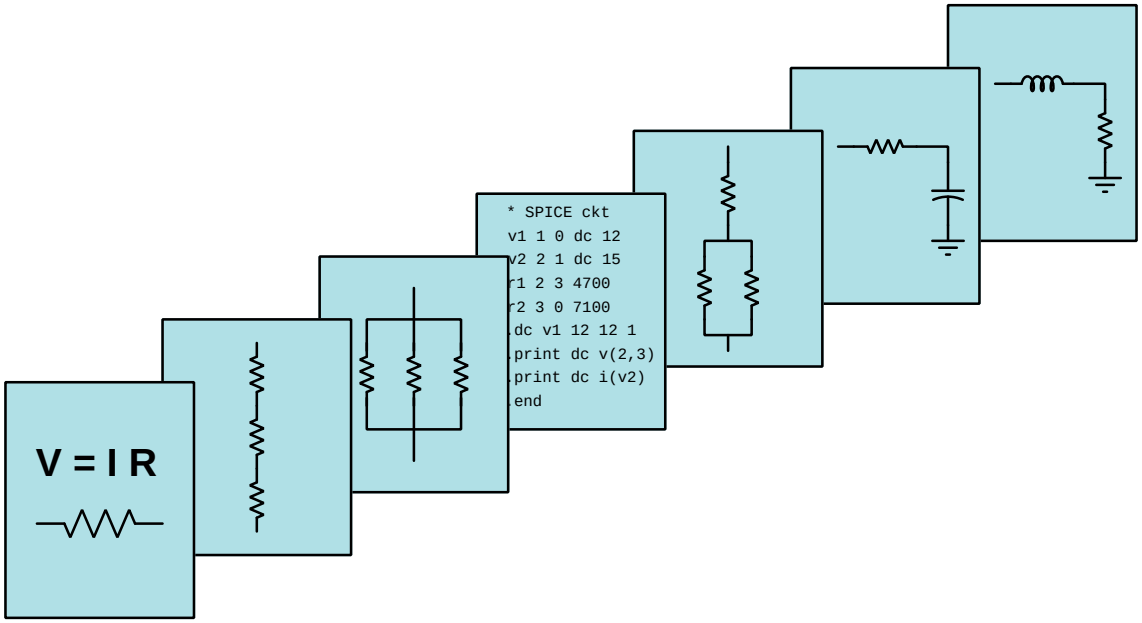
Modular Electronics Learning (ModEL) project



Combinational Logic

Last update = 6 May 2025

Contents

Chapter 1

Introduction

1.1 Recommendations for students

Logic functions such as AND, OR, and NOT form the basis of nearly all digital systems. However, each one of these functions by itself is of limited use. Practical applications usually demand logic functions of additional complexity, and when multiple AND/OR/NOT functions are connected together to form a more advanced logical function the result is called *combinational logic*.

A defining characteristic of any combinational logic system is that the output state(s) are entirely defined by the input states. That is to say, it is possible to write a *truth table* for any combinational logic function exactly describing which input states immediately lead to which output states. This stands in contrast to other digital functions such as *timing functions* and *latching functions*, the output states of which depend on input states as well as past history for that function (e.g. elapsed time for timing functions, and previous output states for latching functions).

Important concepts related to combinational logic include **logic states**, **logic levels**, **high** and **low** logic states, **logic functions**, **truth tables**, **Boolean algebra**, **logic gates**, electromechanical **relays**, logic function **universality**, **DeMorgan's Theorem**, the **normal state** of a switch, algebraic **order of operations**, **sum-of-products**, **product-of-sums**, and logic circuit **minimization**.

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

- How might an experiment be designed and conducted to explore DeMorgan's Theorem? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What are some practical applications of SOP and POS Boolean expressions?
- What does a *truth table* represent?
- What is a *logic function*?
- What are some basic forms of logic functions?

- How may we use lettered variables to represent logic states and logic functions?
- How are digital logic states represented as electrical voltage signals?
- How may logical functions be implemented using electromechanical relays?
- Which logic functions may be configured to emulate other logic functions?
- How do series and parallel switch networks relate to logical functions?
- What is a practical application for "universal" logic functions?
- What is DeMorgan's Theorem and where might we apply it?
- How does a Boolean product relate to logic gate functions, and also to relay contact networks?
- How does a Boolean sum relate to logic gate functions, and also to relay contact networks?
- Why is the *order of operations* important when translating a Boolean expression into a logic circuit?
- Describe how a "Sum of Products" (SOP) Boolean expression may be derived from a truth table.
- Describe how a "Negative Sum of Products" (NSOP) Boolean expression may be derived from a truth table.
- Describe how a "Product of Sums" (POS) Boolean expression may be derived from a truth table.
- Why is it useful to have multiple techniques for translating a truth table into a Boolean expression?
- Why might it be beneficial to reduce or minimize logic circuits?

1.2 Challenging concepts related to combinational logic

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

- **Sourcing versus Sinking output currents** – a common misconception is that since the output of a logic gate is called "output" it must mean that current only ever *exits* that terminal. This is untrue. All that "output" actually signifies is the fact that the gate is outputting *information* consisting of voltage values measured between that terminal and ground. Sometimes the assertion of a "low" (zero-voltage) logical state requires that the gate actually draw current *in* through its "output" terminal!

- **Gates require DC power** – since logic gates are really just transistor amplifier circuits, those treansistors require constant DC voltage applied between their power terminals to function properly. Gates are *not* powered through their input terminals – these terminals only receive *information* in the form of "high" and "low" logic states, and ideally pass negligible current.

- **Pullup and pulldown resistors** – in digital circuits, resistors are often used to provide a secure logic state when an input device (such as a switch) goes to a high-impedance (open) mode. Students often have difficulty figuring out exactly where these resistors should go in a circuit. The most common mistake I've seen is to place one of these "pullup" or "pulldown" resistors in *series* with a gate input, which will accomplish absolutely nothing. The "trick" to getting this placement right, if you can call it a trick at all, is to literally follow the word "pullup" or "pulldown". A *pullup* resistor pulls the logic state of a wire up to the positive supply rail, and so must connect between the gate input and +V. A *pulldown* resistor pulls the logic state of a wire down to ground potential, and so must connect between the gate input and ground. In either case, the resistor provides a sure path to the opposite power rail that the input device connects to when active (closed).

- **DeMorgan's Theorem** – a common tendency for students is to attempt to memorize new mathematical formulae and techniques, and DeMorgan's Theorem is no exception. However, a more insightful approach is to see how DeMorgan's Theorem works in practical applications such as logic function universality, where we use just one type of logic function to emulate other logic functions.

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 explain how gate universality is proven in the Tutorial chapter's examples.

  Assessment – Students explain how logic circuits were derived from Boolean expressions in the Tutorial chapter's examples.

  Assessment – Students explain how logic circuits were derived from truth tables in the Tutorial chapter's examples.

- **Outcome** – Design logic circuits to implement given truth tables

  Assessment – Given a truth table with desired output states, design a semiconductor logic gate circuit to implement that function.

  Assessment – Given a truth table with desired output states, design an electromechanical relay "ladder logic" circuit to implement that function.

- **Outcome** – Independent research

  Assessment – Locate logic gate datasheets and properly interpret some of the information contained in those documents including supply voltage range, input logic voltage levels, output logic voltage levels, maximum switching speed, maximum output current, internal schematic diagrams (not available in all datasheets), etc.

  Assessment – Read and summarize in your own words reliable source documents on the subject of NASA'a Apollo Guidance Computer (AGC) which was built entirely of NOR gates.

Chapter 2

Case Tutorial

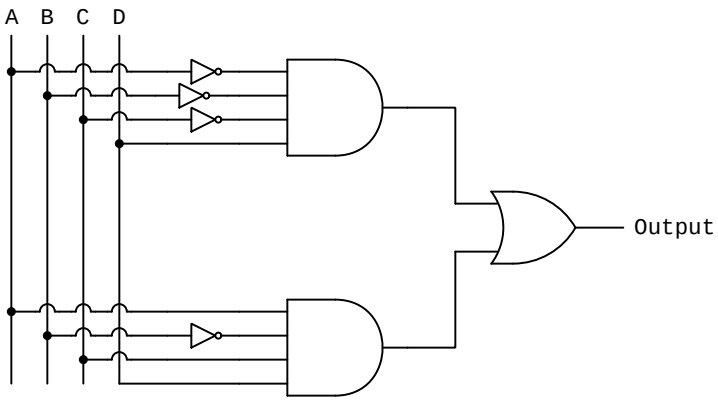
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as 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: gate circuits from 4-input truth tables

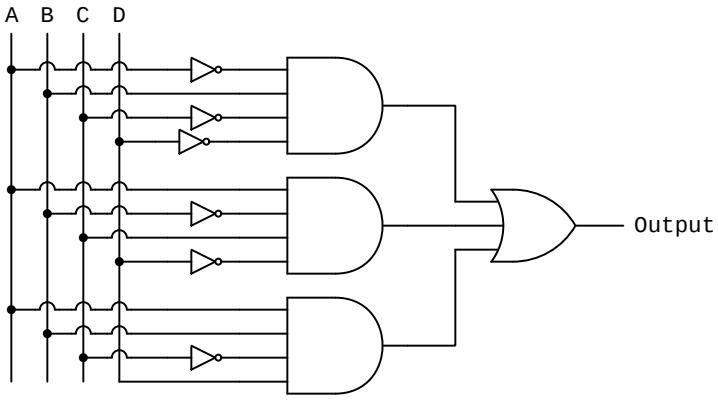
A	B	C	D	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Boolean SOP expression: $\overline{A}\,\overline{B}\,\overline{C}D + A\overline{B}CD$



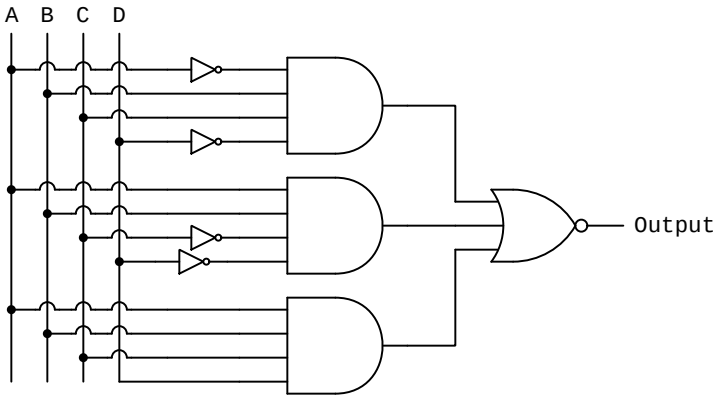
A	B	C	D	Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Boolean SOP expression: $\overline{A}B\overline{C}\,\overline{D} + A\overline{B}C\overline{D} + AB\overline{C}D$



A	B	C	D	Output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

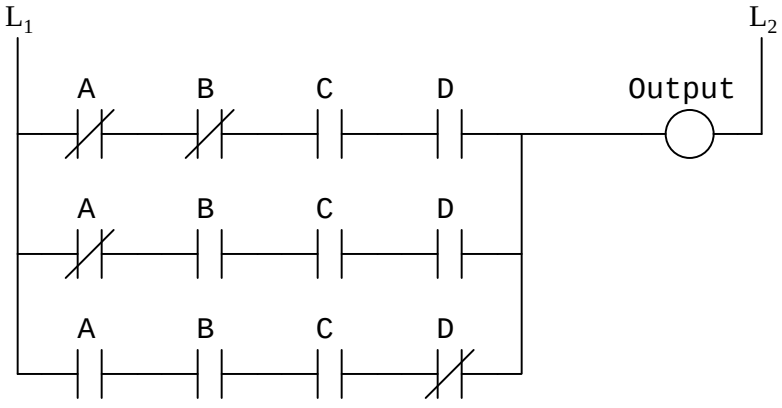
Boolean Negative-SOP expression: $\overline{\overline{A}BC\overline{D} + AB\overline{C}\,\overline{D} + ABCD}$



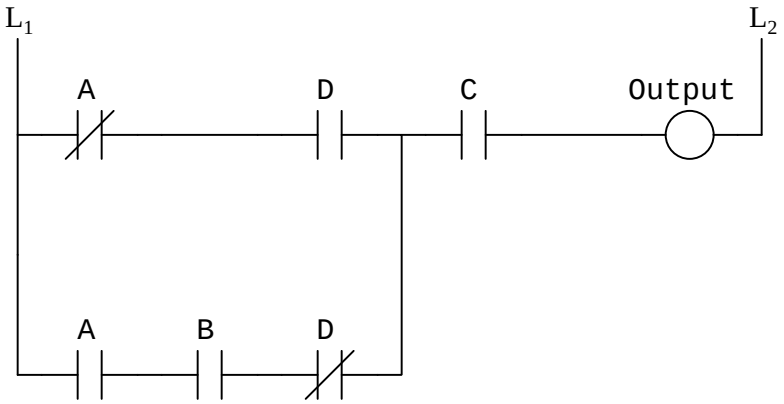
2.2 Example: ladder logic circuits from 4-input truth tables

A	B	C	D	Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Boolean SOP expression: $\overline{A}\,\overline{B}CD + \overline{A}BCD + ABC\overline{D}$



And of course it should be evident from the ladder-logic circuit that certain simplifications may be made while still retaining the same logical functionality:

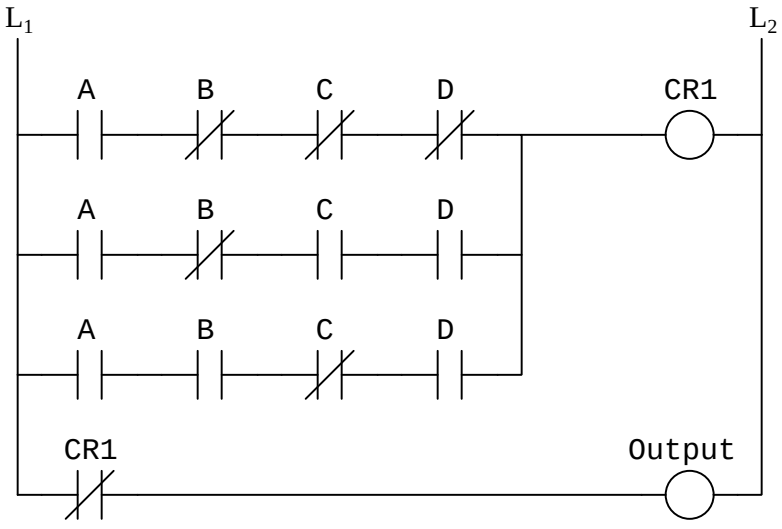


The process of simplifying the original ladder-logic circuit to what you see above is a matter of identifying identical contacts (e.g. the normally-open `C` contact) present in multiple rungs which may be consolidated into a single rung in series with the non-common contact rungs. Note also how the normally-closed `B` and normally-open `B` contacts were eliminated as the upper and middle rungs were consolidated into one rung, because having NO and NC contacts in parallel driven by the same input is pointless – there would *always* be a path through one or the other of them regardless of `B`'s state, therefore `B` need not have a presence in the consolidated rung.

