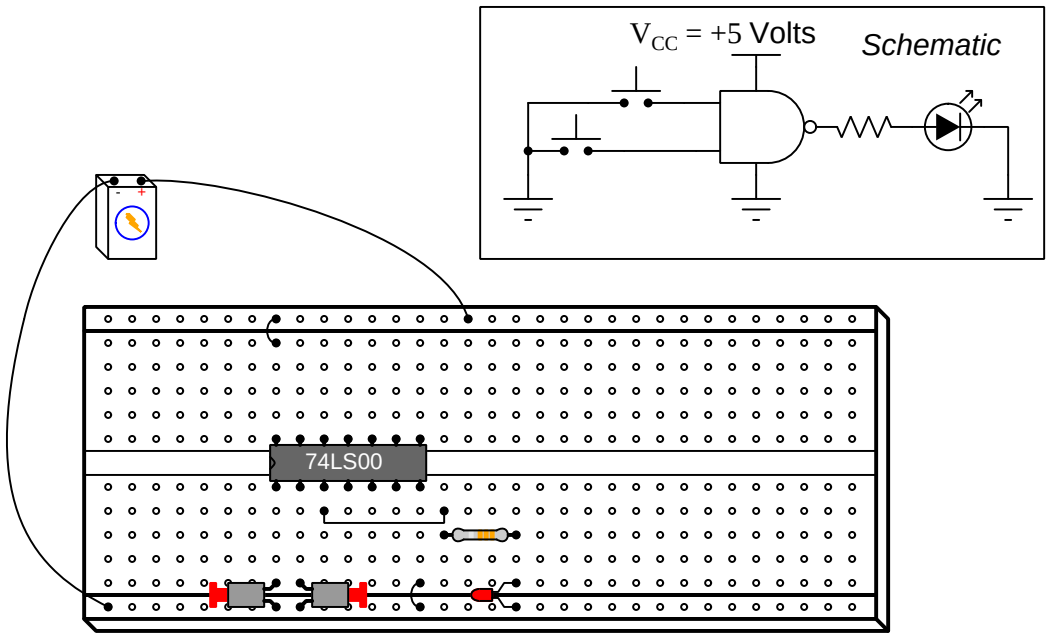
Next we have a CMOS version of the same circuit. Being CMOS, the power supply voltage requirements are not as strict, but we must use a pull-up resistor to ensure a "high" logic state at the input terminal when the pushbutton switch contacts are open. This is the same reason that all unused input pins are tied either to +V ("high") or ground ("low"). Unlike TTL, whose inputs default to a "high" state when floating, CMOS gate inputs assume no default state and therefore must be given connections to either V_{DD} (+V) or V_{SS} (ground) in order to avoid ambiguous states:



Note the unusual locations of the +V power supply pin on this IC: pin 1 rather than the highest-numbered pin (upper-left) as we more commonly find.

2.3 Example: NAND gate demonstration circuit

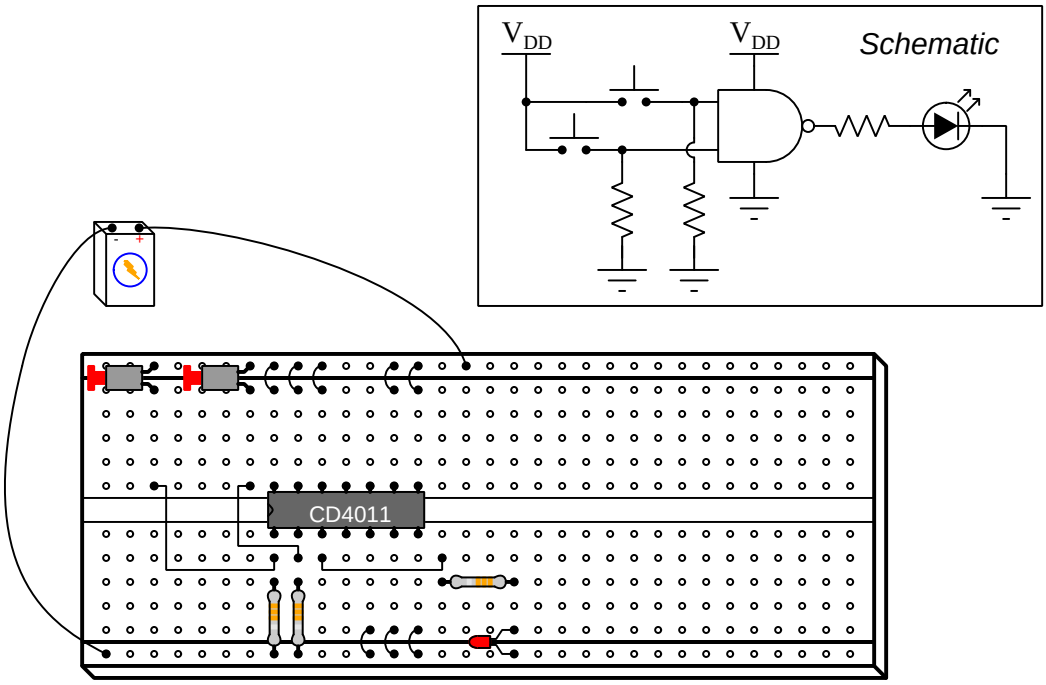
Below we see a schematic diagram as well as pictorial illustration of a NAND gate demonstration circuit using the model 74LS00 quad 2-input NAND gate IC. This particular IC uses bipolar (PNP and NPN) type transistors with low-power Schottky diode architecture (hence the "LS" designation for "Low-power Schottky"), and requires a tightly voltage-regulated power supply at 5 Volts in order for those transistors and diodes to function well together. An interesting characteristic of bipolar-type semiconductor logic gates is that any unconnected input terminal will assume a "high" logical state, which explains why the switches connect to ground. This means pressing each normally-open pushbutton switch will assert a "low" logical state at that input, while releasing each switch lets the gate's input return to its default "high" state:



Research a manufacturer's *datasheet* for this logic gate IC to see the "pinout" diagram showing which pins on the IC package connect to which NAND gate terminals inside. Feel free to research datasheets and build demonstration circuits like this one using types of TTL logic gates other than the 74LS00 quad 2-input NAND:

- 74LS08 – quad 2-input AND gate
- 74LS32 – quad 2-input OR gate
- 74LS02 – quad 2-input NOR gate
- 74LS86 – quad 2-input XOR gate

Next we see a CMOS version of this same circuit using the model CD4011 quad 2-input NAND gate IC. Being CMOS, the power supply voltage limits are quite wide which allows for battery power, but we must use pull-down resistors to ensure "low" logic states at the input terminals when the pushbutton switch contacts are open. This is the same reason that all unused input pins are tied either to +V ("high") or ground ("low"). CMOS gate inputs assume no default state when floating and therefore must be given connections to either V_{DD} (+V) or V_{SS} (ground) in order to avoid ambiguous states:

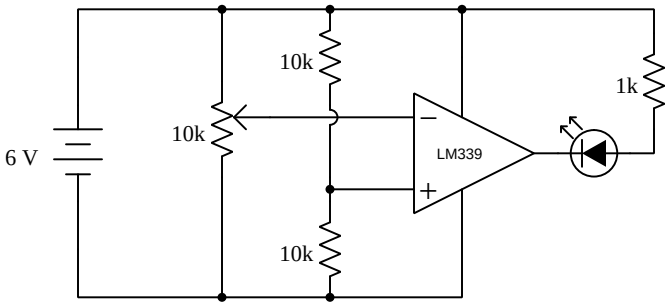


Research a manufacturer's *datasheet* for this logic gate IC to see the "pinout" diagram showing which pins on the IC package connect to which NAND gate terminals inside. Feel free to research datasheets and build demonstration circuits like this one using types of CMOS logic gates other than the CD4011 quad 2-input NAND:

- CD4081 – quad 2-input AND gate
- CD4071 – quad 2-input OR gate
- CD4001 – quad 2-input NOR gate
- CD4070 – quad 2-input XOR gate

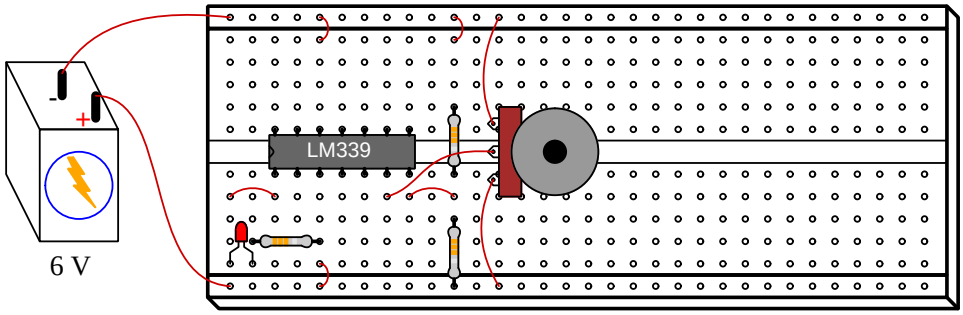
2.4 Example: comparator demonstration circuit

Comparators are hybrid analog-digital integrated circuits that compare two analog input signals to generate one digital output signal. In essence they *compare* one voltage signal against another and drive the output either "high" or "low" depending on which of those two analog voltage input signals are larger. The following demonstration circuit uses just one of the four comparators contained within a model LM339 integrated circuit to demonstrate how comparators function:



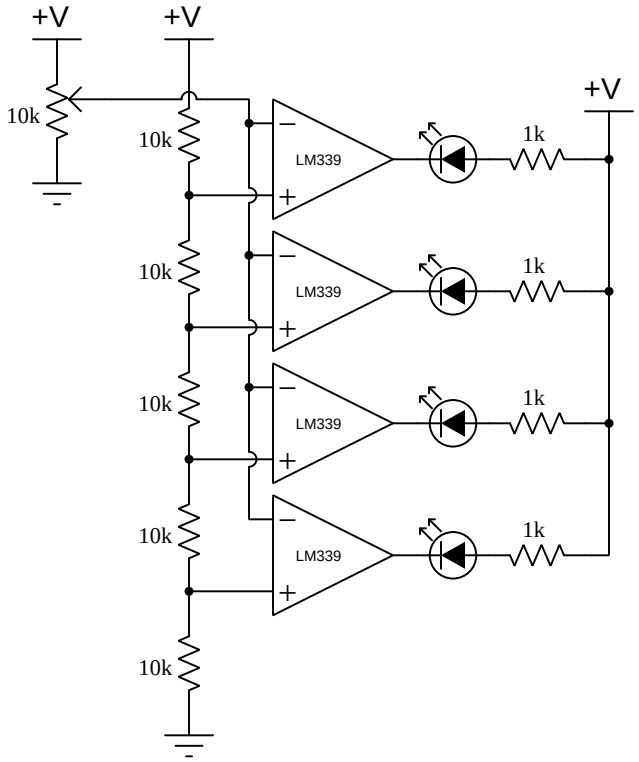
The purpose of the potentiometer is to serve as an adjustable voltage divider, to take the 6 Volt battery's voltage and divide it down to some adjustable value between 0 Volts and 6 Volts, inclusive. The two 10 kilo-Ohm fixed resistors also form a voltage divider, providing a fixed (non-variable) voltage equal to one-half of the battery's voltage.

Below is a pictorial representation of the LM339 IC and other components inserted into a solderless breadboard:

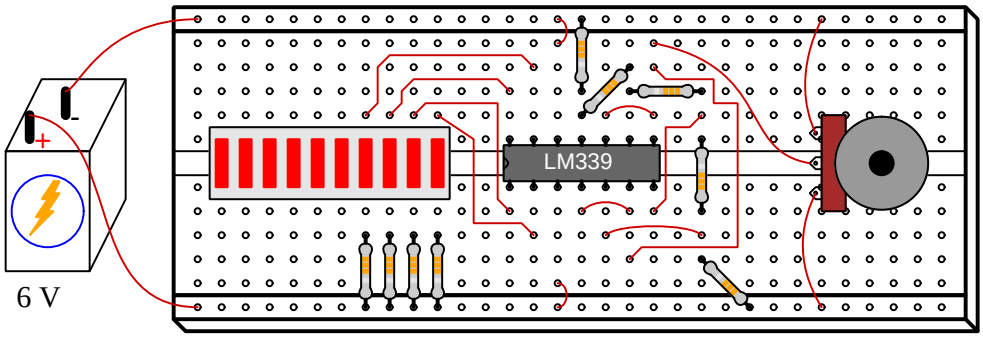


Note the unusual location of the LM339's power supply terminals: (+) on pin 3 and (−) on pin 12. Also note that the model LM339 cannot source output current, but can only sink. This is why the LED must be connected the way it is, sending current *into* the comparator's output terminal when energized.

2.5 Example: simple bargraph display circuit



While individual LEDs will suffice, a ten-segment LED module looks much nicer:

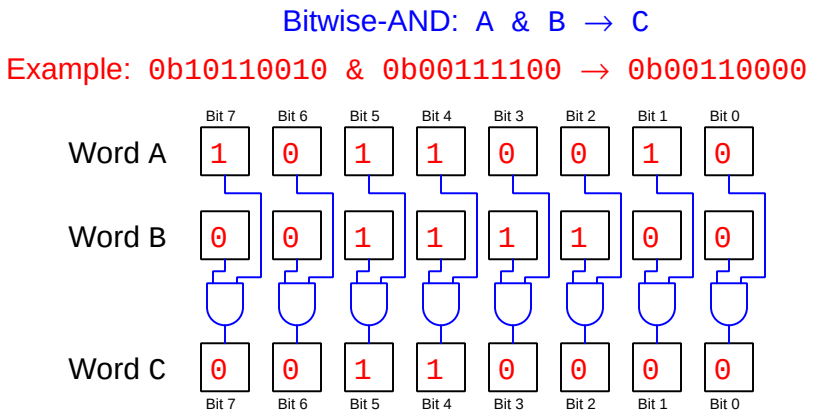


Note the unusual location of the LM339's power supply terminals: (+) on pin 3 and (−) on pin 12. Also note that the model LM339 cannot source output current, but can only sink. This is why the LEDs must be connected the way they are, sending current *into* the respective comparator output terminals when energized.