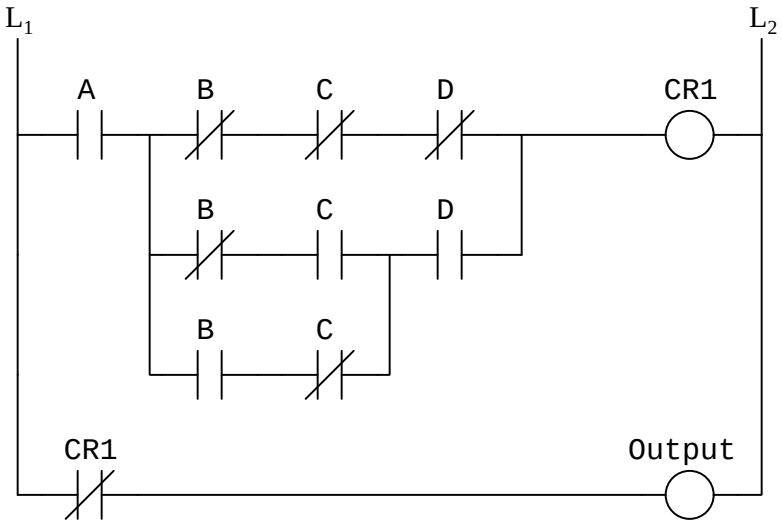
A good simplification strategy is to consolidate contacts shared amongst the greatest number of rungs first. This is why the first consolidation was the normally-open `C` contact, because it was found in all three of the original rungs. This is also why the second consolidation was the upper and middle rungs: with `C` removed they still they had `A` (normally-closed) and `D` (normally-open) in common with each other.

A	B	C	D	Output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Boolean Negative-SOP expression: $\overline{A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}CD + AB\overline{C}D}$

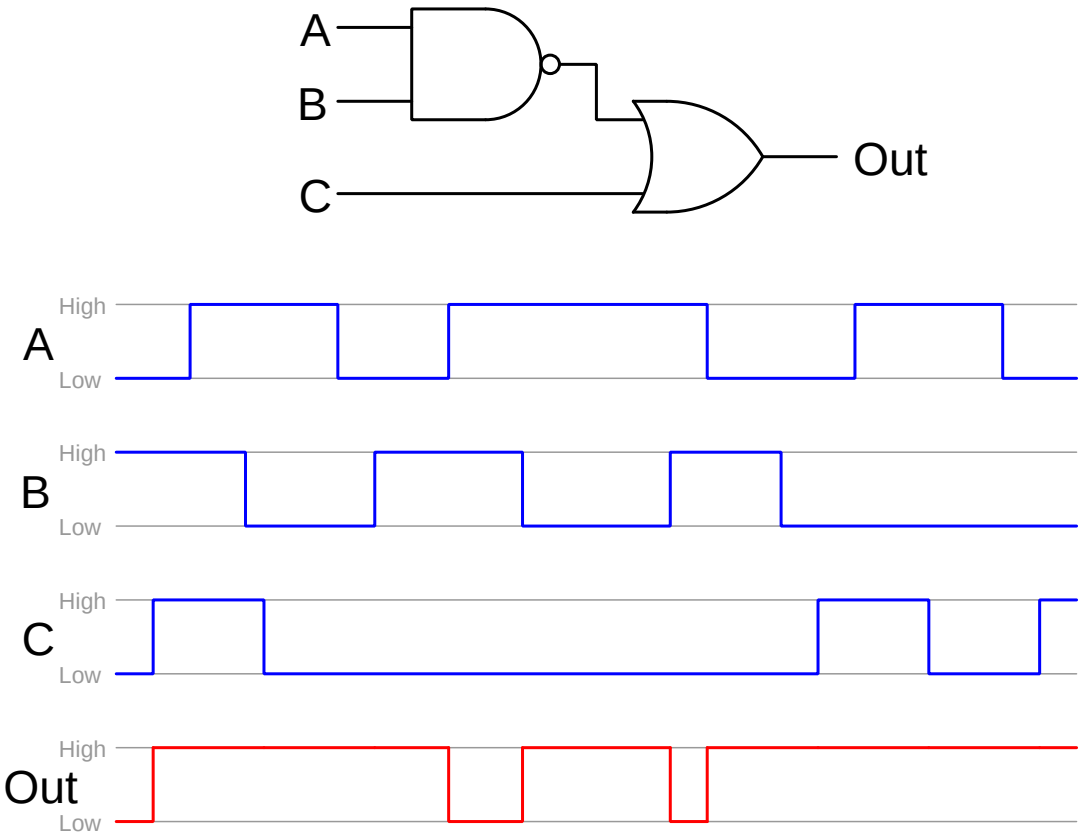


Certain simplifications are possible for this circuit while still retaining the same logical functionality:

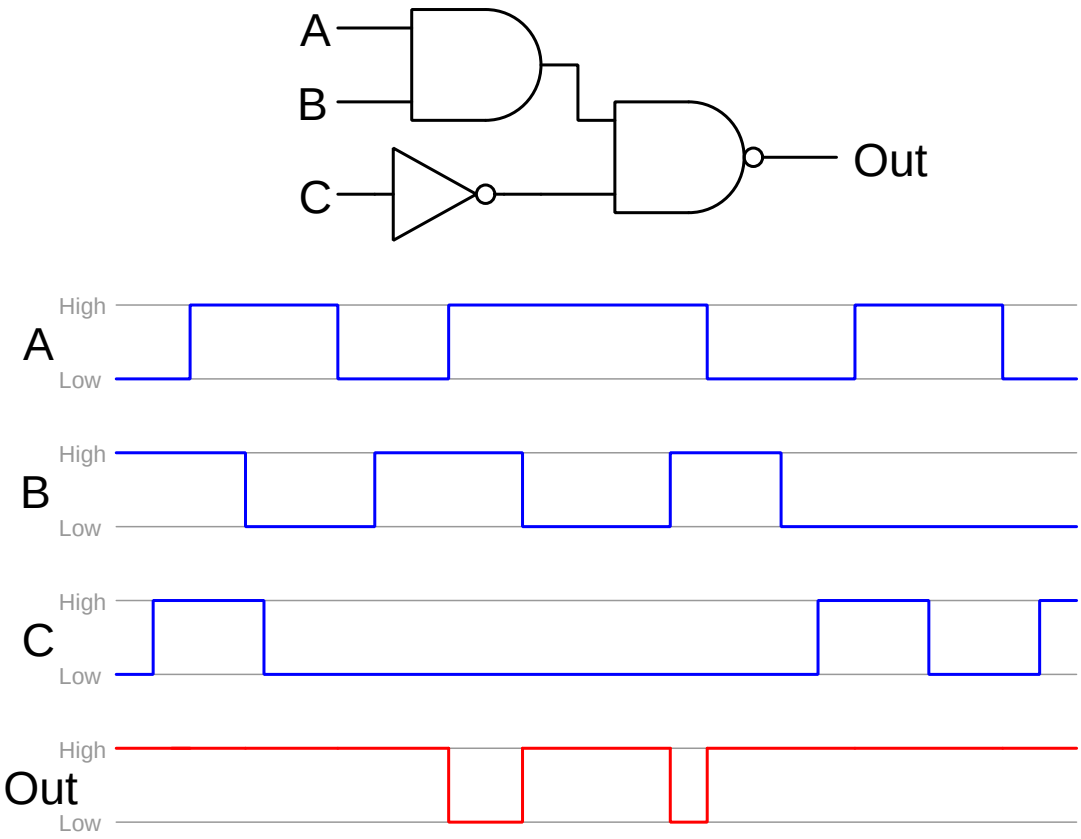


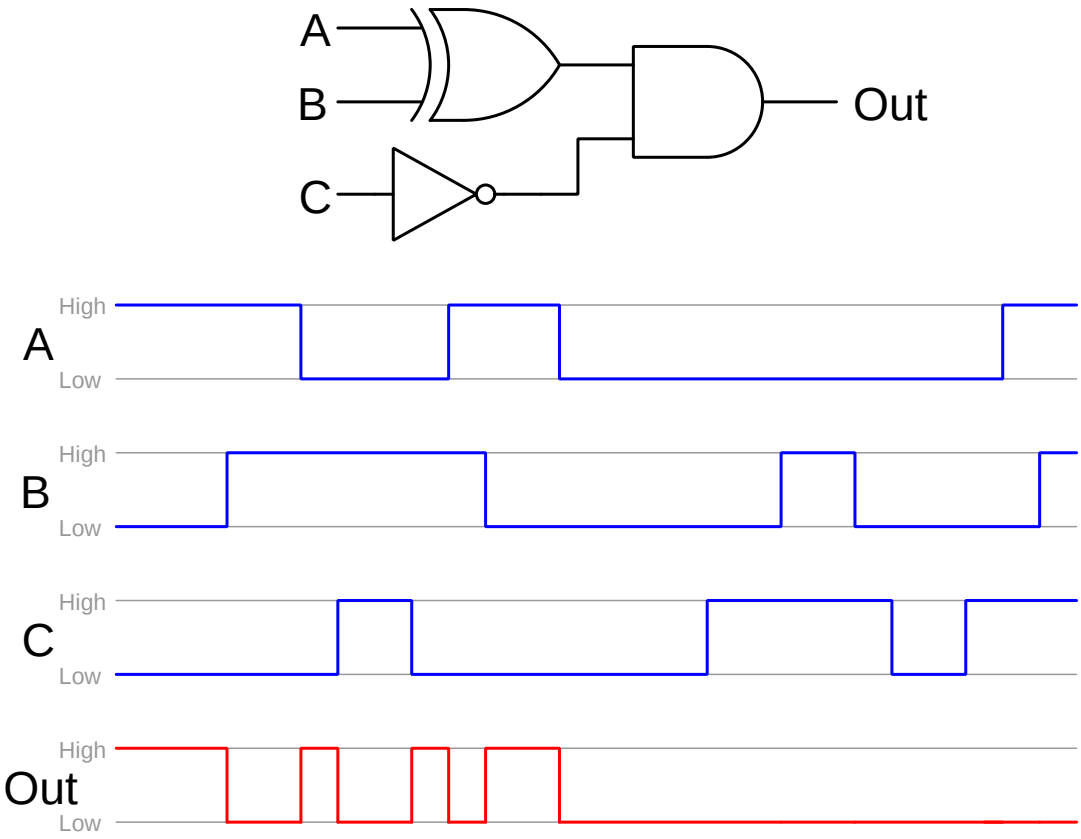
2.3 Example: timing diagrams for combinational gate circuits

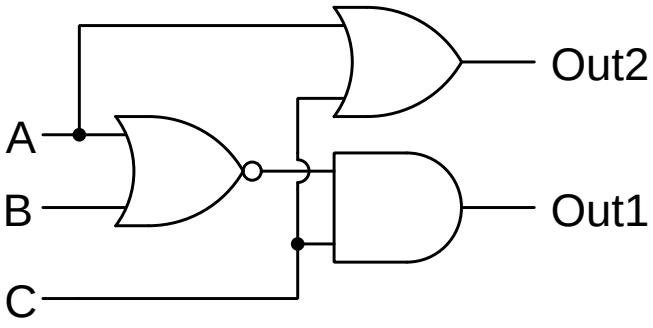
A *timing diagram* shows the "high" and "low" states of logic signals as waveforms in the time domain (i.e. with time being the horizontal axis). These diagrams are essential for analyzing logic circuits with *latching* (memory) capability, but they may also be used to document the behavior of non-latching logic circuits too. The following examples show this for different configurations of logic networks.