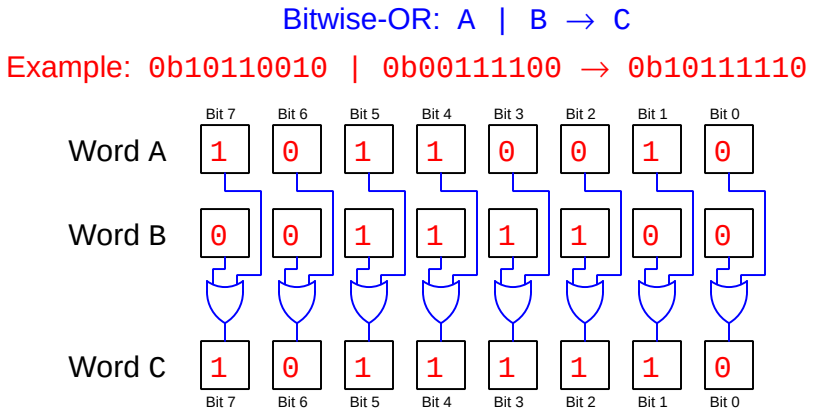
2.6 Example: bitwise logical operations

The following examples show bitwise operations being performed on 8-bit words (bytes).

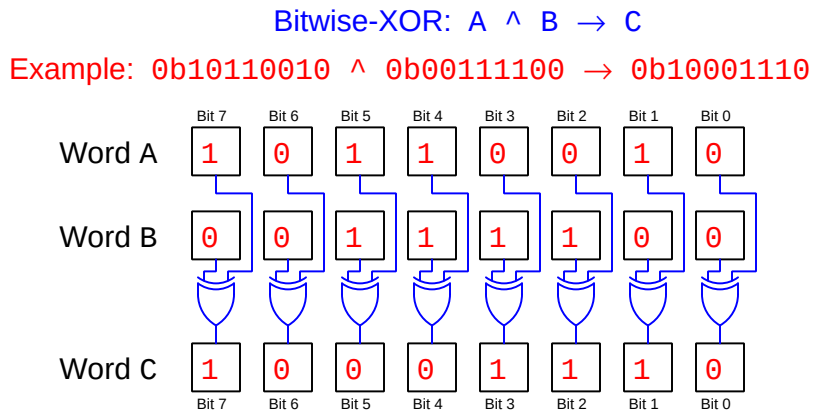
2.6.1 Bitwise-AND



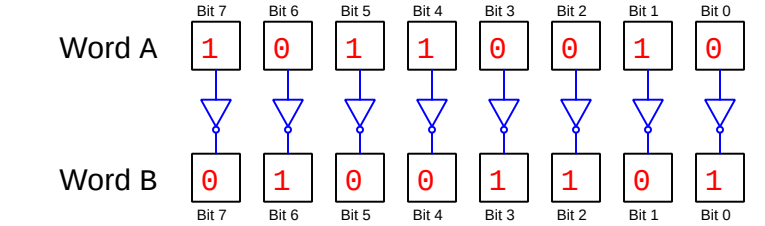
2.6.2 Bitwise-OR



2.6.3 Bitwise-XOR



2.6.4 Bitwise-complement



Chapter 3

Tutorial

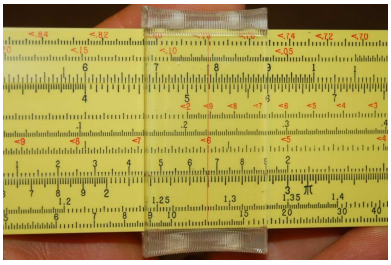
3.1 Analog versus digital

Physical quantities such as electrical voltages and currents are often used as *signals* to represent other types of quantities. Such representations may be broadly categorized into two forms: *analog* and *digital*. An "analog" representation is one capable of being infinitely subdivided, whereas a "digital" representation is inherently limited to discrete positions or states.

This contrast may be clearly seen by comparing analog versus digital representations of identical quantities. Take for instance the following two photographs of clocks – the left-hand photo showing an analog clock with hour, minute, and second hands continuously sweeping around the face of the dial to show the time at any given instant; the right-hand photo showing a digital clock with two decimal digits each showing hour, minute, and seconds:

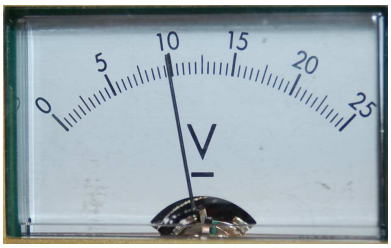
The analog clock's display of time is *continuous*, since the hands of that clock never stop moving. The digital clock's display is *stepped*, since the "seconds" display increments precisely at one-second intervals. Both clocks show the same time of day (1:50 PM and 13 seconds, or 13:50:13 in "military" time format), but do so in completely different ways.

Another useful contrast is in the representation of numerical quantities. The following two photographs compare a *slide rule* with an electronic hand calculator's display of the same number:

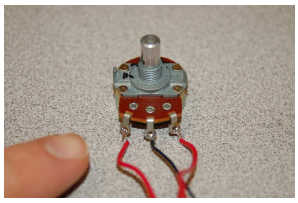


The slide rule, being a mechanical calculator in its own right, shows 2.5 beneath the thin red line (called a *cursor*) on the third scale from the bottom, while the electronic calculator shows the same quantity as a pair of decimal digits separated by a decimal point. As with the analog clock, the slide rule's ability to be positioned with arbitrary precision stands in contrast to the electronic calculator which is limited by the number of digits available on its display. This same property, however, makes the slide rule's display subject to interpretation while the electronic calculator's display is unambiguous.

Electrical quantities such as voltage and current may also be displayed in either analog or digital form. Compare the following photographs, one showing an analog voltmeter and the other showing a digital voltmeter, both displaying 10 Volts:



Similarly, a *potentiometer* and a *resistive decade box* are just analog versus digital devices for providing an adjustable quantity of electrical resistance:



The study of digital circuits is fundamentally about how we use on/off electrical states to represent different kinds of information, and how we construct circuits utilizing these on/off signals to perform useful functions.

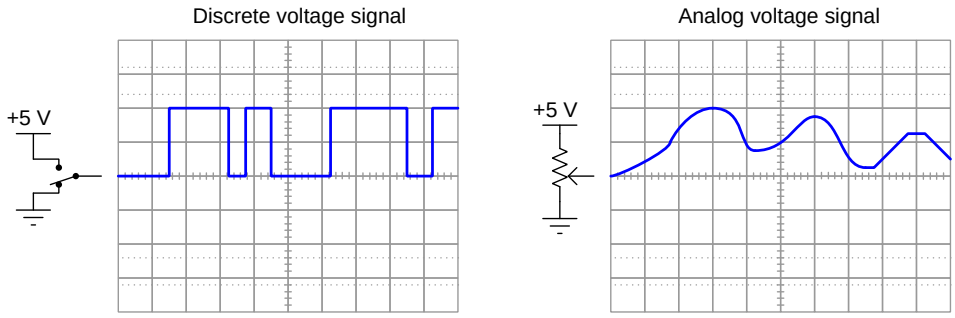
3.2 Logic states

The most elementary form of digital electrical signal is called *discrete*: a signal with only two possible states: *true* and *false*, or 1 and 0, respectively. The minimum and maximum electrical signal levels defining these *logic states* vary by circuit design. A circuit powered by a fixed 5 Volt DC source, for example, may have "logic level" input thresholds of $V_{in} \leq 1.5$ Volts and $V_{in} \geq 3.4$ Volts defining 0 and 1 states, respectively[1].

In the following illustration a single-pole double-throw (SPDT) switch generates a discrete voltage signal. When flipped in the downward direction, the "common" terminal makes contact with ground, making the output signal "low"; when flipped upward, the "common" terminal makes contact with the positive terminal of a 5 Volt power supply, and the signal is now "high". In most[2] digital systems, a "low" voltage value represents a "false" or 0 logic state while a "high" voltage value represents a "true" or 1 logic state:

Discrete signals represent the lower-limit-case of the digital signal realm. That is to say, discrete signals are *digital* in that they possess a limited number of possible states, and this limit is as *low* as possible (having only two states: on and off).

If we examine discrete versus analog signals using an oscilloscope, the difference is quite clear. Setting the oscilloscope's timebase to a slow sweep rate (e.g. one second per division) and then recording the voltage over several seconds' worth of time as we manually manipulate the switch and potentiometer shows us how the analog signal is able to reach any level of voltage between the source's full potential and zero while the switch's discrete signal can only assume the extreme states of full potential versus zero:

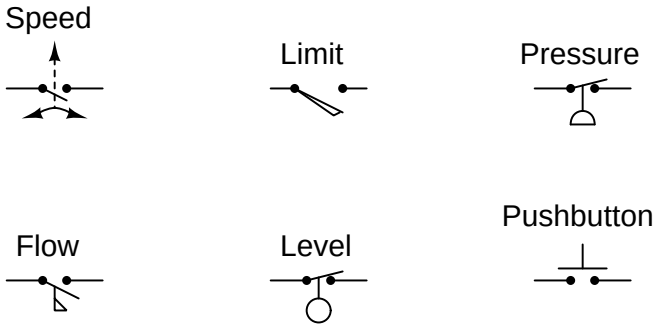


[1] At first one might think it makes more sense to define these states at the extreme limits of 0 and 5 Volts, respectively, but doing so would cause problems if ever noise voltage were superimposed on the signal. One of the benefits of a properly-designed digital system is *noise tolerance*, where modest amounts of noise imposed on the electrical signal will not corrupt that signal's discrete state.

[2] Defining 1 as "high" voltage and 0 as "low" voltage is called *positive logic*. Negative logic systems do exist, where 0 is defined by a "high" voltage level and 1 by a "low" voltage level.

Many industrial applications lend themselves to representation by discrete electrical signals. Devices called *process switches* designed to sense such physical variables as fluid pressure, temperature, fluid flow rate, liquid level may be used to signal whether the measured quantity is "high" or "low" according to some pre-set threshold. *Proximity* switches and *limit* switches detect the position of physical objects, limit switches by direct contact and proximity switches by remote sensing using electric fields, magnetic fields, or beams of light. These discrete signals may be used for practical purposes such as automatically starting and stopping motor-driven machinery, redirecting objects on an assembly line, self-calibrating measurements of position or distance, signaling important status conditions to human operators, counting production tallies, etc.

An assortment of schematic diagram symbols representing different types of switches used in industrial measurement and control systems appears below:



3.3 Bits and words

Discrete signals may have individual meaning, or they may be grouped together for collective meaning. In the latter sense, each discrete signal is called a *bit* and the collection of bits is called a *word*. The number of possible states for any digital word is 2 raised to the power of the number of bits. For example, a four-bit word has 2^4 or sixteen possible states, while an eight-bit word has two hundred and fifty six possible states. Digital words are incredibly useful, as they may be made to represent quite literally *anything* divisible by a fixed number of states. Examples include characters in an alphabet, numerical values, colors, etc. This is how digital computers symbolize all data: by combinations of bits. Furthermore, the discrete nature of bits makes them easy to express in physical form: holes punched in mylar tape, magnetized spots on a ferrite disk, pits burned into a reflective optical surface, electric charge stored within a capacitor, pulses of light conducted along a glass fiber, etc.

For example, the following list of numbers shows all possible combinations of 1's and 0's for a four-bit word:

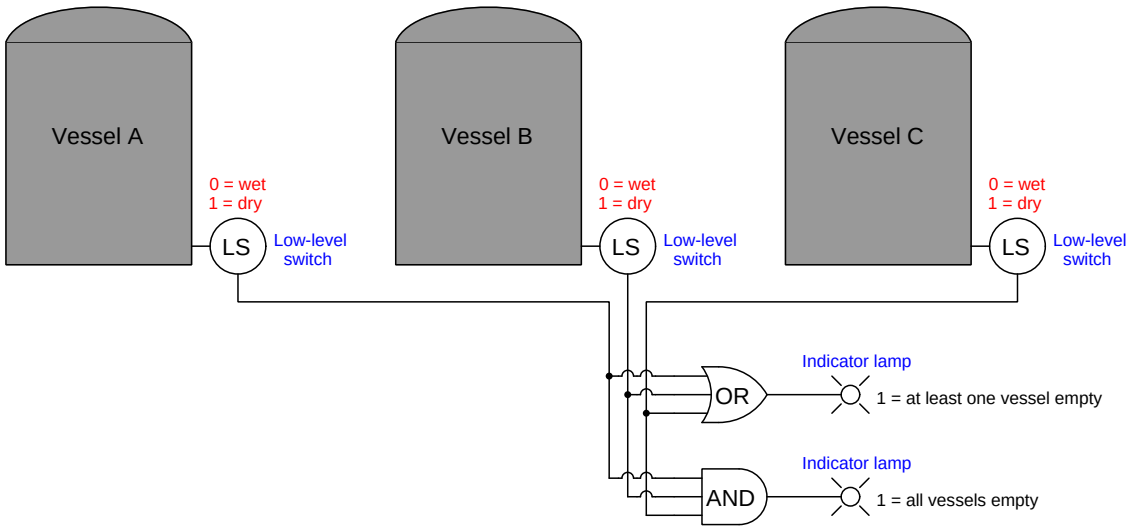
```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Note how exactly *sixteen* unique combinations of four bits exist in this list. This is because each bit was only two possible states (1 or 0) and there are four bits in the word, so $2^4 = 16$.

3.4 Logic functions

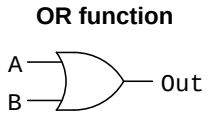
A *logic function* takes one or more discrete input signals and generates at least one discrete output signal in response. These functions are the "decision-making" elements of any digital system, and may be used for a wide range of purposes. Take for example a scenario where three large storage vessels contain water intended for human consumption, and the people using this water storage system desire a simple means of knowing if any or all of the tanks become empty.

The following illustration shows a simplified diagram of a digital logic system for this three-vessel water storage system. Each vessel has a low-level water switch installed near the bottom set to output a "high" (1) signal if ever the water falls below that level, and these three switch signals connect to the inputs of two different logic functions, an *OR* function and an *AND* function:

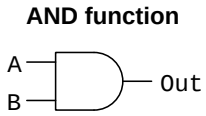


The *OR* function activates the upper indicator lamp if either vessel A or vessel B or vessel C goes dry. The *AND* function activates the lower indicator lamp only if all three vessels go dry. It is also possible to have these logic function activate more than just lamps, too: consider the possibility of having water pumps automatically turned on or off based on these tanks' levels.

An *OR* function outputs a high (1) state if *any* input is high (1). An *AND* function outputs a high (1) state if *all* its inputs are high (1). A common way to show the operation of logic functions is by a table of discrete (0 and 1) values called a *truth table*. Truth tables for two-input OR and AND functions are shown below:

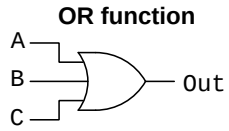


A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

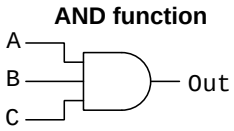


A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Truth tables for three-input OR and AND functions, such as the types used in our water storage system, are not much different. Regardless of the number of inputs, an OR function outputs a high (1) state if *any* input is high (1), and an AND function outputs a high (1) state only if *all* inputs are high (1):



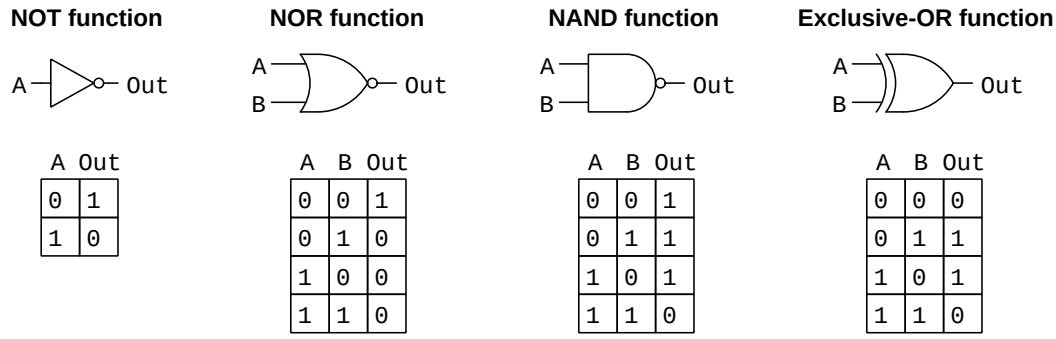
A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Interestingly, it matters not to the operation of this water system exactly *how* these logical functions are implemented. The OR and AND functions could be constructed from electromechanical relays, from semiconductor (transistor) electronic circuits, or even be software functions programmed into a computer. Each of these manifestations are not only possible, but indeed are practical and quite commonly applied in modern systems. We could even envision some technology not yet in existence that would behave in the manner of OR and AND functions, hypothetically applying that technology to this scenario. For now the important point is to grasp the abstract concept of "OR" and "AND" logic functions – those other details will come later.

OR and AND functions are only two examples of discrete logic, as many other logic functions exist. Some of the more common are shown in the following illustration complete with truth tables[3]:



A	Out
0	1
1	0

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

The *NOT* function simply inverts whatever logic state it receives at its input, and for this reason is sometimes called an *inverter*. Inverter functions are quite useful in modifying logic signals and logic functions for different purposes.

The *NOR* and *NAND* functions are variations of *OR* and *AND* functions, respectively: the NOR being an OR function with an inverted output, and the NAND being an AND function with an inverted output. The inversion implicit within the NOR and NAND functions is represented in each case by a "bubble" symbol on its output terminal. Like OR and AND functions, NOR and NAND are also available with more than two inputs.

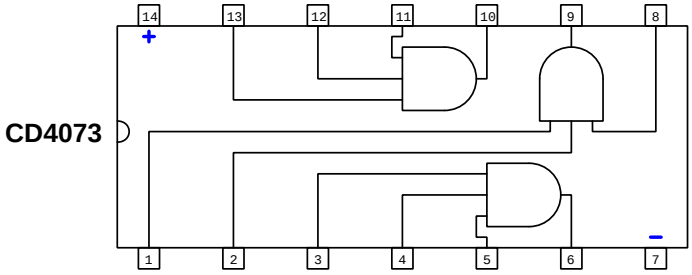
The Exclusive-OR (*XOR*) function outputs a "1" state when its two input states differ. This function is often used to *compare* the states of two bits to check for equality or inequality.

[3] A truth table shows all possible combinations of logic states for a function's inputs – equal to 2^n, where n is the number of discrete inputs – along with the corresponding output states. This is why the NOT function only has two rows in its truth table (because with only one input there are only 2^1 possible states), and why all the other functions with two input channels have four rows in their truth tables (because with two discrete variables there are 2^2 possible states). Logic functions having more than two inputs require 2^n rows in their truth tables, which is why the three-input functions we saw in the water storage system had eight rows in their truth tables.

3.5 IC logic gates

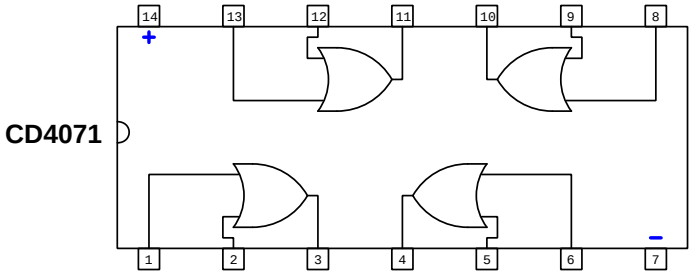
A common implementation of logical functions in electronic form is the *logic gate*, an integrated circuit (IC) made primarily of transistors interconnected to provide the desired logic functionality. The internal circuitry of these IC gates is beyond the scope of this Tutorial, but here we will explore some common IC gate *pinout* diagrams showing which pins on the device connect to which gate terminals.

First we will examine the pinout of the model CD4073 triple 3-input AND gate IC:



All CD4000-series ICs utilize MOSFET transistors as their active elements, and enjoy a wide DC supply voltage range (typically anything between 3 and 18 Volts DC is acceptable). Since these are *active*[4] electronic circuits, they require a constant source of DC voltage on which to operate. For this IC, pin 14 is the positive terminal for the external DC power source and pin 7 is the negative terminal.

Next we will examine another CD4000-series logic gate IC, the CD4071 quad 2-input OR gate:



Like the CD4073, the DC power supply terminals are 14 (+) and 7 (−), respectively, with the same wide permissible voltage range.

[4] An "active" component is one where an electrical signal controls another electrical signal. Transistors are perhaps the most common example of an active component. These components are a stark contrast to "passive" components such as resistors, capacitors, inductors, diodes, etc.

3.6 Boolean expressions

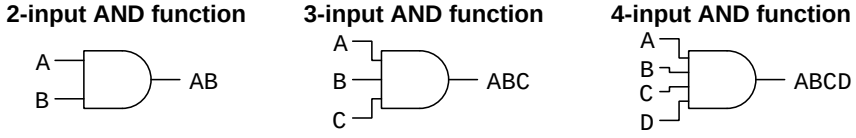
A special form of mathematics called *Boolean algebra* exists for expressing discrete (high/low, true/false) values, and is widely applied to digital circuits. In this mathematical system every variable is restricted to just two possible values – 0 and 1 – in keeping with their discrete nature. Certain logic functions such as AND and OR have arithmetic meaning within the Boolean scheme of mathematics: the AND function represents *multiplication* while the OR function represents *addition*.

Boolean algebra follows strict rules just like any form of mathematics, and these rules allow for the simplification of complex expressions. This makes Boolean expression a useful tool for logic circuit designers, enabling them to take a Boolean statement of some complex function and reduce that statement so that the function may be implemented using fewer circuit components.

The equivalence of OR functions with Boolean addition, and the equivalence of AND functions with Boolean multiplication, may be proven by comparing function truth tables with Boolean arithmetic statements. This next table compares statements for an AND function:

AND function truth statement	**Boolean multiplication statement**
0 and 0 = 0	$0 \times 0 = 0$
0 and 1 = 0	$0 \times 1 = 0$
1 and 0 = 0	$1 \times 0 = 0$
1 and 1 = 1	$1 \times 1 = 1$

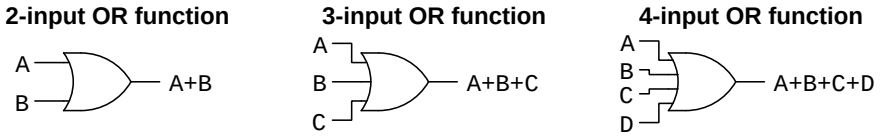
This table shows a common pattern between an AND function and Boolean multiplication: an AND function's output is 1 if and only if all input states are 1; a Boolean product is 1 only if all variables are 1. Here, we annotate AND symbol inputs and outputs using Boolean terms:



Next, we will compare truth table and Boolean arithmetic statements for an OR function:

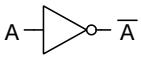
OR function truth statement	**Boolean addition statement**
0 or 0 = 0	0 + 0 = 0
0 or 1 = 1	0 + 1 = 1
1 or 0 = 1	1 + 0 = 1
1 or 1 = 1	1 + 1 = 1

As expected, this table shows a common pattern between an OR function and Boolean addition: an OR function's output is 1 if any input state is 1; a Boolean sum is 1 if any variable is 1. One strange-looking arithmetic statement appears in the last row, where we show the sum of 1 and 1 to be 1. In regular arithmetic we would expect the sum of 1 and 1 to be 2, but remember that this is *Boolean* mathematics, and in the Boolean world there is no such thing as 2. Since 1 + 1 certainly cannot be 0, we are left to conclude 1 + 1 = 1 as being the only sensible result. Here, we annotate OR symbol inputs and outputs using Boolean terms:



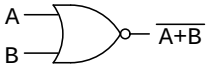
Logical inversion also has a Boolean equivalent, called *complementation* or *negation*. Since Boolean algebra admits of only two possible values for any variable (0 or 1), each value must have exactly one opposite. Different symbols have been invented to show the complement of a Boolean variable, the most common being a short line (called a "bar") drawn above the variable. For example, for any given state of the Boolean variable A, the opposite state could be expressed as $\overline{A}$. If $A = 0$ then $\overline{A} = 1$; if $A = 1$ then $\overline{A} = 0$. When verbally expressing a negated variable such as $\overline{A}$, you would say *not-A*, or *A-bar*. An older notation sometimes seen in typewritten text uses a single apostrophe (called a "prime" symbol)[5], for example A' which would be read as *A-prime*.

NOT function



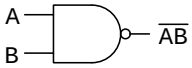
A	Ā
0	1
1	0

NOR function



A	B	A+B
0	0	1
0	1	0
1	0	0
1	1	0

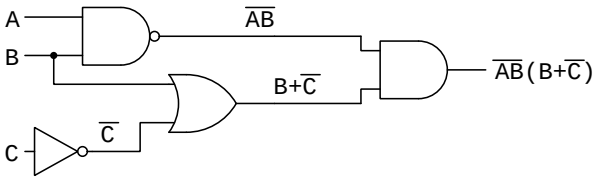
NAND function



A	B	AB
0	0	1
0	1	1
1	0	1
1	1	0

Note how the NOR and NAND expressions are merely inverted versions of OR and AND, respectively. Whereas a two-input OR function's output is $A + B$, a two-input NOR function's output is $\overline{A + B}$; whereas a two-input AND function's output is AB, a two-input NAND function's output is $\overline{AB}$. The output expression for these functions is nothing more than the complement[6] of their simpler counterparts.

Combinations of logical functions also lend themselves to Boolean expression. For example, consider the following combinational function, where each line bears a Boolean expression for its logical state:



The Exclusive-OR function previously mentioned may be expressed as a combination of AND, NOT, and OR functions: $\overline{A}B + A\overline{B}$.

[5] Typewritten text was limited by those characters present on the typewriting machine's keyboard, of which an overhead line or bar was not included. Text-based computer programming languages face a similar limitation, and for this reason alternatives to an overhead bar must be found. A common inversion operator for text-based programming languages is the word `not` prior to the variable. Some programming languages, notably C++, also permit the use of an exclamation point preceding the variable to denote inversion (e.g. `!A`).

[6] An alternative written expression for NOR and NAND logic functions uses the "prime" symbol, i.e. $(A + B)'$ for NOR and $(AB)'$ for NAND. When writing NOR and NAND functions in a text-based computer language, we must similarly group the variables within parentheses and then apply a negating function to that group. In both the Python and C++ programming languages, for example, this could be written as `not (A or B)` for NOR and `not (A and B)` for NAND.

3.7 Digital numbers and codes

As previously mentioned, discrete ("on/off") signals are useful enough in isolation, such as in the water storage vessel example where each discrete signal had its own real-world meaning, but the utility of discrete signals rapidly expands once we gang multiple signals together and regard them as portions of a larger digital *word*. We will now explore some applications of this concept.

First, we may represent whole numbers using digital words comprised of multiple "bits". Examine the following table, where we list all possible combinations of a three-bit word and associate each one with a decimal number:

Three-bit digital word	Decimal equivalent
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

This use of digital words to represent numerical values is known as *binary* numeration. The term "binary" refers to the two-state nature of each bit. The pattern of three-bit combinations is not arbitrary in binary numeration, but follows a definite pattern. Just like each digit of a multi-digit decimal number has a different place-weight associated with it (i.e. a one's place, a ten's place, a hundreds place, etc.) each bit within a binary number bears a place-weight as well. The difference with binary is that the place-weight values scale by powers of two rather than powers of 10 as with decimal numeration. For the three-bit pattern shown in the table, the right-most bit is in the one's place, the middle bit is in the two's place, and the left-most bit is in the four's place. This makes sense if you take any of the binary numbers shown and add up all the place-weight values occupied by a 1 bit, and then compare that sum with the decimal equivalent value shown in the right-hand column. 011 is equivalent to 3 because there is a 1 in the two's place and a 1 in the one's place, so $2 + 1 = 3$. 110 is equivalent to 6 because there is a 1 in the four's place and a 1 in the two's place, so $4 + 2 = 6$.

It is possible to associate digital words with entities other than just whole numbers. Schemes exist to associate binary word patterns with integer[7] numbers, with alphabetical characters[8], with

[7] The *integer* number range covers all whole numbers as well as their negative counterparts. By definition, a *whole* number cannot be negative.

[8] The ASCII and Unicode standards are outstanding examples of this, with the ASCII (American Standard Code for Information Interchange) utilizing a seven-bit digital word to encode all characters commonly found on an American computer keyboard, and Unicode utilizing a twenty-one bit digital word to encode all characters found in all languages on Earth(!).

the position of a rotary shaft[9], with analog electrical signals[10], and even with musical sounds[11]. Digital encoding is really limitless in application, as anything you might be able to imagine that is capable of being subdivided into discrete states may be represented by a digital word.

3.8 Digital storage and communication

Once encoded, data represented by digital words may be recorded in any format capable of retaining binary on/off states. Many different media exist for this purpose, including punched mylar tape, magnetic tape, magnetic disks, optical disks, and semiconductor memory devices. In each case, the bits are manifest as some physical parameter that are either one state or another. For hole-punched tape the distinction is between a punched area or an unpunched area on the tape. For magnetic tape or disk, it is a small region of magnetized material which is either magnetized with one magnetic polarity (e.g. North "up") or the other. For optical media, it is a spot of different color or reflectivity that may be read by an optical sensor. For semiconductor memory, it may be the saturation state of a powered transistor, or the static energization state of a micro-capacitor element. The binary *all-or-nothing* nature of each bit's encoding guards against corruption by noise or natural decay, as it is much easier to discern the coarse 0 or 1 status of a bit than it is to accurately sense the fine value of an analog signal.

Communication of digital data is similarly flexible and surprisingly resistant to corruption by noise. Unlike analog signals which may be garbled rather easily by the addition of noise, on/off *pulses* are more robust and less ambiguous to interpret. We have a wide range of choice regarding media for communication just as we do for storage: wired electrical signals, wireless electromagnetic signals, optical signals, acoustic signals, and even mechanical motion are all possible modes of communicating digital words.

It is interesting to note that the first form of electrical communication was actually digital: the *telegraph*. On/off electrical pulse signals represented discrete characters printed on paper tape and/or heard audibly by a "sounder" relay device, sequences of which would then represent alphanumeric characters and whole words.

[9] A device known as an *absolute shaft encoder* performs this conversion from shaft angle to digital word. The shaft in question is typically attached to a wheel which has had a multi-bit digital code etched or cut into its surface. A set of stationary optical sensors detects this code, and those sensors pick up different bit combinations as the wheel rotates to different angular positions.

[10] A device known as an *analog-to-digital converter*, or *ADC* receives an analog voltage signal and outputs a corresponding digital code. Another device known as a *digital-to-analog converter* or *DAC* does the reverse.

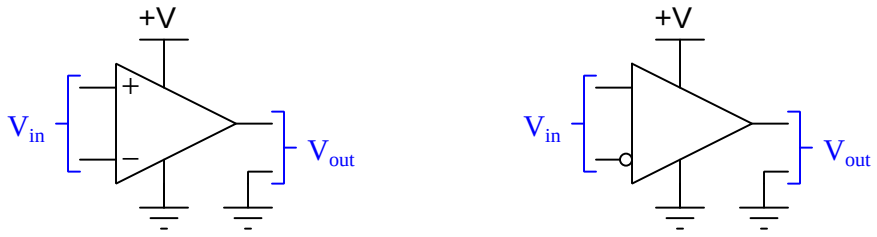
[11] This code is called *MIDI*, which stands for *Musical Instrument Digital Interface*. Each musical sound is encoded by a set of digital words capturing such parameters as *pitch* (called the "note number"), *velocity* (related to *attack* and *decay* rate of the sound), note on/off (signaling the start and end of a note), etc.

3.9 Comparators

Logic signals are discrete in nature: either "high" or "low". Analog signals span a continuous range of voltage values, usually established by the circuit's DC power supply voltage "rails". While logic gate circuits are strictly digital devices, another type of device called a *comparator* blends analog and digital signals in one package: a comparator inputs two analog voltage signals and outputs one digital signal, the state of that discrete output signal depending on which of the two analog voltage input signals is larger. Comparators deserve mention along with logic gates because comparators are often used to "condition" analog signals for input into logic gates.

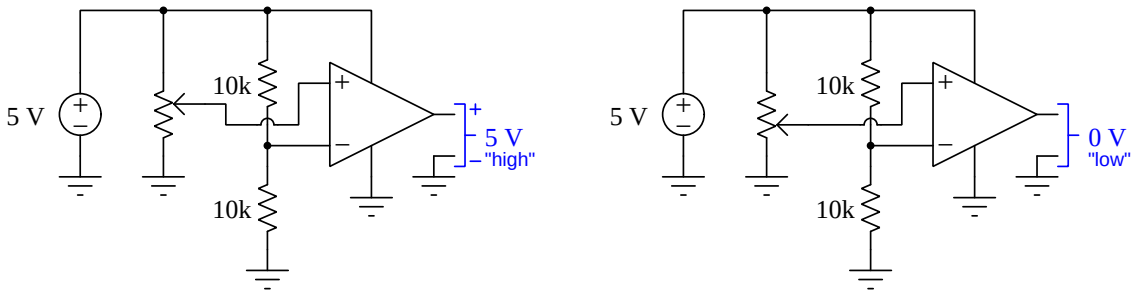
Comparators have two input terminals, marked "+" and "−" (non-inverting and inverting, respectively), one output terminal, and of course two terminals for connection to a DC power supply. Since a comparator's function is to *compare* the two ground-referenced input voltage signals, what it is really doing is measuring the difference in potential (i.e. differential voltage) between those two input terminals. The output's logical state depends strictly on the *polarity* of V_{in}:

Common comparator symbols



Note that in addition to two input terminals and one output terminal, a comparator also requires two additional terminals for its DC power supply, since these are "active" integrated circuits just like logic gate ICs.

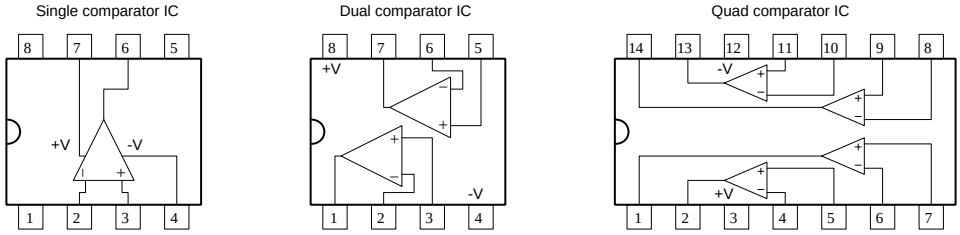
Comparator operation is fairly easy to describe. If the analog signal voltage between the "+" input and ground is larger than the analog voltage signal between the "−" input and ground, the comparator's digital output drives to a "high" state. If the analog signal magnitudes are opposite of this, the output goes "low". The following examples show this in action:



The two 10 kiloOhm resistors create a fixed-ratio voltage divider, providing a 2.5 Volt signal to the "−" input of the comparator with reference to ground. The potentiometer serves as a variable-ratio

voltage divider, providing a user-adjustable analog signal to the comparator's "+" input terminal with reference to ground. Whenever the potentiometer's wiper is moved to a position higher than the half-way point, the comparator deems that variable signal to be greater than the 2.5 Volt reference signal and drives the output high. Whenever the wiper gets moved below the half-way point on the potentiometer, the comparator deems that adjusted signal to be less than the 2.5 Volt reference and drives the output low.

Comparators, like semiconductor logic gates, exist as *integrated circuits* (ICs) with all internal components on a single wafer of silicon semiconductor material, often with more than one amplifier packaged in one IC "chip". Examples of single, dual, and quad comparators contained within *DIP* (Dual Inline Package) integrated circuits appear in the following pinout diagrams:



Note the pin-numbering convention common to DIP integrated circuits: with the indexing notch on the left-hand side of the IC, the lower-left pin will be number 1, with pin numbers increasing as you count in a counter-clockwise direction.

Chapter 4

Historical References

This chapter is where you will find references to historical texts and technologies related to the module's topic.

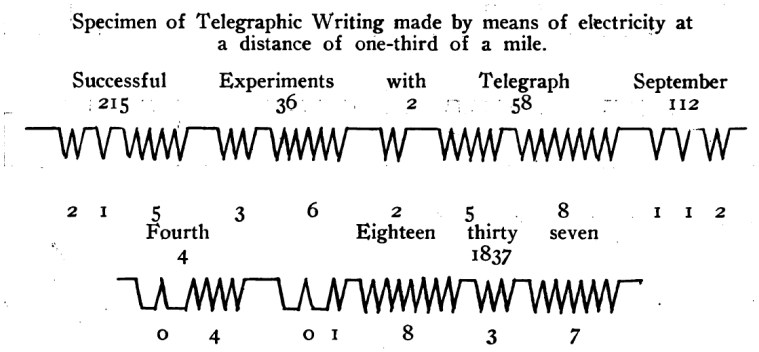
Readers may wonder why historical references might be included in any modern lesson on a subject. Why dwell on old ideas and obsolete technologies? One answer to this question is that the initial discoveries and early applications of scientific principles typically present those principles in forms that are unusually easy to grasp. Anyone who first discovers a new principle must necessarily do so from a perspective of ignorance (i.e. if you truly *discover* something yourself, it means you must have come to that discovery with no prior knowledge of it and no hints from others knowledgeable in it), and in so doing the discoverer lacks any hindsight or advantage that might have otherwise come from a more advanced perspective. Thus, discoverers are forced to think and express themselves in less-advanced terms, and this often makes their explanations more readily accessible to others who, like the discoverer, comes to this idea with no prior knowledge. Furthermore, early discoverers often faced the daunting challenge of explaining their new and complex ideas to a naturally skeptical scientific community, and this pressure incentivized clear and compelling communication. As James Clerk Maxwell eloquently stated in the Preface to his book *A Treatise on Electricity and Magnetism* written in 1873,

> It is of great advantage to the student of any subject to read the original memoirs on that subject, for science is always most completely assimilated when it is in its nascent state . . . `[page xi]`

Furthermore, grasping the historical context of technological discoveries is important for understanding how science intersects with culture and civilization, which is ever important because new discoveries and new applications of existing discoveries will always continue to impact our lives. One will often find themselves impressed by the ingenuity of previous generations, and by the high degree of refinement to which now-obsolete technologies were once raised. There is much to learn and much inspiration to be drawn from the technological past, and to the inquisitive mind these historical references are treasures waiting to be (re)-discovered.

4.1 The original telegraph code

Samuel Morse, inventor of the telegraph, actually did not use what is known today as *Morse Code* to transmit his first messages. Instead, his original method of encoding was based on decimal numbers, with specific words represented by unique numbers. Page 29 of the book *Early History of the Electro-Magnetic Telegraph* by Alfred Vail documents a typical message communicated using Morse's system:



The *words* in the diagram were the intelligence transmitted.

The *numbers* (in this instance arbitrary) are the numbers of the words in a telegraphic dictionary.

The *points* are the markings of the register, each point being marked every time the Electric fluid passes.

The register marks but one kind of mark to wit, (V) this can be varied two ways. By intervals, thus, (V VV VVV), signifying one, two, three, etc., and by reversing, thus, (Λ). Examples of both these varieties are seen in the diagram.

The single numbers are separated by *short,* and the whole numbers by *long intervals.*

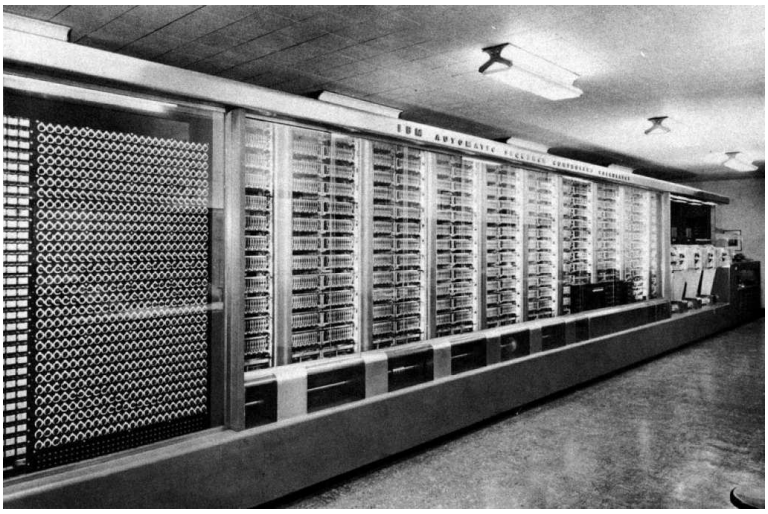
The jagged lines form an oscillograph of the telegraph circuit's digital state over time, and as you can see it used groups of successive pulses to count digits of decimal numbers. For example, the number 215 was encoded as two pulses followed by a brief rest, then one pulse followed by another rest, then a sequence of five pulses. A longer rest separated individual numbers from each other, as opposed to the shorter rests which merely separated digits of the same number from each other.

A "telegraphic dictionary" served to cross-reference common words with number values. As you can see from the example shown, the number 215 represented the word *Successful*, while 36 represented *Experiments*, 2 represented *With*, and so on. One can only imagine how many numbers would have been required to represent even a rudimentary cross-section of the English vocabulary!

4.2 Punched paper tape

On the 7th of August 1944 the International Business Machines Corporation (IBM) presented an electromechanical computer called the *IBM Automatic Sequence Controlled Calculator* to Harvard University. This room-size calculating machine used thousands of electromechanical relays, switches, paper tape reels, motors, and other devices to perform mathematical calculations. It was used by the United States Navy for classified work over a period of many years.

A photograph showing the front panel of this massive[1] machine appears next, taken from the Historical Introduction chapter of *A Manual of Operation for the Automatic Sequence Controlled Calculator* published by Harvard University Press in 1946:



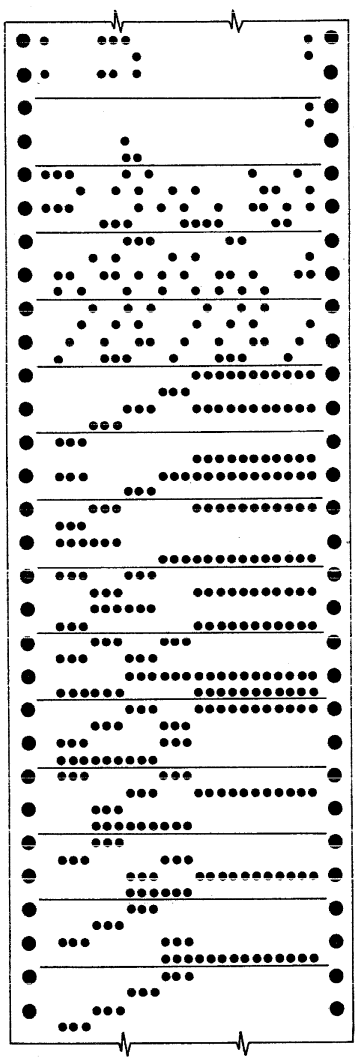
Like other programmable computers of its era, this machine received instructions encoded in the form of punched paper cards and punched paper tape. Holes punched in specific places on paper represented numerical values and instructions for the computer to process. A set of metal pins arrayed on one side of the paper detected the presence of holes by protruding through to make electrical contact with metal pieces on the other side of the paper. The cards and tape were advanced through the reading portions of the machine using cogged wheels, gears, clutches, and electromechanical solenoids.

Although this computer did not use *binary* notation for numerical values as we know binary today[2], it still used discrete hole/no-hole markings to encode the numbers and functional commands and so was entirely digital in nature. Specific hole patterns represented decimal digits, with each column on the tape reserved for one digit of a decimal number.

[1] A quote from page 11 of the *Manual* states that the machine measured 8 feet high, 51 feet long, and weighed approximately 5 tons.

[2] Using place-weighted values corresponding to powers of two.

An illustration of one of the paper tapes (called the *value tape*, used to encode numerical input values for the machine to process) is shown here (taken from page 46 of the *Manual*) alongside comments explaining the meanings of the codes:



VALUE TAPE

Negative Argument -51
Argument Code in 1st col.

Positive Argument 51
Argument Code in 1st col.

Negative Number
-698 321 576 438 499 013 451

Positive Number
521 328 794 532 605 972 100 28

Diagonal Number
012 345 678 901 234 567 890 123

0 000 111 222 333 999 999 999 99

0 999 000 111 222 888 888 888 88

0 888 999 000 111 777 777 777 77

0 777 888 999 000 666 666 666 66

0 666 777 888 999 555 555 555 55

0 555 666 777 888 444 444 444 44

0 444 555 666 777 333 333 333 33

0 333 444 555 666 222 222 222 22

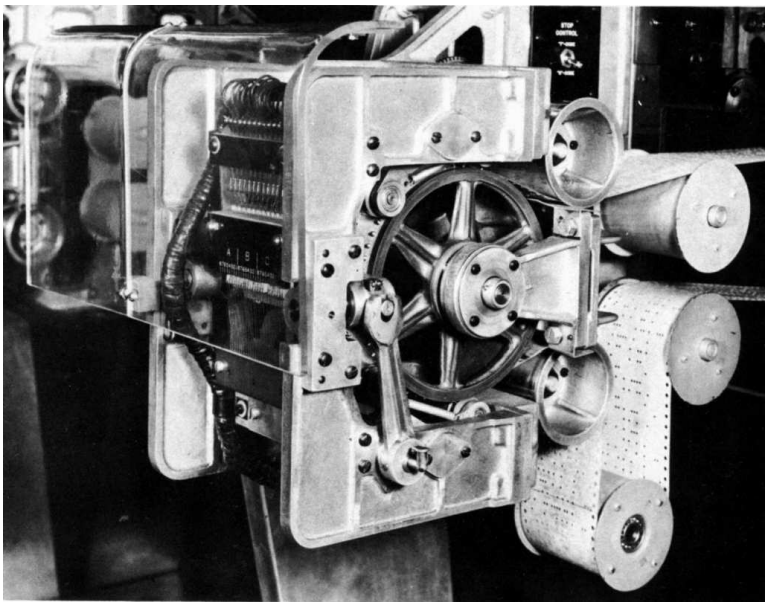
0 222 333 444 555 111 111 111 11

0 111 222 333 444 000 000 000 00

Figure 13

A careful inspection of this example illustration and its comments will reveal just how this computer encoded decimal digits using punched holes. The example numbers shown are intended to make this process relatively easy, and I recommend taking the time to discern this pattern. Note: the large holes found on either edge of the paper strip are not part of the code, but rather serve as indexing holes for the motor-driven wheels to engage with and hold the paper tape.

A photograph showing one of the tape-reading mechanisms called the *Sequence Control Mechanism* appears in this next photograph from the *Manual*. In this photograph you can see some of the punched tape woven through the mechanism, guided and advanced by a series of motor-driven wheels and idler pulleys:



4.3 Claude Shannon makes the connection

One of the great minds of electrical engineering, Claude Shannon, recognized the application of Boolean logic to relay switching circuits while studying for his Master's degree at Massachusetts Institute of Technology (MIT) in 1940. His thesis paper submitted as part of the graduation requirements for this degree has been widely hailed as a breakthrough for the analysis of digital circuits.

On pages 2 and 3 of Shannon's paper, we read the following paragraph where he states an equivalence between propositional logic and electrical switching:

> The method of solution of these problems which will be developed here may be described briefly as follows: Any circuit is represented by a set of equations, the terms of the equations representing the various relays and switches of the circuit. A calculus is developed for manipulating these equations by simple mathematical processes, most of which are similar to ordinary algebraic [sic] algorisms. This calculus is shown to be exactly analogous to the Calculus of Propositions used in the symbolic study of logic. For the synthesis problem the desired characteristics are first written as a system of equations, and the equations are then manipulated into the form representing the simplest circuit. The circuit may then be immediately drawn from the equations. By this method it is always possible to find the simplest circuit containing only series and parallel connections, `[page 2]`
>
> and for certain types of functions it is possible to find the simplest circuit containing any type of connection. In the analysis problem the equations representing the given circuit are written and may then be interpreted in terms of the operating characteristics of the circuit. It is also possible with the calculus to obtain any number of circuits equivalent to a given circuit. `[page 3]`

Note his express intent of using Boolean algebra as a tool for circuit minimization. In his era, when logical functions required electromechanical relays to implement, the minimization of relay coils and contacts meant a logic circuit with fewer components, translating into lower cost of manufacture and greater reliability. When solid-state (semiconductor) transistor-based logic circuits appeared, this same Boolean algebra proved useful as a tool for determining the simplest (and therefore most compact) circuit that could be etched into a silicon chip.

Shannon's mathematical approach to switching circuits was to consider an open circuit to have a value of 1, and a closed circuit to have a value of 0. He referred to this as the *hinderance* of the switching network, similar in concept to *resistance* but discrete in nature rather than continuous. This is inverse of how we now consider most switching circuits in Boolean form, energized (closed) being 1 and de-energized (open) being 0. Based on his definition of 0 and 1 for switching circuits, Shannon then reasoned that series switch networks would be represented by the Boolean addition of their hinderance values and that parallel switch networks would be represented by the multiplication of their hinderances. Again, this seems backwards from our modern perspective where we relate series-connected switch contacts to the logical AND function (Boolean multiplication) and parallel contacts to the OR function (addition), but it is important to realize that either approach is mathematically valid. As with any other application of axiomatic reasoning, the outcomes depend greatly upon one's initial definitions.

Shannon then proceeds in his paper to present a set of postulates relating 0 and 1 values to open and closed switching networks (pages 5 and 6):

Boolean expression	**Electrical interpretation**
$0 \cdot 0 = 0$	A closed circuit in parallel with a closed circuit is a closed circuit
$1 + 1 = 1$	An open circuit in series with an open circuit is an open circuit
$1 + 0 = 0 + 1 = 1$	An open circuit in series with a closed circuit in either order is an open circuit
$0 \cdot 1 = 1 \cdot 0 = 0$	A closed circuit in parallel with an open circuit in either order is a closed circuit
$0 + 0 = 0$	A closed circuit in series with a closed circuit is a closed circuit
$1 \cdot 1 = 1$	An open circuit in parallel with an open circuit is an open circuit

Rather than use bar-lines over Boolean variables to denote inversion or negation, Shannon opted to use apostrophe ("prime") characters. So, rather than print the inversion of X as $\overline{X}$, Shannon writes X'. If groups of variables are inverted (e.g. $\overline{A + B}$), one must use parentheses to group the variables together and then follow the closing parentheses symbol with an apostrophe, for example $(A + B)'$. This choice of symbols may very well have been a result of the crude typesetting available in Shannon's time, apostrophes being much easier to properly typeset on a page than bar-lines.

On page 11 of his paper, Shannon neatly summarizes and compares relay switching logic with Propositional Logic:

Symbol	**Relay interpretation**	**Propositional interpretation**
X	The circuit X	The proposition X
0	The circuit is closed	The proposition is false
1	The circuit is open	The proposition is true
$X + Y$	The series connection of circuits X and Y	The proposition which is true if either X or Y is true
XY	The parallel connection of circuits X and Y	The proposition which is true if both X and Y are true
X'	The circuit which is open when X is closed, and closed when X is open	The contradictory proposition X
$=$	The circuits open and close simultaneously	Each proposition implies the other

Again, recall that Shannon assumed an electrical open to be a 1 and a short to be a 0, which is why series is equivalent to addition and parallel to multiplication in his application of Boolean algebra. This is opposite of how modern relay logic circuits are typically interpreted, with an energized load (i.e. a closed switch allowing current) being 1 and a de-energized load (i.e. an open switch preventing current) being 0.

Chapter 5

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

5.1 Process switch types and symbols

A "process" switch is one actuated by some physical parameter that is not human. A pushbutton switch or a toggle switch would *not* be a process switch, but a fluid pressure switch, a temperature switch, a fluid flow switch, etc., would all be examples of process switches.

As with all other types of electrical switches, the "normal" state of a process switch is a state of minimum or no stimulus. The illustration on the following page shows schematic diagram symbols for several different types of process switch, along with their open/closed states depending on each switch's "normal" type and the applied stimulus.

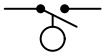
NO pressure switch

Open when there is zero pressure (minimum stimulus)
Closed when pressure increases beyond threshold

NC pressure switch

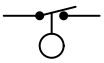
Closed when there is zero pressure (minimum stimulus)
Open when pressure increases beyond threshold

NO level switch



Open when there is zero level (minimum stimulus)
Closed when level increases beyond threshold

NC level switch



Closed when there is zero level (minimum stimulus)
Open when level increases beyond threshold

NO temperature switch

Open when temperature is cold (minimum stimulus)
Closed when temperature increases beyond threshold

NC temperature switch

Closed when temperature is cold (minimum stimulus)
Open when temperature increases beyond threshold

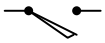
NO flow switch

Open when there is zero flow (minimum stimulus)
Closed when flow increases beyond threshold

NC flow switch

Closed when there is zero flow (minimum stimulus)
Open when flow increases beyond threshold

NO limit switch



Open when nothing touches (minimum stimulus)
Closed when physical contact made

NC limit switch

Closed when nothing touches (minimum stimulus)
Open when physical contact made

NO proximity switch

Open when nothing is near (minimum stimulus)
Closed with object is near

NC proximity switch

Closed when nothing is near (minimum stimulus)
Open when object is near

Chapter 6

Programming References

A powerful tool for mathematical modeling is text-based *computer programming*. This is where you type coded commands in text form which the computer is able to interpret. Many different text-based languages exist for this purpose, but we will focus here on just two of them, *C++* and *Python*.

6.1 Programming in C++

One of the more popular text-based computer programming languages is called *C++*. This is a *compiled* language, which means you must create a plain-text file containing C++ code using a program called a *text editor*, then execute a software application called a *compiler* to translate your "source code" into instructions directly understandable to the computer. Here is an example of "source code" for a very simple C++ program intended to perform some basic arithmetic operations and print the results to the computer's console:

```
#include <iostream>
using namespace std;

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

  cout << "This simple program performs basic arithmetic on" << endl;
  cout << "the two numbers " << x << " and " << y << " and then" << endl;
  cout << "displays the results on the computer's console." << endl;

  cout << endl;

  cout << "Sum = " << x + y << endl;
  cout << "Difference = " << x - y << endl;
  cout << "Product = " << x * y << endl;
  cout << "Quotient of " << x / y << endl;

  return 0;
}
```

Computer languages such as C++ are designed to make sense when read by human programmers. The general order of execution is left-to-right, top-to-bottom just the same as reading any text document written in English. Blank lines, indentation, and other "whitespace" is largely irrelevant in C++ code, and is included only to make the code more pleasing[1] to view.

[1] Although not included in this example, *comments* preceded by double-forward slash characters (`//`) may be added to source code as well to provide explanations of what the code is supposed to do, for the benefit of anyone reading it. The compiler application will ignore all comments.

Let's examine the C++ source code to explain what it means:

- `#include <iostream>` and `using namespace std;` are set-up instructions to the compiler giving it some context in which to interpret your code. The code specific to your task is located between the brace symbols ({ and }, often referred to as "curly-braces").

- `int main (void)` labels the "Main" function for the computer: the instructions within this function (lying between the { and } symbols) it will be commanded to execute. Every complete C++ program contains a `main` function at minimum, and often additional functions as well, but the `main` function is where execution always begins. The `int` declares this function will return an *integer* number value when complete, which helps to explain the purpose of the `return 0;` statement at the end of the `main` function: providing a numerical value of zero at the program's completion as promised by `int`. This returned value is rather incidental to our purpose here, but it is fairly standard practice in C++ programming.

- Grouping symbols such as (parentheses) and {braces} abound in C, C++, and other languages (e.g. Java). Parentheses typically group data to be processed by a function, called *arguments* to that function. Braces surround lines of executable code belonging to a particular function.

- The `float` declaration reserves places in the computer's memory for two *floating-point* variables, in this case the variables' names being `x` and `y`. In most text-based programming languages, variables may be named by single letters or by combinations of letters (e.g. `xyz` would be a single variable).

- The next two lines assign numerical values to the two variables. Note how each line terminates with a semicolon character (`;`) and how this pattern holds true for most of the lines in this program. In C++ semicolons are analogous to periods at the ends of English sentences. This demarcation of each line's end is necessary because C++ ignores whitespace on the page and doesn't "know" otherwise where one line ends and another begins.

- All the other instructions take the form of a `cout` command which prints characters to the "standard output" stream of the computer, which in this case will be text displayed on the console. The double-less-than symbols (`<<`) show data being sent *toward* the `cout` command. Note how verbatim text is enclosed in quotation marks, while variables such as `x` or mathematical expressions such as `x - y` are not enclosed in quotations because we want the computer to display the numerical values represented, not the literal text.

- Standard arithmetic operations (add, subtract, multiply, divide) are represented as `+`, `-`, `*`, and `/`, respectively.

- The `endl` found at the end of every `cout` statement marks the end of a line of text printed to the computer's console display. If not for these `endl` inclusions, the displayed text would resemble a run-on sentence rather than a paragraph. Note the `cout << endl;` line, which does nothing but create a blank line on the screen, for no reason other than esthetics.

After saving this *source code* text to a file with its own name (e.g. `myprogram.cpp`), you would then *compile* the source code into an *executable* file which the computer may then run. If you are using a console-based compiler such as *GCC* (very popular within variants of the Unix operating system[2], such as Linux and Apple's OS X), you would type the following command and press the Enter key:

```
g++ -o myprogram.exe myprogram.cpp
```

This command instructs the `GCC` compiler to take your source code (`myprogram.cpp`) and create with it an executable file named `myprogram.exe`. Simply typing `./myprogram.exe` at the command-line will then execute your program:

```
./myprogram.exe
```

If you are using a graphic-based C++ development system such as Microsoft Visual Studio[3], you may simply create a new console application "project" using this software, then paste or type your code into the example template appearing in the editor window, and finally run your application to test its output.

As this program runs, it displays the following text to the console:

```
This simple program performs basic arithmetic on
the two numbers 200 and -560.5 and then
displays the results on the computer's console.