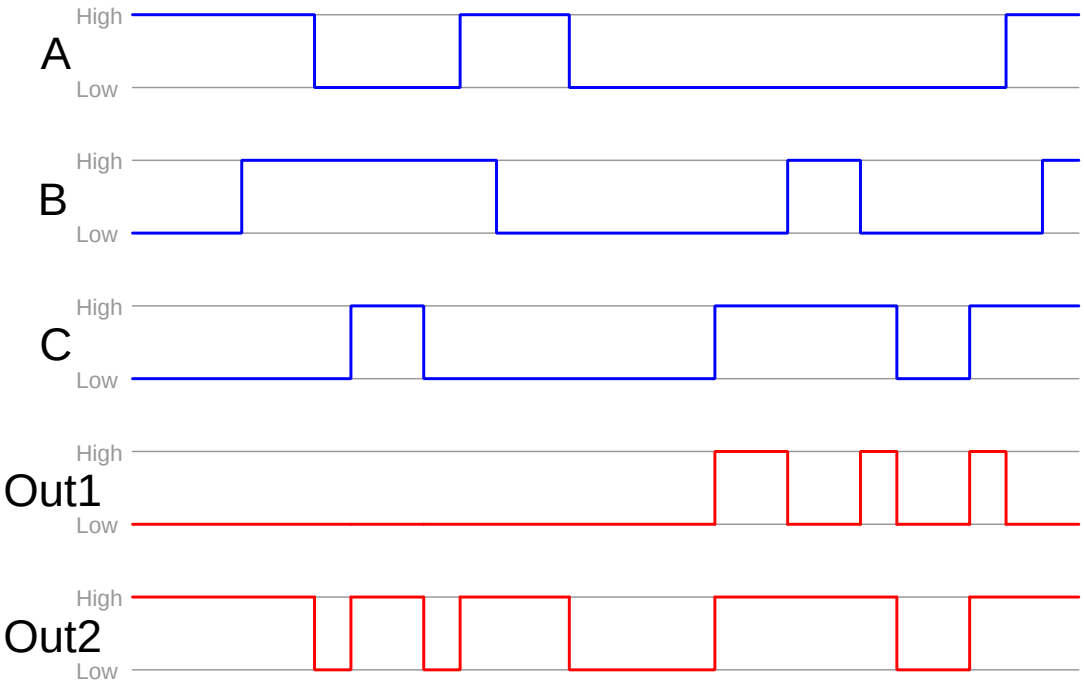


Note: a good way to approach the analysis of this and other circuits where conditions change over time is to make multiple copies of the schematic diagram – one for each different moment in time shown on the timing diagram where there is a unique set of input conditions – and then annotate each of those diagrams with all the logic conditions at that particular moment in time. Essentially, we perform several "thought experiments" on the logic circuit, each one representing a different moment with a unique set of input conditions. This breaks a complex problem down into simpler, more manageable parts.


A
B
C
Out
High
A
Low
High
B
Low
High
C
Low
High
Out
Low


A
B
C
Out
High
Low
A
High
Low
B
High
Low
C
High
Low
Out


A
B
C
Out2
Out1


A
B
C
Out1
Out2
High
Low

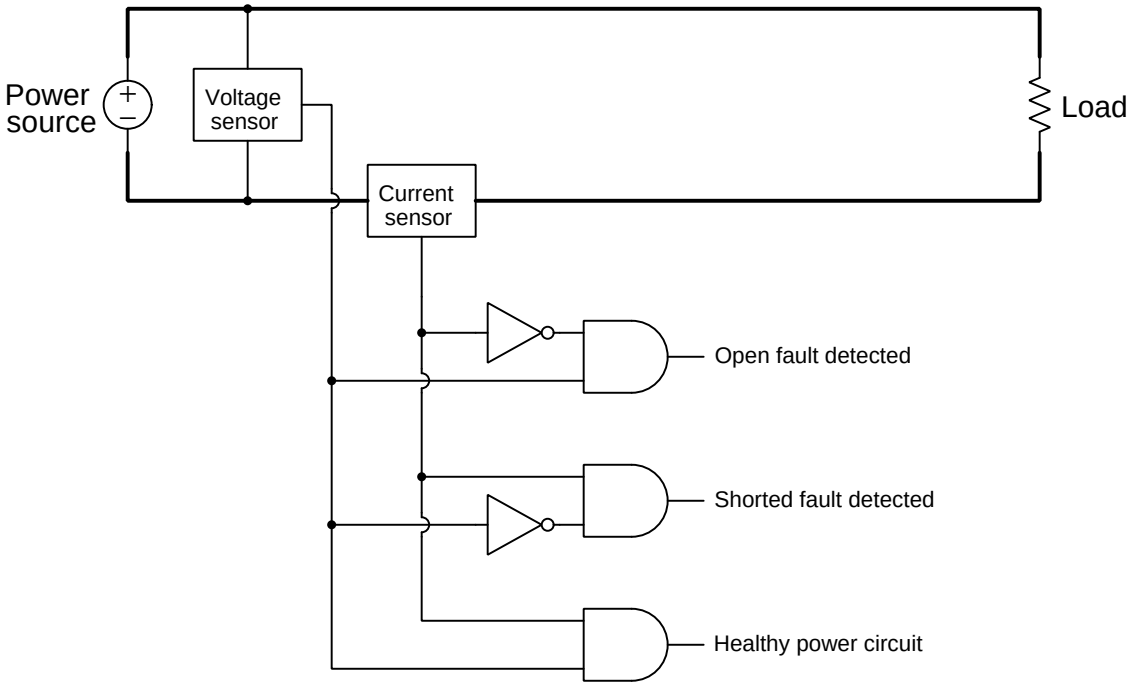
2.4 Gallery of logic gate applications

Semiconductor logic gates have many, many practical applications. The following subsections show just a few!

As is customary with most logic gate schematic diagrams, DC power terminals for the logic gates themselves have been omitted in order to reduce clutter on the diagrams. Know, however, that logic gates are transistor circuits which require DC power to function!!

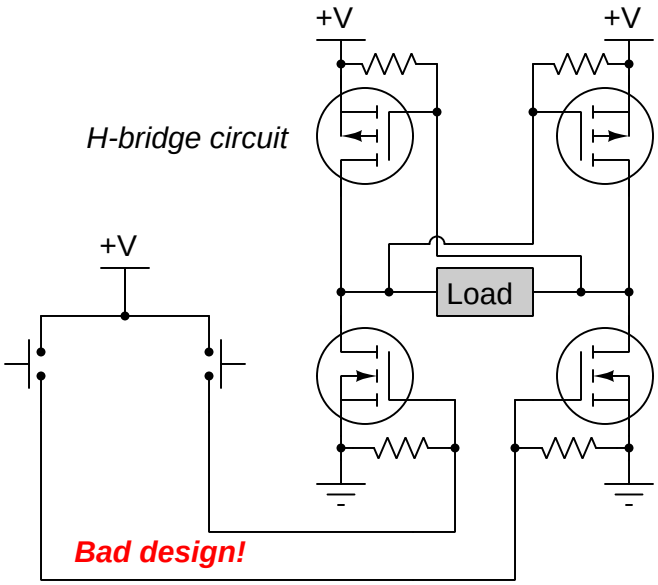
2.4.1 Power circuit fault detector

A power circuit equipped with voltage and current sensors providing boolean ("true" or "false") indications of voltage and current may be equipped with a logic circuit to take those sensor signals and from their values determine if there is any fault in the power circuit:

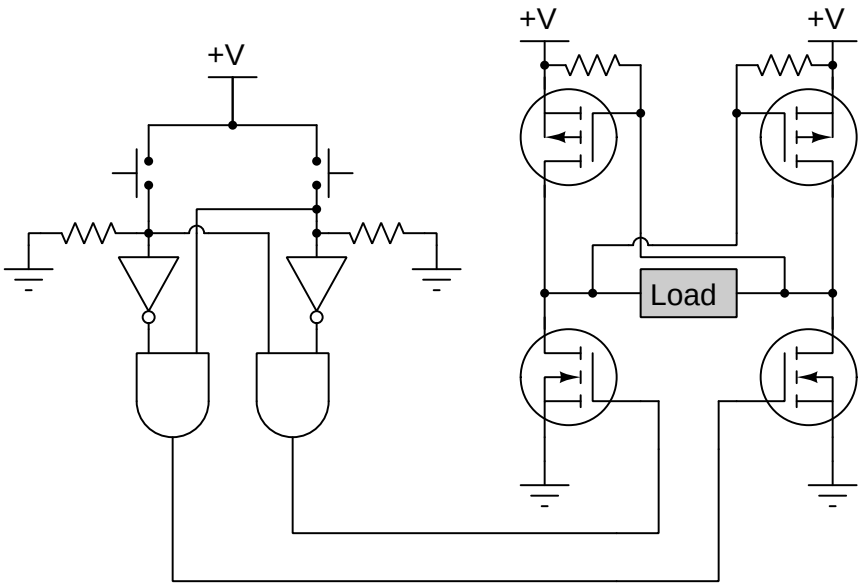


2.4.2 H-bridge driver circuit

H-bridge circuits are networks of four power transistors useful for controlling power to a load. By activating either one of the lower transistors in the bridge, the appropriate upper transistor becomes activated as well, passing current through the load in one direction or the other. However, the simplistic H-bridge circuit shown below is a bad design because all four transistors will activate (and short-circuit the +V source) if anyone happens to press both pushbuttons happen at the same time:

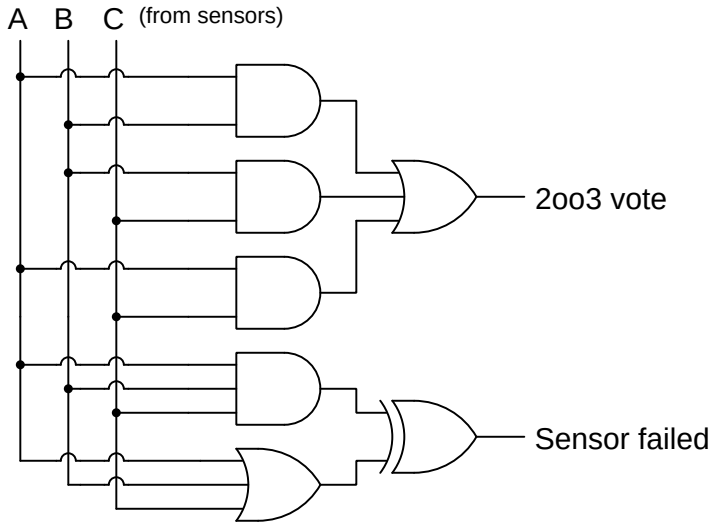


The addition of some logic gates will safeguard against this improper mode of operation, by activating the H-bridge only if *one* of the two pushbuttons is pressed but not if both are simultaneously pressed:



2.4.3 Two-out-of-three voting circuit

High-reliability systems often use redundant components to achieve fault tolerance. For example a system may use three identical sensors (A, B, and C) all detecting the same physical stimulus, all three sensors reporting their statuses to an electronic "voting" logic circuit providing a reliable output signal based on a two-out-of-three ("2oo3") vote:

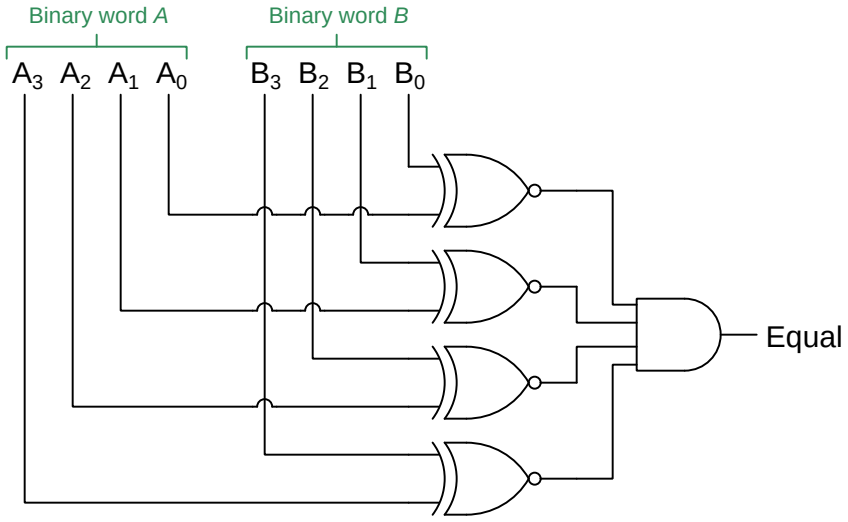


A	B	C	2oo3 vote	Sensor failed
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

All it takes is for at least two of the three redundant sensors to agree with each other to generate a reliable "high" or "low" output from the voting circuit, and if there is any disagreement at all between the three sensors we will know by the "sensor failed" output going high. This provides a fault tolerance of one, which means any one of the three sensors could fail in any state and the voting circuit would still reliably indicate the true status of the measured stimulus.

2.4.4 Binary word comparator

A *binary word* is a collection of *bits* representing a numerical quantity or a symbolic code[1]. This circuit compares two 4-bit binary words to check for equality:

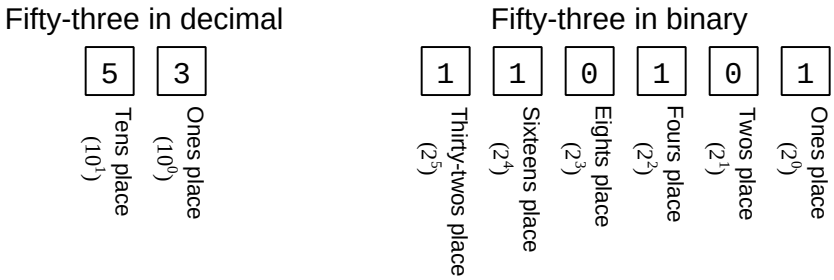


Each Exclusive-NOR gate compares a pair of respective bits in words A and B, outputting a "high" signal if those two bits are equal in state. The AND gate outputs a "high" signal only if every one of its input lines is also "high", which can only happen when the two 4-bit words are identical.

[1] For example, the American Standard Code for Information Interchange (ASCII) uses 7-bit words to represent all numerical and alphabetical characters on a standard English computer keyboard.

2.4.5 Binary decoder circuits

Binary numeration represents numerical quantities using multiple *bits*, each bit capable of being either "high" (1) or "low" (0). Basic whole-number representation in binary is simple – each bit has a place-weight that is some power of two, as opposed to *decimal* numeration where each place-weight is a power of ten. For example, compare the decimal versus binary representations of the number *fifty-three*:

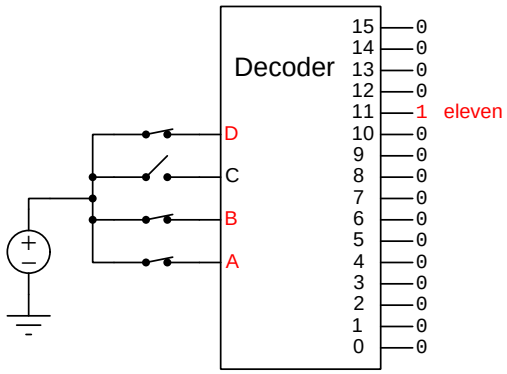


$$\text{Fifty-three} = (5 \times 10^1) + (3 \times 10^0) = 50 + 3$$

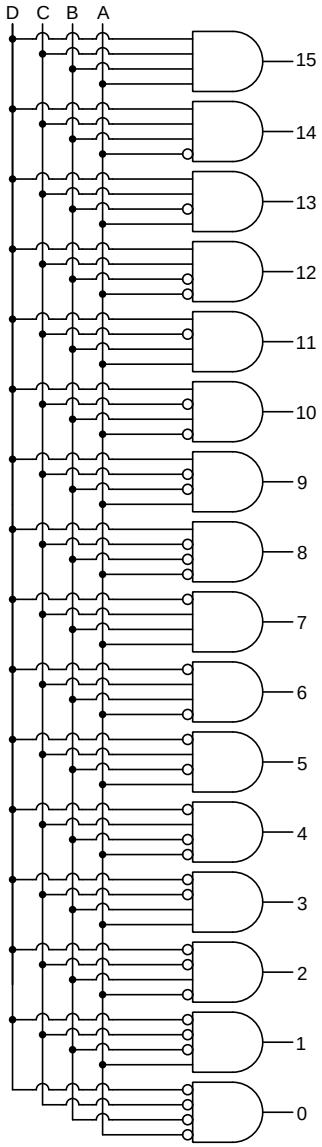
$$\text{Fifty-three} = (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$\text{Fifty-three} = (1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$$

Modern digital computers use binary as the means of representing numbers because each bit of a binary number may be unambiguously symbolized using Boolean-state (true/false, on/off, 1/0) logic circuits.

Any circuit designed to input a binary number and activate one or more outputs selected by the value of that binary number is generally referred to as a *decoder*. For example, in the illustration below we see a 4-line to 16-line binary decoder decoding the number eleven:



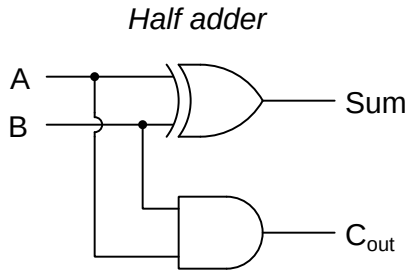
Binary decoders are really nothing more than a collection of AND and inverter (NOT) gates. In the example circuit below, inverters placed on the input of each AND gate appear as "bubbles" for the sake of compactness:



Only one of these AND gates will have its input conditions satisfied to generate a "high" output, for any given combination of bit-states in the four-bit binary input.

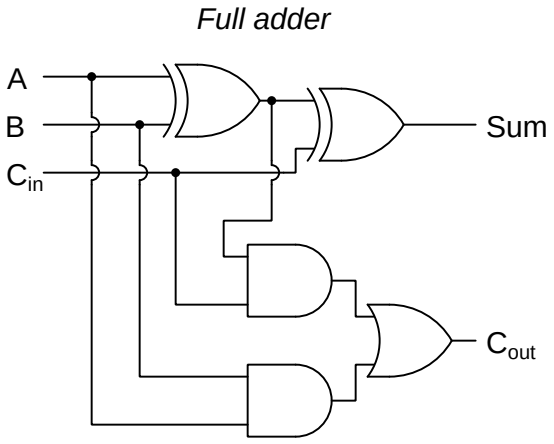
2.4.6 Binary adder circuits

Gate circuits may be built to perform simple arithmetic operations on binary numbers. Shown below is a binary *half-adder* circuit, able to add two binary bits to produce sum and a carry outputs:



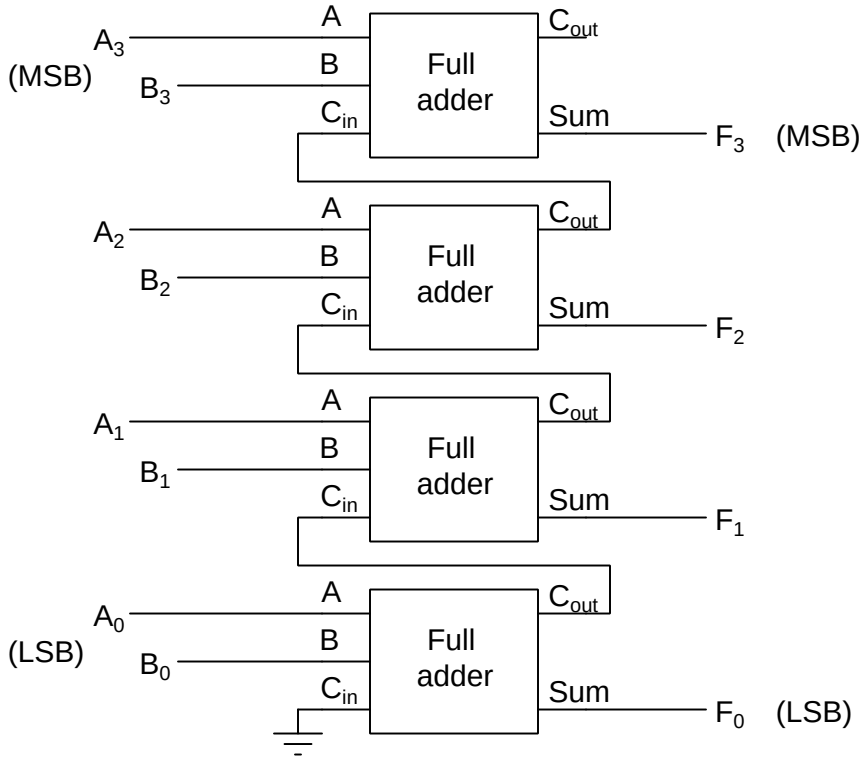
A	B	Carry out	Sum	Explanation
0	0	0	0	$0 + 0 = 0$
0	1	0	1	$0 + 1 = 1$
1	0	0	1	$1 + 0 = 1$
1	1	1	0	$1 + 1 = 2$

Next is a binary *full-adder* circuit, able to add two binary bits as well as a carry-in signal to produce sum and a carry outputs:



Carry in	A	B	Carry out	Sum	Explanation
0	0	0	0	0	$0 + 0 + 0 = 0$
0	0	1	0	1	$0 + 0 + 1 = 1$
0	1	0	0	1	$0 + 1 + 0 = 1$
0	1	1	1	0	$0 + 1 + 1 = 2$
1	0	0	0	1	$1 + 0 + 0 = 1$
1	0	1	1	0	$1 + 0 + 1 = 2$
1	1	0	1	0	$1 + 1 + 0 = 2$
1	1	1	1	1	$1 + 1 + 1 = 3$

Neither a half-adder nor a full-adder circuit is very useful on its own. In order to actually add two binary numbers of any significant magnitude together we must cascade multiple full-adder networks together. Below we see a full adder network for two 4-bit binary numbers ($A + B = F$):



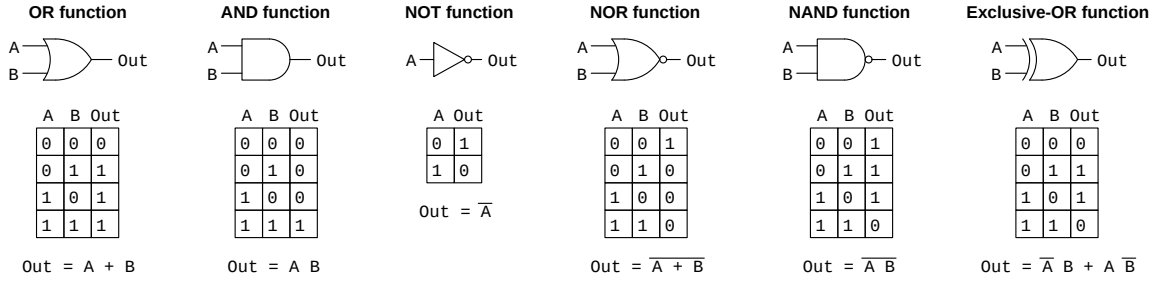
Binary adders are a fundamental building-block of Arithmetic Logic Units (ALUs), which perform numerical operations on binary numbers within microprocessor systems. Most readers of this document will do so using some digital electronic device, and rest assured each and every one of those devices will contain at least one ALU!

Chapter 3

Tutorial

3.1 Logic functions

Digital logic is the realm of "discrete" quantities having only two possible values, or "states": 1 and 0. From this simple idea springs forth the concept of logical *functions* where specific combinations of input signal states result in pre-defined output states. Several fundamental logic functions are shown in the following illustration, each function accompanied by a *truth table* declaring the output state for each possible combination of input states, as well as a *Boolean algebra expression* describing the function mathematically:



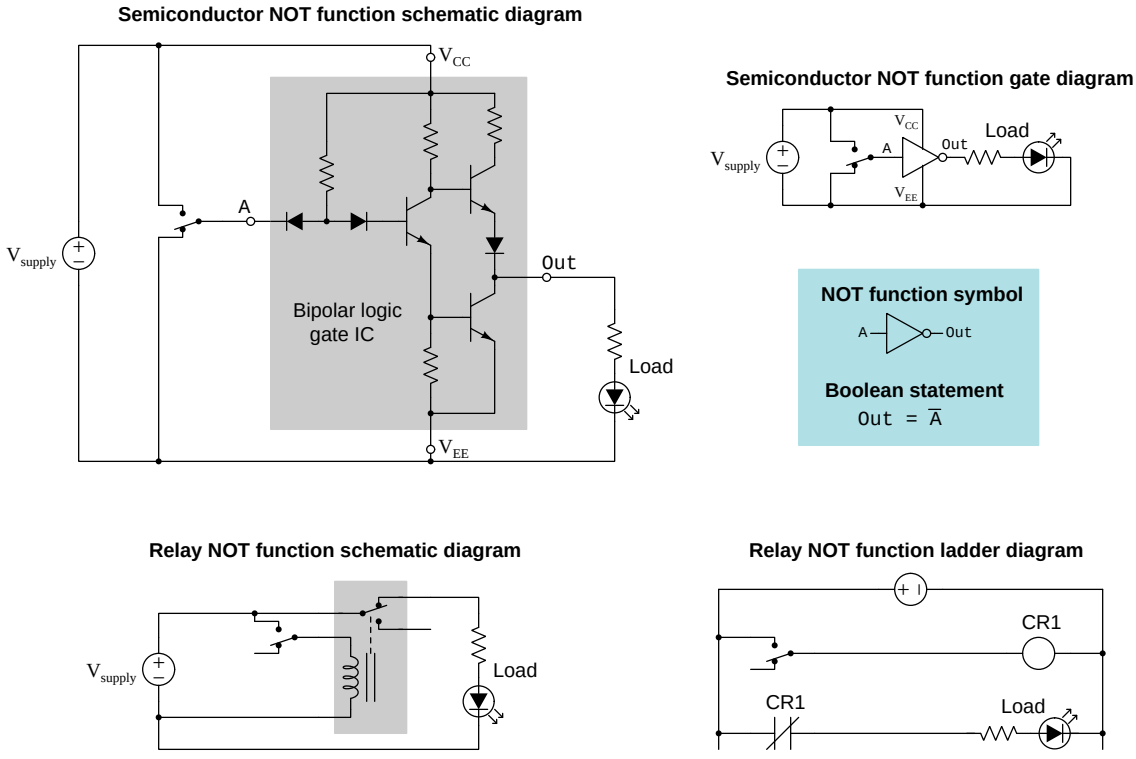
Although the use of arithmetic (e.g. $A + B$ for the OR function, AB for the AND function) may seem strange, it makes sense when you consider the limited values each discrete variable has. If each variable may only be a 0 or a 1, it makes sense, for example, that an AND function whose output is 1 only if all inputs are 1 is equivalent to multiplication, where the product is 1 only if all multiplied values are 1. Likewise, addition makes sense for the OR function up until 1 + 1 = 1, and even that makes sense once you realize there is no such thing as a value of "two" in the Boolean numbering system. An overhead bar symbol represents logical *inversion* or *complementation*, which flips the value to its opposite. Thus, $\overline{A}$ means the opposite[1] logical state of A, and $\overline{A + B}$ (NOR) represents a function with output states exactly opposite of $A + B$ (OR).