Sum = -360.5
Difference = 760.5
Product = -112100
Quotient of -0.356824
```

As crude as this example program is, it serves the purpose of showing how easy it is to write and execute simple programs in a computer using the C++ language. As you encounter C++ example programs (shown as source code) in any of these modules, feel free to directly copy-and-paste the source code text into a text editor's screen, then follow the rest of the instructions given here (i.e. save to a file, compile, and finally run your program). You will find that it is generally easier to

[2] A very functional option for users of Microsoft Windows is called *Cygwin*, which provides a Unix-like console environment complete with all the customary utility applications such as GCC!

[3] Using Microsoft Visual Studio community version 2017 at the time of this writing to test this example, here are the steps I needed to follow in order to successfully compile and run a simple program such as this: (1) Start up Visual Studio and select the option to create a New Project; (2) Select the Windows Console Application template, as this will perform necessary set-up steps to generate a console-based program which will save you time and effort as well as avoid simple errors of omission; (3) When the editing screen appears, type or paste the C++ code within the `main()` function provided in the template, deleting the "Hello World" `cout` line that came with the template; (4) Type or paste any preprocessor directives (e.g. `#include` statements, `namespace` statements) necessary for your code that did not come with the template; (5) Lastly, under the Debug drop-down menu choose either Start Debugging (F5 hot-key) or Start Without Debugging (Ctrl-F5 hotkeys) to compile ("Build") and run your new program. Upon execution a console window will appear showing the output of your program.

learn computer programming by closely examining others' example programs and modifying them than it is to write your own programs starting from a blank screen.

6.2 Programming in Python

Another text-based computer programming language called *Python* allows you to type instructions at a terminal prompt and receive immediate results without having to compile that code. This is because Python is an *interpreted* language: a software application called an *interpreter* reads your source code, translates it into computer-understandable instructions, and then executes those instructions in one step.

The following shows what happens on my personal computer when I start up the Python interpreter on my personal computer, by typing `python3`[4] and pressing the Enter key:

```
Python 3.7.2 (default, Feb 19 2019, 18:15:18)
[GCC 4.1.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` symbols represent the prompt within the Python interpreter "shell", signifying readiness to accept Python commands entered by the user.

Shown here is an example of the same arithmetic operations performed on the same quantities, using a Python interpreter. All lines shown preceded by the `>>>` prompt are entries typed by the human programmer, and all lines shown without the `>>>` prompt are responses from the Python interpreter software:

```
>>> x = 200
>>> y = -560.5
>>> x + y
-360.5
>>> x - y
760.5
>>> x * y
-112100.0
>>> x / y
-0.35682426404995538
>>> quit()
```

[4] Using version 3 of Python, which is the latest at the time of this writing.

More advanced mathematical functions are accessible in Python by first entering the line `from math import *` which "imports" these functions from Python's math *library* (with functions identical to those available for the C programming language, and included on any computer with Python installed). Some examples show some of these functions in use, demonstrating how the Python interpreter may be used as a scientific calculator:

```
>>> from math import *
>>> sin(30.0)
-0.98803162409286183
>>> sin(radians(30.0))
0.49999999999999994
>>> pow(2.0, 5.0)
32.0
>>> log10(10000.0)
4.0
>>> e
2.7182818284590451
>>> pi
3.1415926535897931
>>> log(pow(e,6.0))
6.0
>>> asin(0.7071068)
0.78539819000368838
>>> degrees(asin(0.7071068))
45.000001524425265
>>> quit()
```

Note how trigonometric functions assume angles expressed in *radians* rather than *degrees*, and how Python provides convenient functions for translating between the two. Logarithms assume a base of e unless otherwise stated (e.g. the `log10` function for common logarithms).

The interpreted (versus compiled) nature of Python, as well as its relatively simple syntax, makes it a good choice as a person's first programming language. For complex applications, interpreted languages such as Python execute slower than compiled languages such as C++, but for the very simple examples used in these learning modules speed is not a concern.

Another Python math library is `cmath`, giving Python the ability to perform arithmetic on complex numbers. This is very useful for AC circuit analysis using *phasors*[5] as shown in the following example. Here we see Python's interpreter used as a scientific calculator to show series and parallel impedances of a resistor, capacitor, and inductor in a 60 Hz AC circuit:

```
>>> from math import *
>>> from cmath import *
>>> r = complex(400,0)
>>> f = 60.0
>>> xc = 1/(2 * pi * f * 4.7e-6)
>>> zc = complex(0,-xc)
>>> xl = 2 * pi * f * 1.0
>>> zl = complex(0,xl)
>>> r + zc + zl
(400-187.38811239154882j)
>>> 1/(1/r + 1/zc + 1/zl)
(355.837695813625+125.35793777619385j)
>>> polar(r + zc + zl)
(441.717448903332, -0.4381072059213295)
>>> abs(r + zc + zl)
441.717448903332
>>> phase(r + zc + zl)
-0.4381072059213295
>>> degrees(phase(r + zc + zl))
-25.10169387356105
```

When entering a value in rectangular form, we use the `complex()` function where the arguments are the real and imaginary quantities, respectively. If we had opted to enter the impedance values in polar form, we would have used the `rect()` function where the first argument is the magnitude and the second argument is the angle in radians. For example, we could have set the capacitor's impedance (`zc`) as $X_C \angle -90^o$ with the command `zc = rect(xc,radians(-90))` rather than with the command `zc = complex(0,-xc)` and it would have worked the same.

Note how Python defaults to rectangular form for complex quantities. Here we defined a 400 Ohm resistance as a complex value in rectangular form (400 +j0 Ω), then computed capacitive and inductive reactances at 60 Hz and defined each of those as complex (phasor) values ($0 - jX_c$ Ω and $0 + jX_l$ Ω, respectively). After that we computed total impedance in series, then total impedance in parallel. Polar-form representation was then shown for the series impedance (441.717 Ω $\angle$ -25.102^o). Note the use of different functions to show the polar-form series impedance value: `polar()` takes the complex quantity and returns its polar magnitude and phase angle in *radians*; `abs()` returns just the polar magnitude; `phase()` returns just the polar angle, once again in radians. To find the polar phase angle in degrees, we nest the `degrees()` and `phase()` functions together.

The utility of Python's interpreter environment as a scientific calculator should be clear from these examples. Not only does it offer a powerful array of mathematical functions, but also unlimited

[5] A "phasor" is a voltage, current, or impedance represented as a complex number, either in rectangular or polar form.

assignment of variables as well as a convenient text record[6] of all calculations performed which may be easily copied and pasted into a text document for archival.

It is also possible to save a set of Python commands to a text file using a text editor application, and then instruct the Python interpreter to execute it at once rather than having to type it line-by-line in the interpreter's shell. For example, consider the following Python program, saved under the filename `myprogram.py`:

```
x = 200
y = -560.5

print("Sum")
print(x + y)

print("Difference")
print(x - y)

print("Product")
print(x * y)

print("Quotient")
print(x / y)
```

As with C++, the interpreter will read this source code from left-to-right, top-to-bottom, just the same as you or I would read a document written in English. Interestingly, whitespace *is* significant in the Python language (unlike C++), but this simple example program makes no use of that.

To execute this Python program, I would need to type `python myprogram.py` and then press the Enter key at my computer console's prompt, at which point it would display the following result:

```
Sum
-360.5
Difference
760.5
Product
-112100.0
Quotient
-0.35682426405
```

As you can see, syntax within the Python programming language is simpler than C++, which is one reason why it is often a preferred language for beginning programmers.

[6] Like many command-line computing environments, Python's interpreter supports "up-arrow" recall of previous entries. This allows quick recall of previously typed commands for editing and re-evaluation.

If you are interested in learning more about computer programming in *any* language, you will find a wide variety of books and free tutorials available on those subjects. Otherwise, feel free to learn by the examples presented in these modules.

6.3 Modeling Boolean logic using C++

Here is an example C++ program intended to demonstrate Boolean logic operations, using the `bool` data type defined in the C++ language:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;

  A = 0;
  B = 1;

  cout << "A  or  B = " << (A || B)  << endl;
  cout << "A nor  B = " << !(A || B) << endl;
  cout << "A and  B = " << (A && B)  << endl;
  cout << "A nand B = " << !(A && B) << endl;

  return 0;
}
```

When compiled and executed, this program generates the following output:

```
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
```

As we explore the operation of this program, we will discuss the following programming principles along the way:

- Preprocessor directives, namespaces
- The `main` function
- Variable types (`bool`, `int`) and declarations
- Variable assignment/initialization (`=`)
- Printing text output (`cout`, `<<`)
- Boolean operators (`!`, `||`, `&&`, `or`, `and`, `not`)
- Accepting user input (`cin`, `>>`)
- Loops (`while`)
- Comparison (`<=`)
- Increment operator (`++`)

At the start of the code listing we see the `#include` and `namespace` directives instructing the compiler how to interpret later instructions such as `cout`. The `main` function encapsulates all the code we wish to run, and presents it to the compiler software as the first point of execution for the compiled program. Variables `A` and `B` are declared to be of the `bool` type, which means they can only possess one of two values, either *true* or *false*; 1 or 0. `A` and `B` are assigned values, then four `cout` lines command the computer to display text to its console: the text between quotation marks being printed verbatim and mathematical expressions printed as their logical values. Note how addition is equivalent to a logical OR, multiplication equivalent to a logic AND, and an exclamation point preceding an expression equivalent to logical inversion. `endl` control characters force a line of text to end and a new line of text to begin.

Another way to perform Boolean arithmetic in C++ is to use the words `or` and `and` and `not`. This next program demonstrates:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;

  A = 0;
  B = 1;

  cout << "A  or  B = " << (A or B)     << endl;
  cout << "A nor  B = " << not(A or B)  << endl;
  cout << "A and  B = " << (A and B)    << endl;
  cout << "A nand B = " << not(A and B) << endl;

  return 0;
}
```

Once again, when compiled and executed this program generates the exact same output:

```
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
```

So far all we've really shown is how to perform Boolean arithmetic two different ways in C++, by logical operators (`||`, `&&`, `!`) and by word-operators (`or`, `and`, `not`). None of this is particularly challenging, and so we will use this opportunity to explore some more advanced C++ programming concepts.

One way to make our simple C++ program more interesting is provide a way for the user to input different values for `A` and `B` while the program is running, instead of only being able to assign values to `A` and `B` within the source code. The `iostream` library in C++ provides an instruction to do this called `cin`, which is a counterpart to `cout`. The following usage of `cin` is fairly self-explanatory:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;

  cin >> A;
  cin >> B;

  cout << "A  or  B = " << (A or B)     << endl;
  cout << "A nor  B = " << not(A or B)  << endl;
  cout << "A and  B = " << (A and B)    << endl;
  cout << "A nand B = " << not(A and B) << endl;

  return 0;
}
```

Instead of the "*put to*" symbol (`<<`) used with the `cout` instruction, the `cin` instruction requires a "*get from*" symbol which points the other direction (`>>`).

When compiled and executed this program waits for you to enter two number values (pressing Enter after each entry) before displaying the results. In this case, I entered 0 for `A` and 1 for `B`:

```
0
1
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
```

Running the program repeatedly, I can enter different values for `A` and `B` and receive results corresponding to those input states:

```
1
1
A  or  B = 1
A nor  B = 0
A and  B = 1
A nand B = 0


0
0
A  or  B = 0
A nor  B = 1
A and  B = 0
A nand B = 1
```

If we wish our program be more user-friendly, we might precede each `cin` with a `cout` providing some prompting text for the user to read:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;

  cout << "Enter value for A ";
  cin >> A;
  cout << "Enter value for B ";
  cin >> B;

  cout << "A  or  B = " << (A or B)     << endl;
  cout << "A nor  B = " << not(A or B)  << endl;
  cout << "A and  B = " << (A and B)    << endl;
  cout << "A nand B = " << not(A and B) << endl;

  return 0;
}
```

Here I show this program's output with a value of 1 entered for `A` and a value of 0 entered for `B`:

```
Enter value for A 1
Enter value for B 0
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
```

Another useful feature we could add to our program is to cause it to *loop* so that the user gets more than one opportunity to test combinations of values for `A` and `B`. An easy way to add this functionality to our program is to use something called a `while` loop, shown in the following code listing:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;

  while (1)
  {
    cout << "Enter value for A ";
    cin >> A;
    cout << "Enter value for B ";
    cin >> B;

    cout << "A  or  B = " << (A or B)     << endl;
    cout << "A nor  B = " << not(A or B)  << endl;
    cout << "A and  B = " << (A and B)    << endl;
    cout << "A nand B = " << not(A and B) << endl;
  }

  return 0;
}
```

The `while` statement has a set of parentheses for accepting an *argument*. All code encompassed within the `while`'s braces will repeatedly execute so long as the logical state of the `while` statement's argument is "true". Since we have made the `while` statement's argument equal to `1`, that condition is always satisfied which means the loop will repeat indefinitely.

Note how the variable declaration (`bool A, B;`) and the `return 0` lines are not included within the `while` loop's braces. This is intentional. In the case of the declaration, there is literally no point in *re-declaring* memory space for the boolean variables `A` and `B`. Once is enough, after which we are free to re-initialize those variables with new values as many times as we wish. In the case of the `return` statement, it is important we leave this out of the `while` loop or else the loop would only execute once! In C and C++, the act of *returning* forces an exit from the current function.

This program contains no provision for halting the `while` loop, and so our best option for exiting the program is to press <*Ctrl-C*> which is a standard "abort" command for console applications such as this.

Here I show this "looping" program execute four times, giving me opportunity to try all four combinations of 0 and 1 states for `A` and `B`. You can see where I terminated the program using a <Ctrl-C> keystroke combination, as the console shows `^C`:

```
Enter value for A 0
Enter value for B 0
A  or  B = 0
A nor  B = 1
A and  B = 0
A nand B = 1
Enter value for A 0
Enter value for B 1
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
Enter value for A 1
Enter value for B 0
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
Enter value for A 1
Enter value for B 1
A  or  B = 1
A nor  B = 0
A and  B = 1
A nand B = 0
Enter value for A ^C
```

Another type of loop offered by C++ is the `for` loop, which limits repeats based on the range of values explored by a variable specified in the `for` statement's argument. Shown here is another version of the program, using a `for` loop to limit execution to *three* times:

```
#include <iostream>
using namespace std;

int main (void)
{
  bool A, B;
  int n;

  for (n = 1 ; n <= 3 ; ++n)
  {
    cout << "Enter value for A ";
    cin >> A;
    cout << "Enter value for B ";
    cin >> B;

    cout << "A  or  B = " << (A or B)     << endl;
    cout << "A nor  B = " << not(A or B)  << endl;
    cout << "A and  B = " << (A and B)    << endl;
    cout << "A nand B = " << not(A and B) << endl;
  }

  return 0;
}
```

`for` loops require a variable for control, and so we had to add another declaration line, this time for an *integer* variable named `n`. Note how the `for` statement has three different sections within its argument: a starting value for `n`, a condition for `n` allowing continued looping, and a statement to increment the value of `n` by one (`++n`). Here, we initialize the value of `n` to one, then check to see its value is less than or equal to three before looping.

When run, we see the program allow exactly three iterations, after which it cleanly exits on its own with no need for a manual interrupt (<Ctrl-C>):