[1] When spoken, one generally says "A-bar" or "not-A" to represent the complement of A.

All of the two-input logic functions previously shown, with the exception of the Exclusive-OR (also called XOR), are available in versions having more than two inputs. A four-input OR function, for example, would have an expanded truth table with sixteen (2^4) rows, only the first of which has a 0 output state (with all four inputs in their 0 states); and a Boolean equivalent expression of Out $= A + B + C + D$.

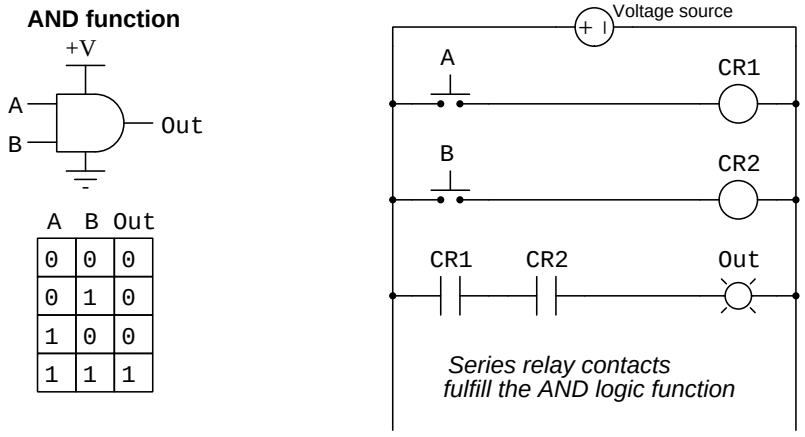
Electrical logic circuits use discrete voltage signals to represent 0 and 1 logical states. Typically, a "high" voltage value (at or near the positive power supply rail voltage with respect to ground) represents 1 and a "low" voltage value (at or near ground potential) represents 0. Logical functions take the form of transistor or relay networks in digital circuits, transistor-based logic circuit elements being called *gates* and relay-based logic being called *relay ladder logic*.

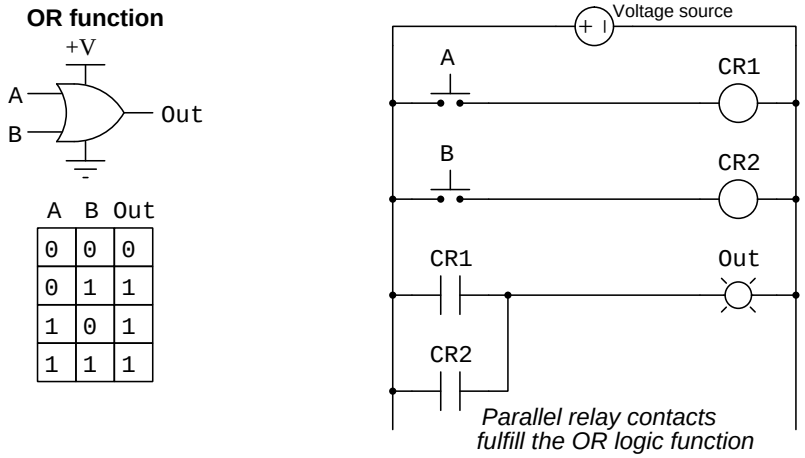
The NOT function, for example, may be constructed using bipolar junction transistors and packaged in an integrated circuit (IC), or alternatively it could manifest as an interconnection of electromechanical relays. Four diagrams below show how the NOT function may be implemented using either solid-state or relay technology, two of these diagrams use standard electronic schematic diagram symbols, while the other two use special symbols made for the purpose of simplifying digital diagrams:



Semiconductor technologies other than bipolar junction transistors (BJTs) may alternatively be used. A type of logic gate called *CMOS* using complementary N-channel and P-channel MOSFETs is also quite popular.

Basic logical functions such as AND and OR may be implemented using electromechanical relays just as they can using transistors. AND and OR functions in particular have direct relation to *series* and *parallel* contact connections, respectively. Please note that electrical power supply connections are typically omitted from these diagrams for simplicity, but are shown here in order to present a complete view of all required connections to make these logic systems functional:



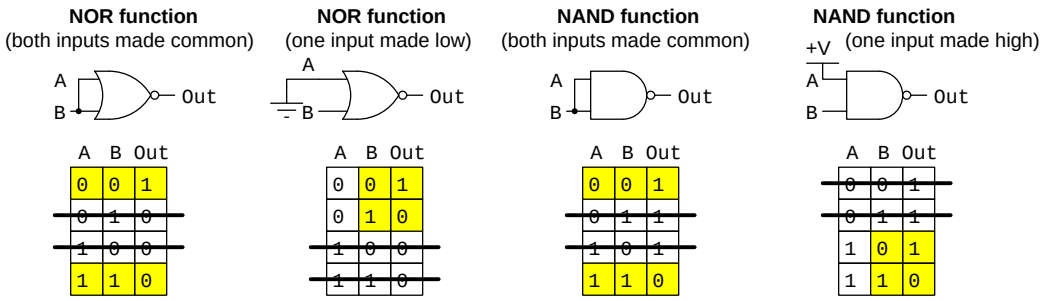


It is important to closely study the conventions of each diagram style, where we find similar or even identical symbols used to represent different things. A small circle, for example, refers to a terminal on an integrated circuit (IC) package, whereas on a gate diagram an identical circle represents logical negation (inversion, or complementation). A larger circle drawn as a component in a ladder diagram represents the coil of an electromechanical relay.

3.2 Universal logic functions

When we combine basic logical functions to make other (usually more complex) logical functions, it is called *combinational logic*. A simple and practical application of combinational logic is the use of one basic function to create another basic function. Two logical functions are classified as *universal* because any other logical function may be made from combinations of either type. Both NAND and NOR functions share this property of universality, meaning we may build up any logic function at all simply by connecting a sufficient number of NAND functions together, or a sufficient number of NOR functions together. The key to this universality is the ability of both NAND and NOR functions to operate as inverters (i.e. as NOT functions), and from this property we may combine multiple instances of each gate type to form any other.

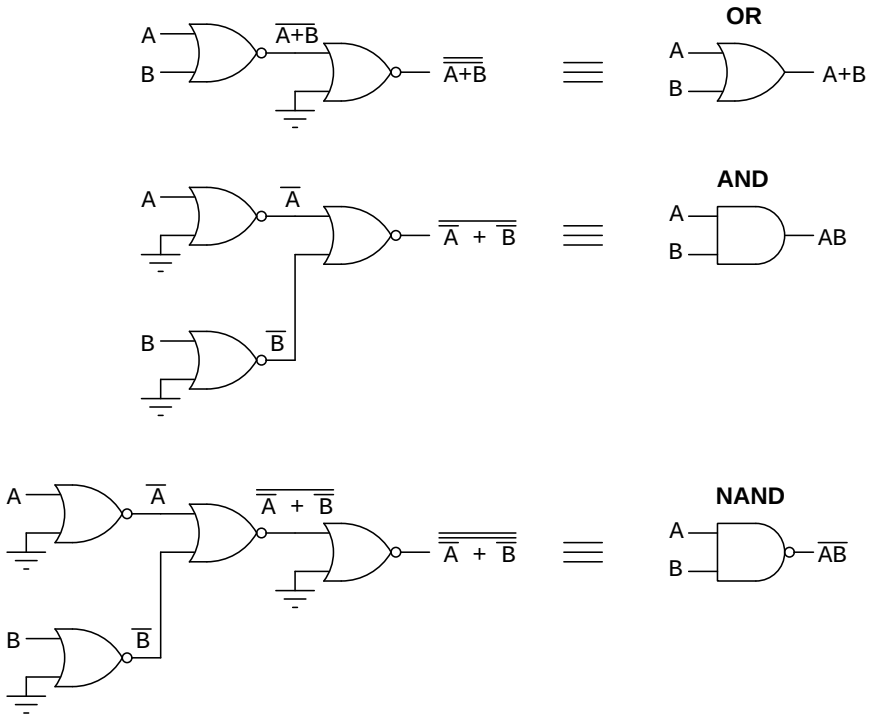
First, demonstrating how NAND and NOR gates may function as inverters, either by connecting both input terminals together to function as a single input, or by connecting the unused input terminal to a particular power supply rail to force its logical state all the time[2]:



An excellent *active reading* exercise is to take the time to annotate the logic diagrams presented in this (and future) Tutorials with various combinations of "1" and "0" input states, then reference truth tables for the respective functions to determine what the output states must be. For example, if we annotate the upper-left NOR diagram with a "0" state at its input we see that both NOR inputs must be "0" which according to the NOR truth table must yield a "1" output; annotating the same diagram with a "1" state at its input means both of the NOR inputs must be "1" as well which according to the NOR truth table yields a "0" output.

[2] Although from a strictly logical perspective it makes no difference whether a NAND or NOR gate is turned into an inverter (NOT gate) by connecting its input terminals together or by forcing the state of one of those inputs with a direct connection to a power supply rail, usually the latter method is preferred in actual circuit design. The reason for the latter preference has to do with the current sourcing or sinking demands of logic gate inputs: tying input terminals together means the driving device must service the current requirements of two inputs rather than just one, and additional loading degrades high-speed performance.

The following illustration demonstrates[3] how it is possible to use nothing but combinations of NOR gates to construct the other three basic logic functions (OR, AND, and NAND). As an aid to understanding, every signal line bears a Boolean expression describing its state:

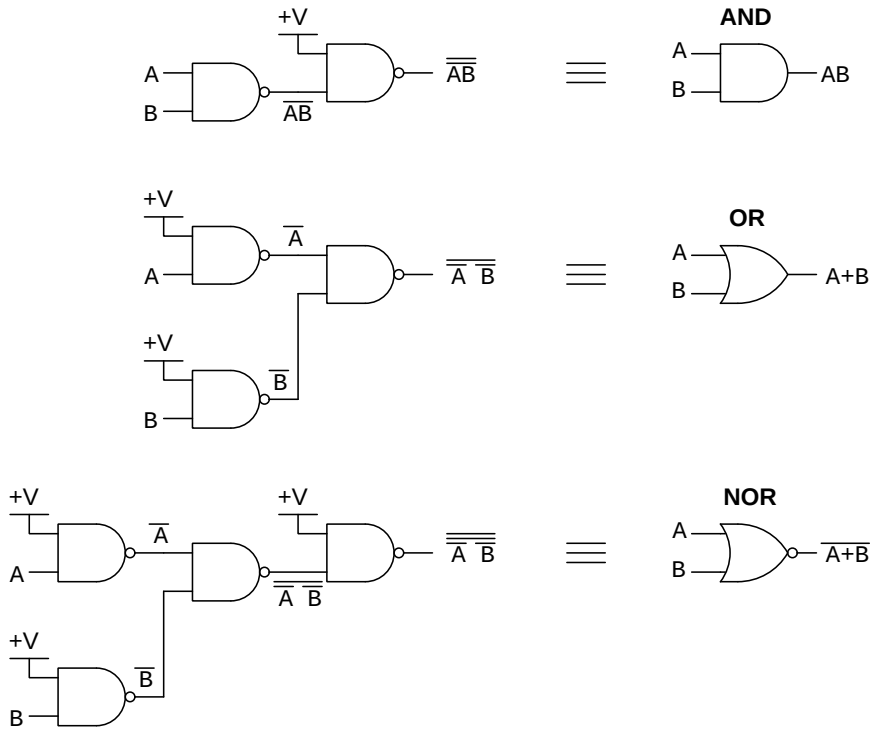


Upon first inspection it may not be clear why some of these NOR gate networks are logically equivalent to their basic equivalents shown on the right. The first example, turning a NOR into an OR by inverting its output, is simple enough because any double-negation cancels itself, and so $\overline{\overline{A + B}}$ is equivalent to $A + B$. The next example, turning a NOR gate into an AND gate by inverting its input signals, is not so obvious. They key to understanding this conversion is a theorem in Boolean algebra known as *DeMorgan's Theorem*, which states any negation bar spanning an arithmetic operation may be divided ("broken") and that arithmetic operation exchanged for the other[4] type. Following DeMorgan's Theorem, we may break the long bar over $\overline{\overline{A} + \overline{B}}$ and convert the addition into multiplication, resulting in $\overline{\overline{A}}\,\overline{\overline{B}}$, which then becomes AB according to the principle of double-negations canceling. Likewise, the next example where NOR gates become a NAND gate, involves first canceling double-bars to turn $\overline{\overline{\overline{A} + \overline{B}}}$ into $\overline{A} + \overline{B}$, at which point DeMorgan's Theorem shows us the equivalence between this statement and $\overline{AB}$.

[3] This illustration itself, of course, does not actually *demonstrate* the universality of NOR gates. In order for a true demonstration to be complete, one must observe the system operating as intended. For this, it is left as an exercise to the reader to perform "thought experiments" on these three logic circuits, imagining the input terminals in their various possible states and following through to the consequent output states based on the truth table of a NOR function.

[4] In standard Boolean algebra, the only two arithmetic operations are addition and multiplication.

This next illustration shows the same principle of universality with combinations of NAND gates rather than combinations of NOR gates:



Once again we see DeMorgan's Theorem in action, where a long complementation bar may be "broken" into shorter pieces with a corresponding change of arithmetic operation below the break (e.g. $\overline{\overline{A}\,\overline{B}}$ becomes $\overline{\overline{A}} + \overline{\overline{B}}$ which becomes $A + B$ after canceling the double-bars). This example, like the others, shows the power of Boolean algebra as a means of expressing logical functions: it may not be apparent by inspection of the diagram how combining three NAND gates results in an OR function, but a rule such as DeMorgan's Theorem lets us manipulate these symbols to prove that equivalence. In other words, Boolean algebra is a *tool* for understanding combinational logic.

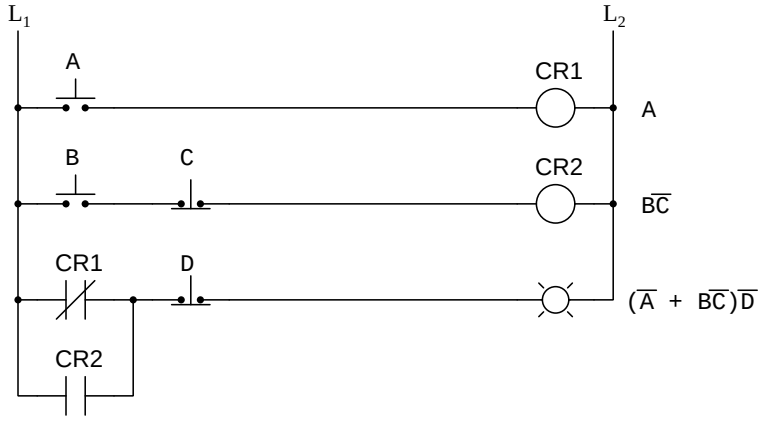
Combinational gate circuits may be created to implement any conceivable digital function. The examples shown so far merely illustrate one application, and that is the creation of basic logic functions using just one type of "universal" gate.

The practical procedure for deriving a Boolean expression from any combinational gate circuit diagram is apparent in both the NOR and NAND combinational logic examples: begin by labeling the inputs with unique letters (variables), then label the output terminals of each logic gate based on those inputs and the Boolean mathematical operation for each logic function (e.g. addition for OR, multiplication for AND, complementation for NOT), repeating this procedure through to the final output(s) of the circuit. Similarly, the procedure for determining the logic states within any combinational gate circuit diagram is to label all the given input states, then reference truth tables to determine the output state of each gate those inputs feed into; then repeat this procedure as

the output states of those gates feed into the input terminals of other gates. A truth table may be generated for any combinational function the same way: analyze all logic states within the circuit for every possible combination of input states, the number of rows in this truth table being equal to 2 raised to the power of the number of discrete inputs.

3.3 Combinational relay logic

Logic functions and Boolean algebra are abstractions, and as such are independent of physical form. This means the same principles apply to digital circuits built using different technology, for example electromechanical relays rather than semiconductor gates. Consider the following "ladder logic"[5] relay circuit shown below, using pushbutton switches for inputs and a lamp for the single output. We will define[6] a 1 state as *electrically conductive* (for a switch contact) or *energized* (for a load), and write a Boolean expression to the right of each "rung" in the "ladder" circuit. With these definitions, series-connected contacts implement the AND function, parallel-connected contacts implement the OR function, and normally-closed contacts represent the NOT function:



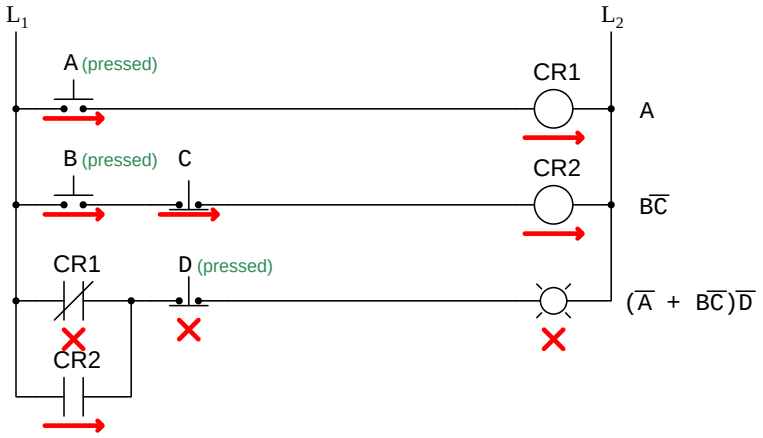
The state of relay coil CR1 is identical to the state of switch A's actuation (i.e. 1 = pressed switch = closed contact = energized coil), and so that coil's Boolean expression is simply A. In the next rung we have a normally-open B switch in series with a normally-closed C switch, and so coil CR2's expression is $B\overline{C}$. Relay contact CR2 is normally-open and so carries with it the same state as coil CR2 ($B\overline{C}$), but contact CR1 is normally-closed and so it inverts or complements its coil's state ($\overline{A}$). Those two relay contacts are connected together in parallel, and so their combined expression is the Boolean *sum* $\overline{A} + B\overline{C}$. Connected in series with those paralleled relay contacts is a normally-closed pushbutton switch $\overline{D}$, and so the expression of the lamp in the last rung is the Boolean *product* $(\overline{A} + B\overline{C})\overline{D}$.

Similar to gate circuits, the procedure for deriving a Boolean expression from any combinational relay circuit diagram begins by labeling the inputs with unique letters (variables), then labeling each relay coil based on the Boolean mathematical operation for each connection type (e.g. addition for parallel, multiplication for series, complementation for normally-closed), repeating this procedure through to the final load(s) of the circuit.

[5] The term "ladder logic" refers to the ladder-like orientation of the components and connecting wires in the diagram. Each horizontal row is called a *rung*, while the two vertical lines (representing the poles of a voltage source) are called *rails*. The voltage source symbol is usually omitted for simplicity, with the labels L1 and L2 representing these "line voltage" rails not unlike V_{CC} and V_{EE} in a bipolar logic gate circuit.

[6] This would be *positive logic* as applied to a relay circuit, analogous to defining 1 as a "high" voltage state and 0 as a "low" voltage state for a semiconductor gate circuit.

Logic states within a relay ladder logic circuit are easy to track using simple annotations to represent the electrical status of each contact and each load. I recommend a line or arrow drawn near the component to represent *continuity* (switch) or *energization* (load), and an "X" symbol to represent *non-continuity* (switch) or *de-energization* (load). For example, the previous ladder logic circuit is shown here annotated for a condition where switches A, B, and D are pressed but C is released:



Switch contacts A and B are both closed (lines drawn) because they are normally-open switches and are both being pressed. Switch C is closed (line drawn) because it is normally-closed and is not being pressed. Switch D is open (X drawn) because it is normally-closed and is being pressed.

Relay coils CR1 and CR2 are both energized (lines drawn) because there is electrical continuity through their rungs. Relay contact CR1 is open (X drawn) because it is normally-closed and actuated by its energized coil. Relay contact CR2 is closed (line drawn) because it is normally-open and actuated by its energized coil.

Whether the contact in question is a pushbutton switch or a relay contact, its electrical status is a function of its *normal* type and its actuation state. Always remember that the "normal" condition for any electrical switch is its state when at rest (i.e. no actuating stimulus applied), and that electrical contact symbols are always drawn[7] in their "normal" (resting) states.

[7] This is why one should never draw a slash mark through a relay contact symbol as a means to annotate closure, because this unnecessarily confuses the disparate concepts of a switch's *current* status with its *normal* status. Sadly, this is an all-too-common habit both of students and of (some) working professionals, and it represents a temptation to conflate two different concepts in a misguided attempt to simplify the task of relay circuit analysis. A relay contact drawn with a diagonal slash through it is a *normally*-closed contact, and not necessarily in its closed state at the time of analysis.

3.4 Boolean expressions into circuits

At times it is necessary to design a digital circuit to implement a given Boolean expression. In such cases the procedure is simply to step through the Boolean expression using proper algebraic order-of-operations[8], applying the following equivalent circuit structures to the Boolean operations:

- Boolean **addition** is equivalent to the *OR* function, also equivalent to *parallel* switch contacts
- Boolean **multiplication** is equivalent to the *AND* function, also equivalent to *series* switch contacts
- Boolean **inversion** is equivalent to the *NOT* function, also equivalent to *normally-closed* switch contacts

For example, suppose the Boolean expression we needed to implement is as follows:

$$\text{Output} = D(A\overline{B} + \overline{A}C)$$

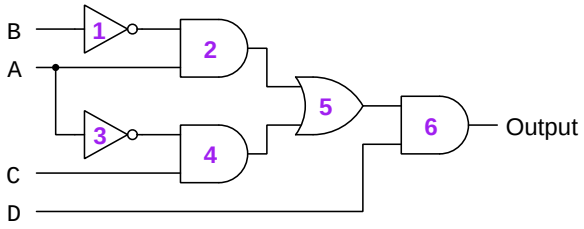
The proper order of operations for evaluating this expression is shown here:

1. Invert the value of B
2. Multiply $\overline{B}$ by A
3. Invert the value of A
4. Multiply $\overline{A}$ by C
5. Add $A\overline{B}$ and $\overline{A}C$
6. Multiply $A\overline{B} + \overline{A}C$ by D

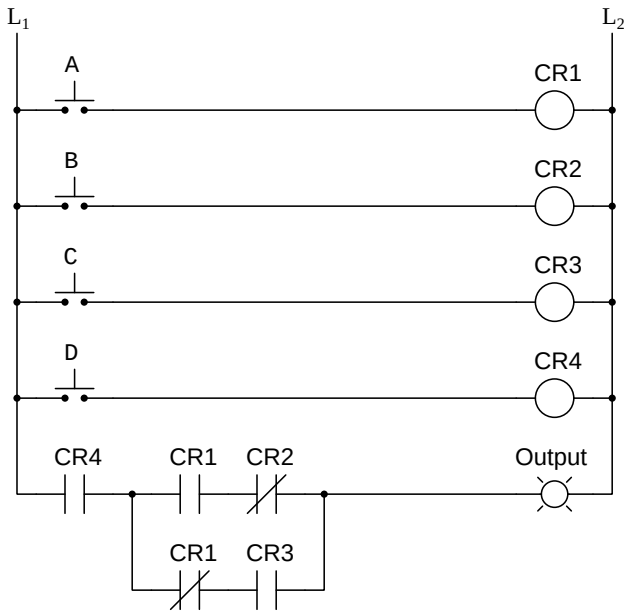
[8] Beginning algebra students typically learn a mnemonic such as *PEMDAS* to remember which operations take precedence over others. In the case of *PEMDAS*, it is first any operation enclosed within Parentheses, followed by Exponents, then by Multiplication and Division, and finally by Addition and Subtraction. Boolean order-of-operations is simpler because there is no such thing as an exponent, division, or subtraction. A comparable mnemonic for Boolean expressions is *PIMA*: Parentheses followed by Inversion (complementation or negation), followed by Multiplication, followed by Addition.

Building a gate circuit to implement $D(A\overline{B} + \overline{A}C)$ is as simple as following this ordered list, using a NOT gate for inversion, an AND gate for multiplication, and an OR gate for addition. Note the number labels for each of the logic gates in the schematic diagram, showing which logic gate implements which step we followed with mathematical order-of-operations.

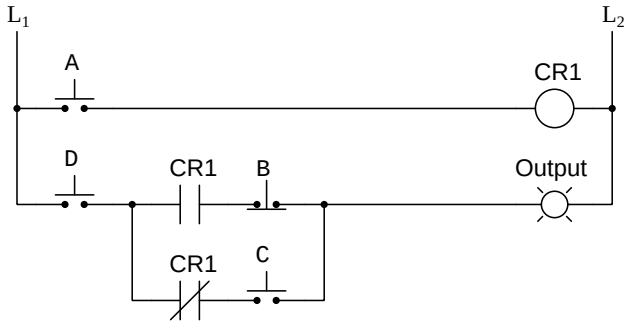
1. Invert the value of B – *Connect input B to a NOT gate*
2. Multiply $\overline{B}$ by A – *Connect the NOT gate's output to one input of an AND gate, with the other AND gate input connected to input A*
3. Invert the value of A – *Connect input A to a second NOT gate*
4. Multiply $\overline{A}$ by C – *Connect the second NOT gate's output to one input of a second AND gate, with the other AND gate input connected to input C*
5. Add $A\overline{B}$ and $\overline{A}C$ – *Connect the two AND gate outputs to the inputs of an OR gate*
6. Multiply $A\overline{B} + \overline{A}C$ by D – *Connect the OR gate's output to one input of a third AND gate, and input D to the other input of this AND gate*



Much the same is true for creating a relay logic circuit from a Boolean expression: we use normally-closed contacts to invert, series-connected contacts for multiplication, and parallel-connected contacts for addition. Again, implementing the Boolean expression $D(A\overline{B} + \overline{A}C)$, but this time in relay ladder logic form:



A useful strategy for logic circuits with mechanical switch inputs to minimize the number of electromechanical relays required is to only include a relay when a variable appears more than once in the Boolean expression. For the expression $D(A\overline{B} + \overline{A}C)$ only A requires a relay. All other inputs are simply switches wired NO or NC as needed:



If the given Boolean expression contains no repeated variables, the circuit may be constructed entirely of switches with no relays necessary!