```
Enter value for A 0
Enter value for B 0
A  or  B = 0
A nor  B = 1
A and  B = 0
A nand B = 1
Enter value for A 1
Enter value for B 1
A  or  B = 1
A nor  B = 0
A and  B = 1
A nand B = 0
Enter value for A 0
Enter value for B 1
A  or  B = 1
A nor  B = 0
A and  B = 0
A nand B = 1
```

Chapter 7

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

7.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

7.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

7.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Digital signal

Analog signal

Discrete signal

Logic state

Process switch

Proximity switch

Bit

Word

Logic function

OR function

AND function

Truth table

NOT function

NOR function

NAND function

XOR function

Boolean algebra

Complementation

Combinational function

Binary number

Place-weight

Digital encoding

Digital medium (or media, plural)

Noise immunity

7.1.3 Analog versus digital quantities

Identify the following quantities as being either continuous (i.e. analog) or discontinuous (digital):

- Continuity of an electrical switch:
- Resistance of a rheostat:
- Quantity of money in a billfold (bills and coins):
- A person's weight, in pounds or kilograms:
- Number of pebbles held in a hand:
- Status of a traffic light:
- Direction indicated by a magnetic compass:
- Quantity of water stored in a vessel:
- Run/stop status of an electric motor:
- Temperature of a furnace:
- Angular position of a shaft:
- Status of a water drain valve:

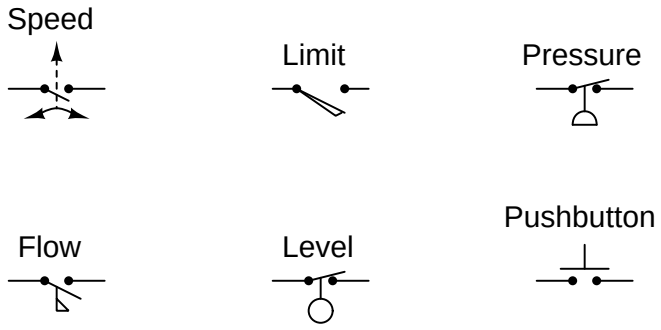
Challenges

- Explain how something inherently analog may be used to represent a digital quantity or state.
- Explain how something inherently digital may be used to represent an analog quantity.

7.1.4 Switch states

The "normal" status of an electrical switch is always the state in which its schematic symbol is drawn, being defined by the manufacturer of that switch. This is the state of the switch when it is physically at rest, with little or no stimulation.

Identify the "normal" status of each switch shown below, whether it is *normally-open* (NO) or *normally-closed* (NC). Then, identify what stimulation would be required to maintain each switch in the *closed* state.



Challenges

- Suppose the speed switch was coupled to an electric motor shaft, turning a water pump that was necessary to supply water pressure to a town. This pump is critically important, and runs all the time. Would this switch, installed in this application, be called *normally-open* or *normally-closed*?

7.1.5 Designing switch/lamp circuits

In each of the following circuit examples, sketch the schematic diagram showing how to connect a voltage source, switches, and a lamp to fulfill the given truth table.

Circuit #1 – design and sketch a circuit using a pressure switch to control a lamp:

Applied pressure	**Lamp status**
Less than 30 PSI	off
More than 30 PSI	ON

Circuit #2 – design and sketch a circuit using a temperature switch and a pushbutton switch to control a lamp:

Temperature	**Pushbutton status**	**Lamp status**
Below 100 degrees F	Unpressed	off
Below 100 degrees F	Pressed	ON
Above 100 degrees F	Unpressed	off
Above 100 degrees F	Pressed	off

Circuit #3 – design and sketch a circuit using a pushbutton switch and a level switch to control a lamp:

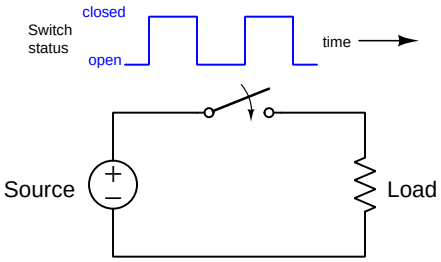
Pushbutton status	**Liquid level**	**Lamp status**
Unpressed	Below 60 cm	ON
Unpressed	Above 60 cm	ON
Pressed	Below 60 cm	ON
Pressed	Above 60 cm	off

Challenges

- Suppose the speed switch was coupled to an electric motor shaft, turning a water pump that was necessary to supply water pressure to a town. This pump is critically important, and runs all the time. Would this switch, installed in this application, be called *normally-open* or *normally-closed*?

7.1.6 Pulse-width modulation

A very common technique applied in power electronics – called *pulse-width modulation*, or *PWM* – is to rapidly switch power on and off to a load in order to approximate the effect of less-than-full power applied to that load. For example, if the switch is rapidly[4] opened and closed in square-wave fashion (i.e. on for half the time, off for half the time), the load will dissipate one-half of its full power:



Is PWM an example of digital or analog power control?

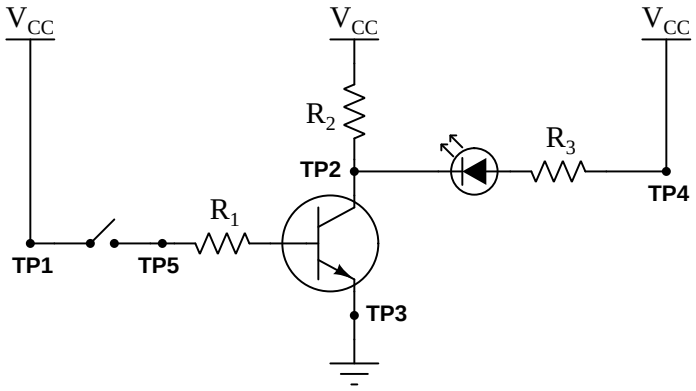
Challenges

- Identify any factors that would dictate the practical switching frequency for this power control circuit.
- Fuel injectors used to meter gasoline into electronically-controlled internal combustion engines use PWM to vary how much fuel is injected. Why do you suppose this is done rather than use a variable-width nozzle to control the flow of gasoline sprayed into the engine's intake?
- Identify some alternatives to the use of an on/off switch to control power to the load.

[4] For some applications the ideal switching speed might be as rapid as thousands of cycles per second, in which case a *solid-state* switch (i.e. power transistor) driven by a square-wave oscillator circuit would be the only reasonable option.

7.1.7 Logic levels in a simple transistor circuit

Determine the logic levels (either "high" or "low') at each of the test points in this digital logic circuit with the toggle switch in either position, as well as the status of the transistor and LED:



Switch open:

- TP1 =
- TP2 =
- TP3 =
- TP4 =
- TP5 =
- Transistor =
- LED =

Switch closed:

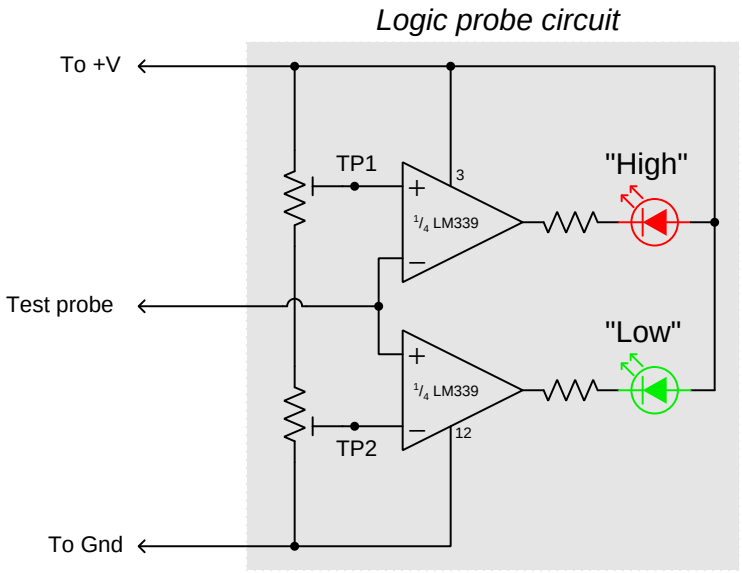
- TP1 =
- TP2 =
- TP3 =
- TP4 =
- TP5 =
- Transistor =
- LED =

Challenges

- If we consider this circuit to be a logic gate with TP5 as the input terminal and TP2 as the output terminal, what is its function?
- Explain how you would properly select the resistance value for R_1.
- Explain how you would properly select the resistance value for R_2.
- Explain how you would properly select the resistance value for R_3.

7.1.8 Simple logic probe circuit

A *logic probe* is a very useful tool for working with digital logic circuits. It indicates "high" and "low" logic states by means of LEDs, giving visual indication only if the voltage levels are appropriate for each state. The following schematic diagram shows a logic probe circuit built using comparators:



Identify the purpose of the two potentiometers in this circuit.

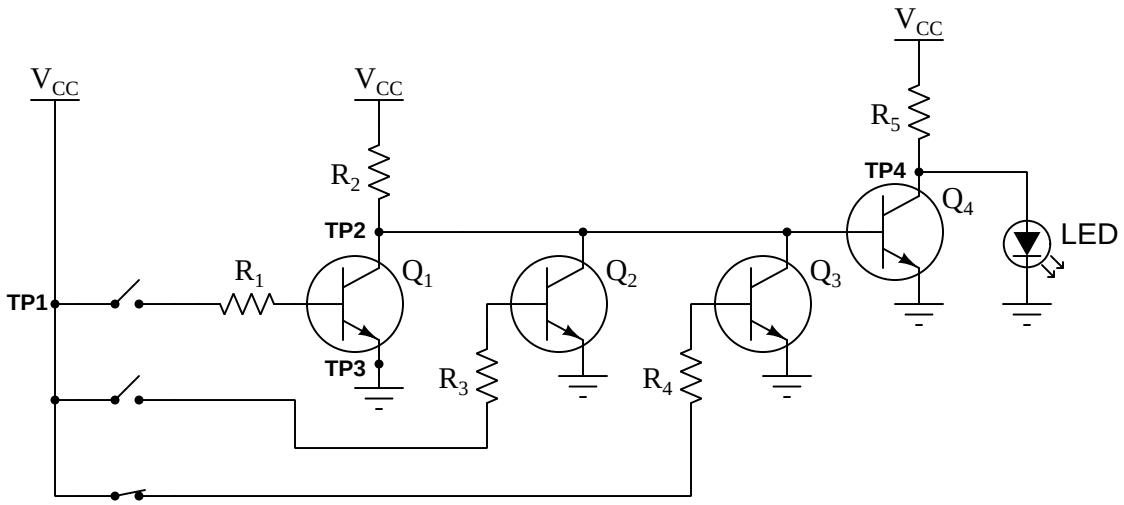
Pose a "thought experiment" by which you could demonstrate how this circuit is supposed to function, given a discrete voltage signal input to its test probe.

Challenges

- Write a formula for calculating appropriate current-limiting resistor sizes for the two LEDs in this circuit, given the value of $+V$ and the LED forward voltage and current values.

7.1.9 Logic levels in a bipolar logic circuit

Determine the logic levels (either "high" or "low') at each of the test points in this bipolar-transistor logic circuit with the toggle switches in the positions shown, as well as the status of the transistors and LED:



- TP1 =
- TP2 =
- TP3 =
- TP4 =
- Transistor Q_1 =
- Transistor Q_2 =
- Transistor Q_3 =
- Transistor Q_4 =
- LED =

If we consider this circuit to be a logic gate with the three switches as inputs and TP4 as the output terminal, what is its function?

Challenges

- Explain how you would properly select the resistance value for R_1.
- Explain how you would properly select the resistance value for R_2.
- Explain how you would properly select the resistance value for R_5.

7.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[5]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[6] on an answer key!

[5] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[6] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

7.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = 1.25663706212(19) $\times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = 8.8541878128(13) $\times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = 6.67430(15) $\times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

7.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[7] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[7] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[8] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[9] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[10] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[8] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[9] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[10] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[11] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[11] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

7.2.3 Using Python to evaluate basic logic expressions

Python is a computer programming language that is able to be run in an *interpreted* environment. This means you can start up a software application called a *Python interpreter*, and within that application type Python commands which will be immediately executed. One of the many features of this programming language is the ability to handle discrete logical values such as *True* and *False*, and the ability to apply logical operations to those values such as *and* and *or* and *not*.

The following example shows four commands typed at the prompt (`>>>`) of a Python interpreter[12] demonstrating the truth table for a two-input OR function, with the results immediately following the typed line. The final command, `quit()`, exits the Python interpreter:

```
Python 3.7.2 (default, Feb 19 2019, 18:15:18)
[GCC 4.1.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
>>> quit()
```

Once you have Python installed and working on your computer[13], demonstrate the following:

- A two-input AND function
- A three-input OR function
- A three-input AND function
- A NOT function
- A two-input NAND function
- A two-input NOR function

[12] To start the Python interpreter, simply type `python3` (for version 3 of Python, the newest at the time of this writing) at the command-line prompt of any computer with Python installed.

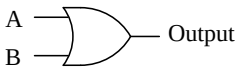
[13] One option if you do not wish to install Python on your computer is to use the online interpreter available at `https://python.org/shell`.

Challenges

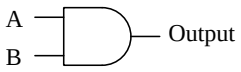
- How many commands would you need to enter at the Python prompt to fully explore the truth table of a four-input AND function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of a four-input OR function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of an n-input logical function?

7.2.4 Logic gate truth tables

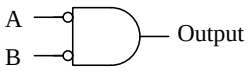
Identify each of these logic gates by name, and complete their respective truth tables:



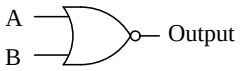
A	B	Output
0	0	
0	1	
1	0	
1	1	



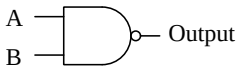
A	B	Output
0	0	
0	1	
1	0	
1	1	



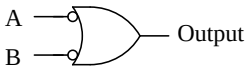
A	B	Output
0	0	
0	1	
1	0	
1	1	



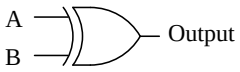
A	B	Output
0	0	
0	1	
1	0	
1	1	



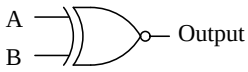
A	B	Output
0	0	
0	1	
1	0	
1	1	



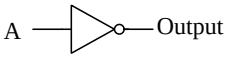
A	B	Output
0	0	
0	1	
1	0	
1	1	



A	B	Output
0	0	
0	1	
1	0	
1	1	



A	B	Output
0	0	
0	1	
1	0	
1	1	



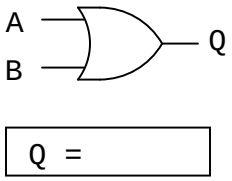
A	Output
0	
1	

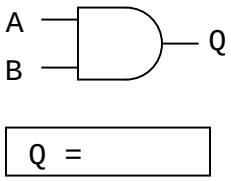
Challenges

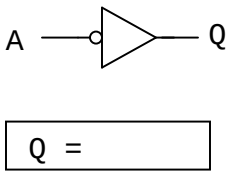
- Describe the process by which you could *figure out* an unfamiliar logic gate's truth table, if portions of that logic gate symbol were familiar.
- Careful inspection will reveal that some of the gates shown have identical truth tables. Do you see any patterns that predict when the truth table for one type of gate might be identical for another type of gate?

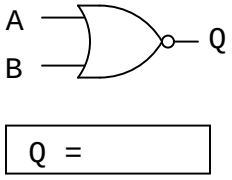
7.2.5 Boolean expressions for logic functions

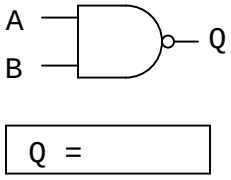
Write the Boolean expression for each of these logic functions, showing how the output (Q) algebraically relates to the inputs (A and B):

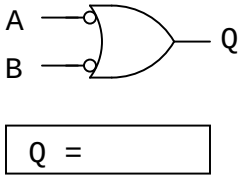


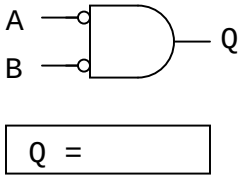










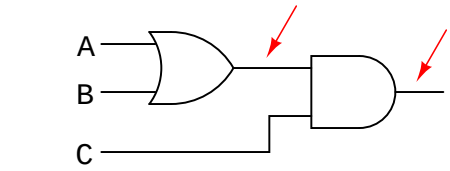


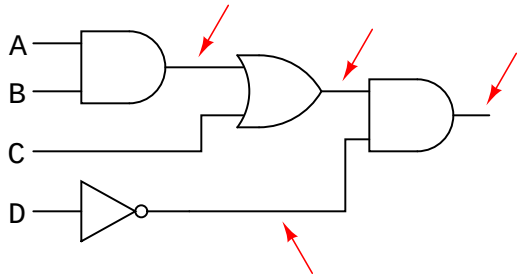
Challenges

- ???
- ???
- ???
- ???

7.2.6 Boolean expressions for simple combinational networks

Convert the following logic function networks into equivalent Boolean expressions, writing Boolean sub-expressions next to each function output in the diagrams:





Challenges

- Write a truth table for each of these combinational functions.

7.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

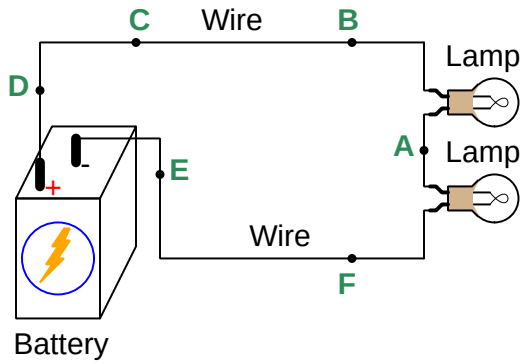
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

7.3.1 Electrical faults in a simple battery/lamp circuit

The following circuit uses a single battery to power two electric lamps. Following this diagram is a set of fault scenarios, where one wire segment or component has failed in a certain way (e.g. failed *open*, failed *shorted*). Your task is to analyze each of the scenarios, one at a time, and identify all potential faults, any one of which could account for the symptoms described.



Scenario #1: both lamps are dark, and we measure full battery voltage between points C and E.

Scenario #2: both lamps are dark, and we measure full battery voltage between points A and F.

Scenario #3: both lamps are dark, and we measure zero voltage between points B and F.

Scenario #4: both lamps are dark, and we measure full battery voltage between points C and E, but no voltage between points C and F.

Scenario #5: the upper lamp is dark and the lower lamp is glowing brightly, and we measure full battery voltage between points C and E.

Finally, identify the guaranteed effect of an electrical *short* on either voltage or current for the failed component, as well as the guaranteed effect of an electrical *open* on either voltage or current for the failed-open wire or component.

Challenges

- For any of these scenarios, is there a single given symptom conclusive enough to positively identify the location and nature of the fault without any further information? If not, can you postulate one symptom that would be sufficient?
- For any of these scenarios, is there a symptom given to us that does *not* help us diagnose the location or nature of the fault at all?
- For any of these scenarios, what would be a good *next test* to perform with a multimeter to further identify the nature and/or location of the fault?

- Identify points in this circuit that are electrically common with each other (when the circuit is intact and functioning properly).
- Identify points in this circuit that are electrically distinct from each other (when the circuit is intact and functioning properly).

7.3.2 Strange arithmetic

Boolean algebra is a strange sort of math. For example, the complete set of rules for Boolean addition is as follows:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

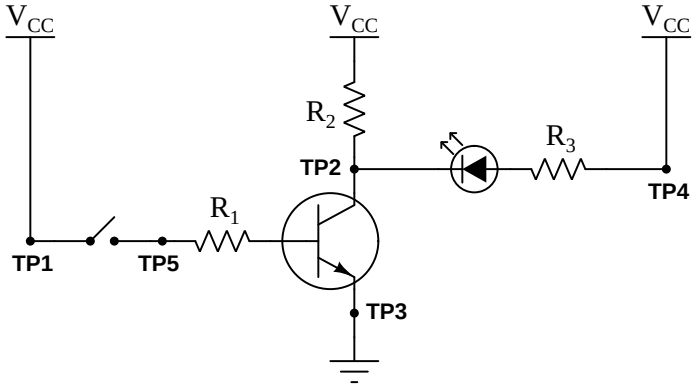
Suppose a student saw this for the very first time, and was quite puzzled by it. What would you say to him or her as an explanation for this? How in the world can $1 + 1 = 1$ and not 2? And why are there no more rules for Boolean addition? Where is the rule for $1 + 2$ or $2 + 2$?

Challenges

- Compare and contrast *negative* quantities with Boolean *complements*.

7.3.3 Effects of faults in simple digital circuit

Identify the effect(s) of each fault in the following circuit, with only one fault considered at a time:



- R_1 fails open:
- R_2 fails open:
- R_2 fails shorted:
- R_3 fails open:
- R_3 fails shorted:
- Transistor fails shorted (emitter-to-collector):

Challenges

- Identify the factors affecting proper sizing for each of the three resistors.
- Identify the type of logic function represented by this simple circuit.

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.

- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.

- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.

- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long technical career, education beats training every time.

- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.

- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.

- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.

- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and to dialogue with the industry representatives at those meetings.
- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.
- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.
- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for helping students build professional habits.
- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.
- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable**. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

B.4 Design of these learning modules

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[4] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[5] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[6] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

A Manual of Operation for the Automatic Sequence Controlled Calculator, Harvard University Press, Cambridge, Massachusetts, 1946.

Bogart, Theodore F. Jr., *Introduction to Digital Circuits*, Glencoe division of Macmillan/McGraw-Hill, 1992.

Shannon, Claude Elwood, *A Symbolic Analysis of Relay and Switching Circuits*, thesis paper submitted as partial fulfillment of the requirements for the degree of Master of Science at MIT, 1940.

Vail, Alfred, *Early History of the Electro-Magnetic Telegraph*, Hine Brothers, New York, NY, 1914.

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

24 April 2026 – added a new Diagnostic Reasoning question.

22 January 2026 – added a new Technical Reference section on process switch types and symbols.

27 November 2025 – added a new Conceptual Reasoning question challenging students to sketch switch/lamp circuits implementing simple digital function truth tables.

13 November 2025 – typographical error corrections courtesy of Wyatt Gleason.

4 March 2025 – added a Conceptual Reasoning question on identifying switch states.

16 October 2024 – added an Introduction section on challenging concepts.

19 August 2024 – added a Case Tutorial chapter with a section on bitwise operations with graphic images borrowed from mod_c.

21 July 2024 – moved all content from the Case Tutorial section on the NAND gate demonstration circuit to a common file to be shared among multiple modules, and also added a TTL version to that demonstration to complement the CMOS.

15 July 2024 – divided the Introduction chapter into two sections, one for students and one for instructors, and added content to the instructor section recommending learning outcomes and measures.

17 April 2024 – minor typo corrections, and new pagebreaks added in the "Boolean expressions" Tutorial section for better readability.

13 April 2024 – added a new Conceptual Reasoning question showing a three-input OR gate made

of BJTs.

3 January 2024 – edited the water storage tank illustration to show more than two tanks, using three-input OR and AND functions to handle three tanks' level switch signals. Also edited other portions of that section to discuss logic functions with more than two inputs each.

10 August 2023 – deleted an unnecessary line of text in the "Example: simple logic functions using switches" Case Tutorial section.

7 December 2022 – added content on integrated logic circuits and comparators to the Tutorial.

10 August 2022 – updated reference on Unicode, from 16 bits (its original form) to 21 bits. Also added an oscillograph contrast of discrete versus analog signals at the suggestion of Grant Dearing.

20 April 2022 – made some minor edits to the Tutorial.

17 November 2021 – deleted Cygwin reference in footnote.

30 July 2021 – minor edit to inverter symbol in Case Tutorial.

26-27 July 2021 – added some Case Tutorial sections showing some basic logic gate demonstration circuits.

2 July 2021 – added sections to the Tutorial and added a new Case Tutorial chapter.

9 May 2021 – commented out or deleted empty chapters.

17 April 2021 – added potentiometer/decade box analogy to the Tutorial.

8 October 2020 – eliminated an example program from the "Modeling Boolean logic using C++" section. I was using arithmetic operators in the first example. Had I used the values 1 for both A and B, the OR function's output would have displayed as 2 rather than 1 as it should be for a Boolean sum. Turns out the `cout` function assumes an integer result rather than a Boolean result even when the quantities being added are both Boolean variables.

6 October 2020 – minor edits to Tutorial, and added some instructor comments.

29 September 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions.

10 April 2020 – minor typographical error correction, courtesy of Ron Felix.

3 March 2020 – added mention of *telegraph* systems as early digital communication networks, and added a Historical Reference section describing Samuel Morse's early encoding scheme.

5 January 2020 – added bullet-list of relevant programming principles to the Programming References section.

2 January 2020 – removed from from C++ code execution output, to clearly distinguish it from the source code listing which is still framed.

1 January 2020 – changed `main ()` to `main (void)` in C++ programming example, and changed them significantly to show more types of logical functionality. Also, added more explanatory text to the C++ programming examples, and added examples using custom functions.

23 December 2019 – shortened name of Historical References section to "Punched paper tape", also made a minor edit to a question.

13 November 2019 – minor edits.

14 April 2019 – added index entry for "process switch".

12 April 2019 – edited Quantitative Reasoning question on the use of Python to evaluate basic logic functions, asking the reader to demonstrate at least one example of the function, not the entire truth table.

11 April 2019 – added a Quantitative Reasoning question centered around the use of Python to evaluate basic logic functions.

24 March 2019 – added Conceptual Reasoning questions.

10 March 2019 – elaborated on the concept of a "discrete" signal as a type of digital signal, and added some entries to the Foundational Concepts list.

5 March 2019 – completed Foundational Concepts list.

18 February 2019 – minor edits to the Tutorial, and added terms to the Foundational Concepts subsection.

3 February 2019 – added the Exclusive-OR function to the Tutorial.

6 January 2019 – clarified equivalence between AND/OR gate truth tables and Boolean arithmetic operations. Also added explanations of how to express logical complementation in Boolean form, and added a reference to Claude Shannon's groundbreaking Master's thesis to the Historical References chapter.

24 December 2018 – added a few index references.

18 December 2018 – document first created.

Index

Absolute encoder, 30
Active component, 25, 31
ADC, 30
Adding quantities to a qualitative problem, 94
Analog, 17
Analog-to-digital converter, 30
AND function, 22
Annotating diagrams, 93
ASCII, 30

Binary numeration, 29
Boolean addition, 27
Boolean algebra, 26
Boolean multiplication, 26

C++, 28, 44
Checking for exceptions, 94
Checking your work, 94
Code, computer, 103
Comparator, 31
Compiler, C++, 44
Complementation, 28
Computer programming, 43

DAC, 30
Digital, 17
Digital-to-analog converter, 30
Dimensional analysis, 93
DIP, 32
Discrete, 19
Dual Inline Package, 32

Edwards, Tim, 104
Encoder, 30

Gate, logic, 25, 31
Graph values to solve a problem, 94
Greenleaf, Cynthia, 63

Hinderance, 38
How to teach with these modules, 101
Hwang, Andrew D., 105

IC, 25, 31
Identify given data, 93
Identify relevant principles, 93
Integrated circuit, 25, 31
Intermediate results, 93
Interpreter, Python, 48
Inverted instruction, 101

Java, 45

Knuth, Donald, 104

Lamport, Leslie, 104
Limit case, 19
Limiting cases, 94
Logic function, 3
Logic gate, 25, 31
Logic state, 19

Maxwell, James Clerk, 33
Metacognition, 68
Moolenaar, Bram, 103
Murphy, Lynn, 63

NAND function, 24
Negation, 28
NOR function, 24
NOT function, 24

Open-source, 103
OR function, 22

Passive component, 25
Pinout diagram, 25, 32

Power supply rail, 31
Prime, 28
Problem-solving: annotate diagrams, 93
Problem-solving: check for exceptions, 94
Problem-solving: checking work, 94
Problem-solving: dimensional analysis, 93
Problem-solving: graph values, 94
Problem-solving: identify given data, 93
Problem-solving: identify relevant principles, 93
Problem-solving: interpret intermediate results, 93
Problem-solving: limiting cases, 94
Problem-solving: qualitative to quantitative, 94
Problem-solving: quantitative to qualitative, 94
Problem-solving: reductio ad absurdum, 94
Problem-solving: simplify the system, 93
Problem-solving: thought experiment, 93
Problem-solving: track units of measurement, 93
Problem-solving: visually represent the system, 93
Problem-solving: work in reverse, 94
Process switch, 20
Programming, computer, 43
Pulse width modulation, 74
PWM, 74
Python, 28, 48

Qualitatively approaching a quantitative problem, 94

Rail, power supply, 31
Reading Apprenticeship, 63
Reductio ad absurdum, 94, 100, 101

Schoenbach, Ruth, 63
Scientific method, 68
Shaft encoder, 30
Shannon, Claude, 38
Signal, 17
Simplifying a system, 93
Socrates, 100
Socratic dialogue, 101
Source code, 44
SPICE, 63
Stallman, Richard, 103
Switch, process, 20

Telegraph, 30
Thought experiment, 93
Torvalds, Linus, 103
Truth table, 23

Unicode, 30
Units of measurement, 93

Visualizing a system, 93

Whitespace, C++, 44, 45
Whitespace, Python, 51
Work in reverse to solve a problem, 94
WYSIWYG, 103, 104