3.5 Truth tables into circuits

When designing a digital logic circuit to perform some practical function, we usually begin the design process by sketching a truth table describing exactly what we wish this circuit to do. The truth table is where we declare which input state combinations will cause the output to achieve a certain state. Once we've defined our truth table, we need to find some way to translate this abstract representation into a real working circuit diagram.

Fortunately, Boolean expressions make this process relatively straightforward. The following subsections describe slightly different methods of developing a Boolean expression from any arbitrary truth table, and then from that Boolean expression we may create a logic gate or relay "ladder logic" circuit to fulfill the function (as described in earlier sections).

The first technique utilizes a Boolean form called *sum-of-products* (SOP), consisting of a Boolean expression that is a sum of terms comprised of multiple variables multiplied together. For example, $ABC + DEF$ is a sum-of-products where ABC and DEF are both products that are being added (summed) together. The basic circuit form used to implement any SOP expression is a set of AND gates (or series-connected switch contacts) for each product and an OR gate (or parallel connection) for the sum. The basic SOP technique works very well when a minority of the truth table's output values are true (1).

The second technique is a variation of sum-of-products method optimized to work best when a minority of the truth table's output values are false (0).

The third technique uses a Boolean form called *product-of-sums* (POS), consisting of a Boolean expression that is the product of several sums contained in parentheses. For example, $(A + B + C)(D + E + F)$ is a product-of-sums where $(A + B + C)$ and $(D + E + F)$ are both sums that are being multiplied together. The basic circuit format for a POS expression is a set of OR gates (or parallel-connected switch contacts) for each sum and an AND gate (or series connection) for the product. The POS technique works well for truth tables where a minority of the output values are false (0).

In full disclosure, *any* of these techniques may be used to translate a given truth table into a correct circuit. What makes one of them better than another for any specific problem is really the balance of true (1) versus false (0) states in the truth table's output column. That is to say, some techniques generate Boolean expressions with fewer terms than others for a given truth table, and therefore translate that truth table into equivalent circuits with fewer components than others.

For students new to this topic I recommend mastering the first two techniques, both based on sum-of-products (SOP) Boolean expressions. The product-of-sums (POS) technique works well, but can be confusing because nearly everything about it seems "backwards" compared to the basic SOP technique. If you master the SOP-based techniques – which I call "Sum of Products" and "Negative Sum of Products" respectively – you will be able to translate any arbitrary truth table into a reasonably efficient circuit with a minimum of confusion.

3.5.1 Sum of Products

The number of products in any *sum-of-products* Boolean expression is equal to the number of "1" output states listed in the truth table. Take the following truth table as an example:

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Only two rows in this table contain "1" output states, and so our sum-of-products expression may be described in the following terms:

The Output will be 1 when $A = 0$ *and* $B = 0$ *and* $C = 1$

. . . *or* . . .

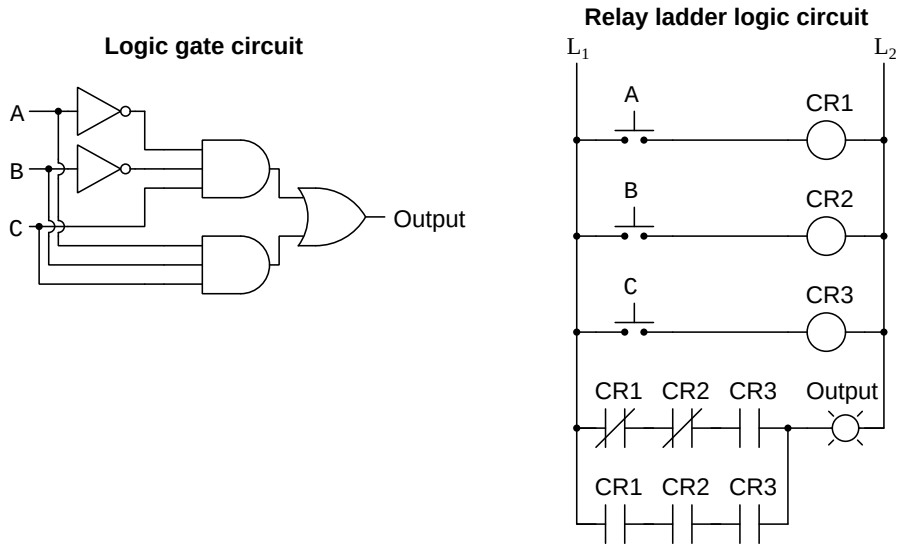
when $A = 1$ *and* $B = 1$ *and* $C = 1$

The italicized words *and* and *or* are intentionally emphasized for the purpose of associating with Boolean operations of multiplication and addition, respectively. Writing this same statement in mathematical form:

$$\text{Output} = \overline{A}\,\overline{B}C + ABC$$

The product $\overline{A}\,\overline{B}C$ expresses the "truth" of the second row, where $A = 0$ *and* $B = 0$ *and* $C = 1$, the individual variables complemented or uncomplemented as necessary to make the product equal 1 for those conditions. The product ABC expresses the "truth" of the eighth row, where $A = 1$ *and* $B = 1$ *and* $C = 1$. The overall function is the sum of these two products; i.e. the function will output a 1 if either of these products are "true". Each of the two products represents a three-input AND function, while the sum of those two products represents a two-input OR function.

Taking this sum-of-products expression (Output = $\overline{A}\,\overline{B}C + ABC$) and converting into both logic gate and relay ladder logic circuits as previously described:



The structure of these two circuits reflects the structure of the sum-of-products Boolean expression. Note how the product $\overline{A}\,\overline{B}C$ is evident in the upper AND gate of the logic gate circuit, as well as in the second rung from the bottom of the ladder logic diagram; also how the product ABC is represented by the lower AND gate and also by the lowest rung. The OR gate (and the paralleled network of the lowest two rungs) embodies the sum between the two products.

If we consider the fact that the number of products in a sum-of-products Boolean expression is equal to the number of truth table rows with a "1" output state, it becomes apparent that both the sum-of-products expression and the equivalent circuit(s) becomes larger as the number of "1" output states in the truth table increases. In other words, the more "1" output states in a truth table, the larger the sum-of-products solution will be. For truth tables containing more "1" output states than "0" output states, this fact can be problematic.

In answer to this challenge, multiple techniques exist to derive Boolean expressions from truth tables. The best technique for any application depends on the nature of the truth table, in particular whether the function has a super-majority of "1" or "0" output states.

3.5.2 Negative Sum of Products

One such alternative approach designed to work best for truth tables containing a minority of "0" output states, is to write a sum-of-products expression while thinking in "negative" terms, identifying Boolean products describing truth table rows with "0" output states rather than "1" output states. The intrinsic problem-solving technique here is to *simplify the problem*, imagining the truth table's majority-1 output states to be majority-0 instead and then writing an SOP expression for that inverted table using the previously-described technique. To make the SOP expression correct for the original table, one must place a long inversion (complement) bar covering all terms. Take the following truth table as an example:

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Only two rows in this table contain "0" output states, and so our sum-of-product expression may be described in the following terms:

The Output will be 0 when $A = 0$ *and* $B = 1$ *and* $C = 1$

. . . *or* . . .

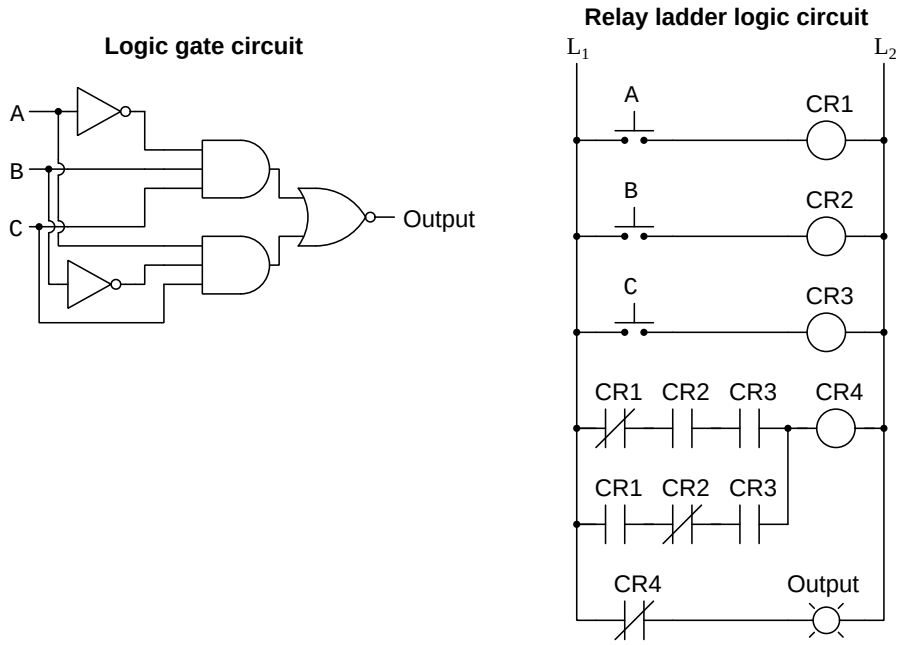
when $A = 1$ *and* $B = 0$ *and* $C = 1$

Once again the italicized words *and* and *or* intentionally emphasize the Boolean operations of multiplication and addition, respectively. Writing this same statement in mathematical form:

$$\text{Output} = \overline{\overline{A}BC + A\overline{B}C}$$

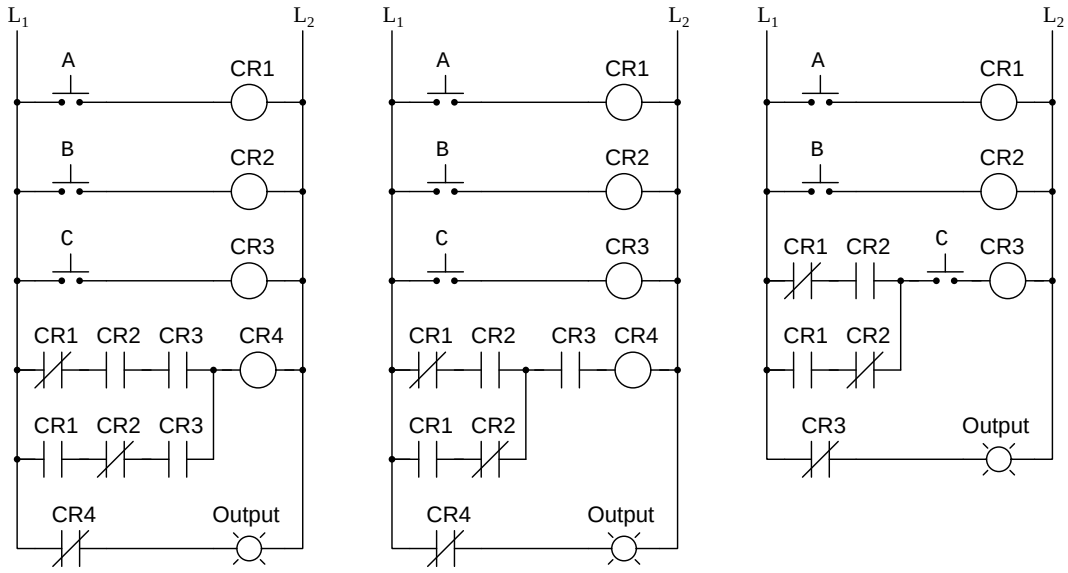
The product $\overline{A}BC$ expresses the "negative truth" of the fourth row, where $A = 0$ *and* $B = 1$ *and* $C = 1$. The product $A\overline{B}C$ expresses the "negative truth" of the sixth row, where $A = 1$ *and* $B = 0$ *and* $C = 1$. By "negative truth" we mean that the overall function is the *inverted* sum of these two products; i.e. the output is 0 if either of these products equal 1. In other words, if either $\overline{A}BC$ or $A\overline{B}C$ is true, the output will be false. Each of the two products represents an AND function, while the final sum and negation represents a NOR function.

Taking this sum-of-products expression (Output = $\overline{\overline{A}BC + A\overline{B}C}$) and converting into both logic gate and relay ladder logic circuits as previously described:



Once again we see how the circuits' topologies reflect the structure of the sum-of-products Boolean expression. Note how the product $\overline{A}BC$ is evident in the upper AND gate of the logic gate circuit, as well as in the third rung from the bottom of the ladder logic diagram; also how the product $A\overline{B}C$ is represented by the lower AND gate and also by the second rung from the bottom. The NOR gate implements the inverted sum of the two products in the gate circuit. In the relay circuit, this final inversion requires a fourth relay with a normally-closed contact, its coil powered by the positive sum (parallel network) of two rungs.

Close inspection of the relay ladder logic circuit for the Boolean expression $\overline{\overline{A}BC + A\overline{B}C}$ reveals potential for simplification. Notice how the two normally-open CR3 contacts are redundant to each other, and how we may implement the exact same logical function by eliminating one of those CR3 contacts. Going further, we may eliminate one relay altogether, since C now appears only once in the Boolean expression and therefore relay CR3 contributes nothing to the circuit. The following set of diagrams show the original circuit (left) and two levels of simplification (middle and right):



The first simplification – eliminating one of the CR3 relay contacts – could have been done (first) to the Boolean expression prior to drawing any circuit diagram. If we examine the two products underneath the long complement bar, we see both of them have a common C variable. This means we may *factor* C out of the two products as demonstrated here:

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

The simplified expression is directly implemented in the middle relay ladder logic diagram shown above. The far-right diagram simplification comes not from Boolean algebra, but rather from the realization that relay CR3 just wasn't necessary.

Boolean algebra offers several identities and properties useful[9] for simplifying long expressions. A full exploration of these techniques will be left to another learning module.

[9] Interestingly, while simplified Boolean expressions require fewer circuit components to implement, which typically decreases construction cost and increases operational reliability, Boolean simplifications lend little benefit to *programmed* logic functions, whether in a microcontroller or in a PLC (Programmable Logic Controller). AND, OR, and NOT functions programmed in code cost nothing either in capital investment or in non-reliability, and so a large expression programmed into a digital controller works just as well as an equivalent reduced expression. In fact, there are circumstances where the simplified version of a Boolean logic function actually *makes less sense* to anyone examining the code than the raw expression generated directly from the truth table, in which case Boolean reduction does more harm than good!

3.5.3 Product of Sums

Another alternative approach well-suited to truth tables having more "1" output states than "0" is to write a *product-of-sums* Boolean expression (e.g. $(A + B + C)(D + E + F)$) rather than a sum-of-products (e.g. $ABC + DEF$). Here, the strategy is to identify those "0" input conditions leading to an output value of 0. As before, we will begin with an example truth table, our goal being to derive both logic gate and relay ladder logic circuit implementations of the arbitrary function represented by the truth table:

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

The "0 truth" sum representing the fifth row is $\overline{A}+B+C$, the individual variables complemented or uncomplemented as necessary to yield a sum of 0 when $A = 1$ and[10] $B = 0$ and $C = 0$. The "0 truth" sum representing the eighth row is $\overline{A} + \overline{B} + \overline{C}$. Multiplying[11] these two sums to arrive at a product-of-sums expression for the truth table:

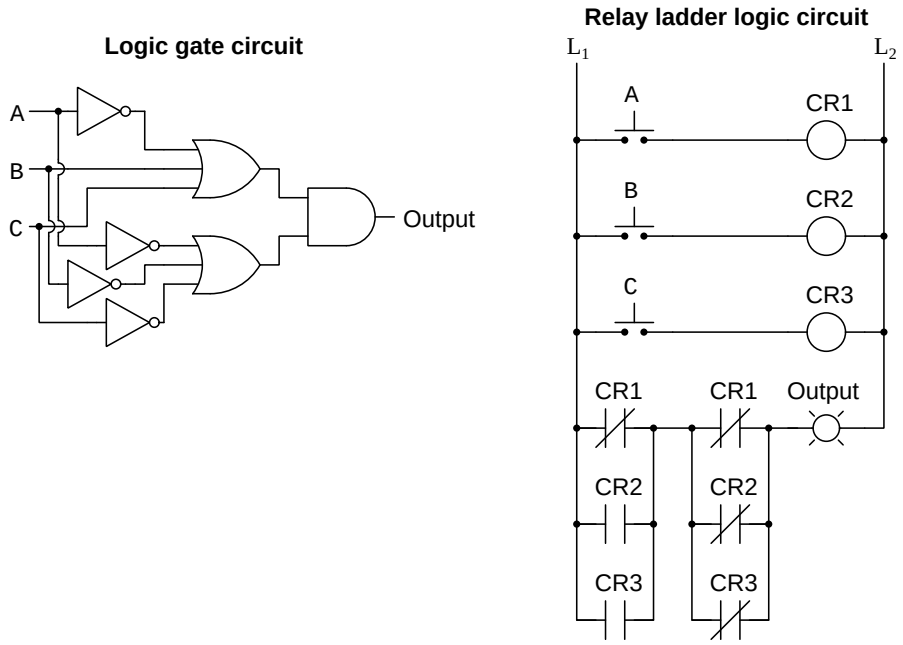
$$(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})$$

Implementing this Boolean product-of-sums expression in circuit form requires either OR gates or parallel relay contacts for the sums (addition), and AND gates or series relay contacts for the product (multiplication).

[10] At first it may seem incorrect to use Boolean *addition* (which is normally equivalent to an OR function) to associate particular states of A, B, *and* C, but this is actually legitimate because we are thinking in "negative" terms. Recall that a three-input OR function outputs a 1 if $A = 1$ *or* $B = 1$ *or* $C = 1$; but it is also true to say that an OR function outputs a 0 if $A = 0$ *and* $B = 0$ *and* $C = 0$. Therefore, when thinking in terms of a "0" result, it is fair to use Boolean addition (i.e. an "OR" function) to identify some simultaneous set of "0" input states. This is actually an example of DeMorgan's Theorem in action: inverting both the inputs and output of a fundamental logic function changes its type from OR to AND or vice-versa.

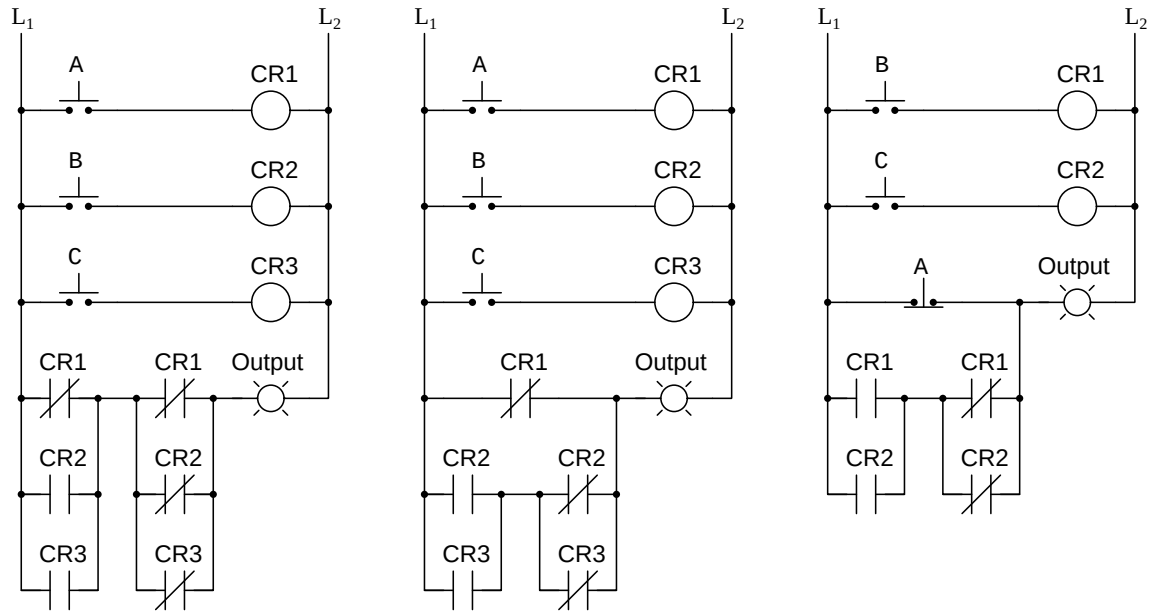
[11] The reason we choose Boolean *multiplication* to piece these two sums together is because we want the output to be 0 if either sum is equal to 0. Again, this may seem wrong to use multiplication to represent an "or" condition, but it works because we are thinking in "negative" terms, and an AND function (multiplication) always has a 0 output if *any* of its inputs are 0.

Both equivalent logic gate and relay ladder logic circuits for the product-of-sums expression $(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})$ are shown in the following diagrams:



As with the previous example, a close inspection of the relay ladder logic circuit reveals potential for simplification. Note how the two normally-closed CR1 relay contacts are redundant to each other, and may be replaced by a single normally-closed CR1 contact passing power to the Output lamp: any time pushbutton A is unpressed, both CR1 contacts will be in their closed (resting) states, guaranteeing energization of the load. From an electrical perspective, it makes no sense to have two series-connected relay contacts with the exact same state doing what one contact could do by itself. That simplification leads to the next, where we realize with only one CR1 contact in the circuit we may simply replace it with a normally-closed "A" pushbutton switch and dispense with one relay.

The following diagrams show these three versions of the circuit, the original (left), the first simplification (center), and the simplest version (right):



Mathematically demonstrating *why* these three circuits are functionally equivalent is much more difficult to do with a product-of-sums expression than for a sum-of-products expression, which is a strong argument against the product-of-sums approach of converting a truth table into an equivalent circuit and simplifying through Boolean algebraic techniques.

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 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] algorithms. 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.

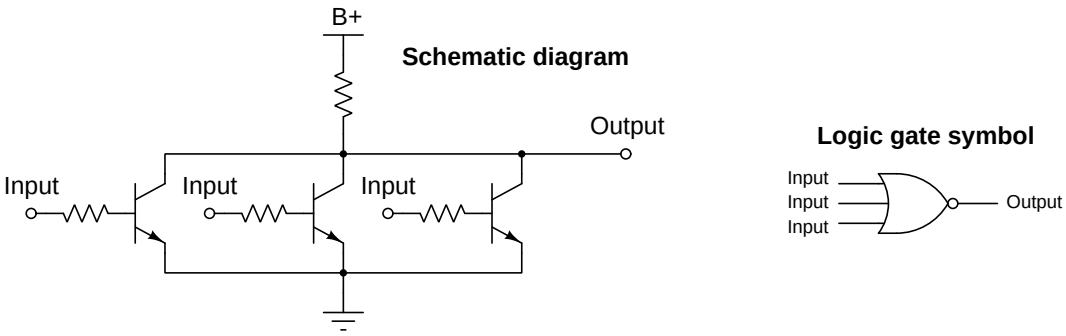
On page 11 of his paper, Shannon neatly summarizes and compares relay switching logic with Propositional Logic:

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.

4.2 NASA's Apollo Guidance Computer

The digital computer used for guidance functions in the 1960's era Apollo spacecraft (the one used to transport the first humans to Earth's Moon) built by NASA used bipolar transistor logic, consisting almost entirely of NOR logic gates. The schematic diagram for each three-input NOR gate was as follows, along with its corresponding logic gate symbol:



Note how the positive power supply terminal is labeled *B+*, an anachronistic reference to the positive terminal of a high-voltage *battery* (hence the letter "B") used to power vacuum tube circuits. Based on the knowledge that bipolar transistors are normally "off" devices, and require the base-emitter junction to be forward-biased in order to turn "on", we can tell if any input goes to a high state (i.e. connected to the positive rail of the DC power source), that respective NPN transistor will turn on and bring the output terminal's potential down (nearly) to ground. In other words, *any high input forces the output to be low*: the very definition of a NOR function.

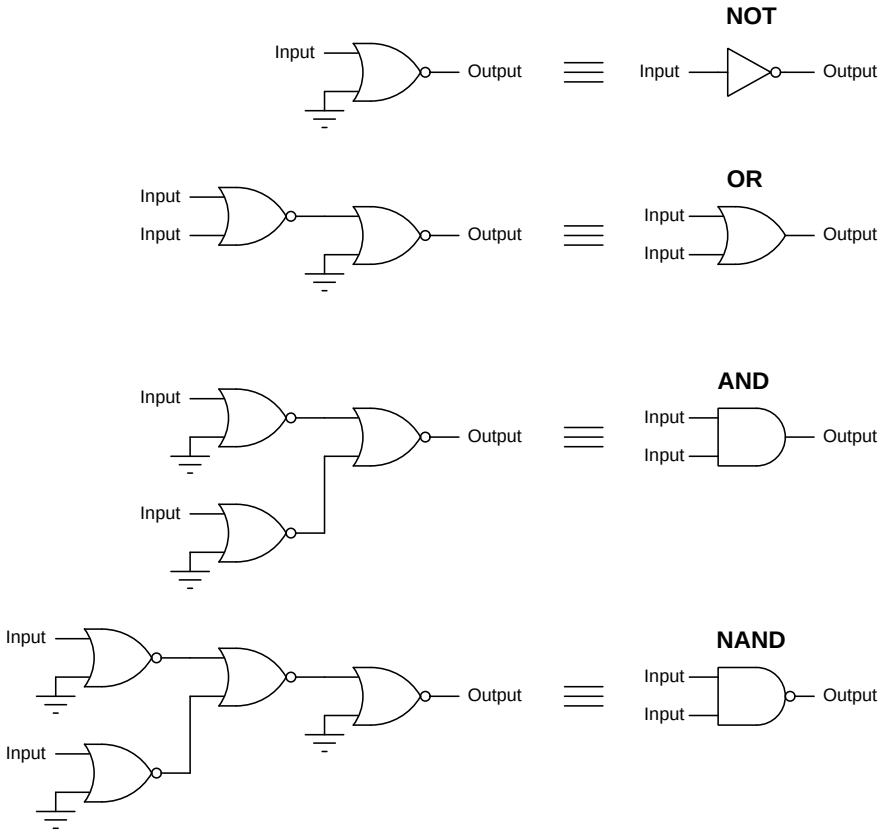
The three transistors can only *sink* current, and therefore this NOR gate's sourcing capability is limited by the resistor between the three collectors and the B+ power supply terminal. In other words, this NOR gate had a significantly greater current-sinking rating than its current-sourcing rating.

A rather short technical document entitled "A Case History Of The AGC Integrated Logic Circuits" describes how nearly the entire computer consisted of these three-input NOR logic gates:

> The standardization approach, which is particularly adaptable to digital computers, has been demonstrated with the Polaris flight computer and extended with integrated circuits to the Apollo Guidance Computer. Both computers were designed to use a three input NOR Gate as the only logic element. All logic functions are generated by interconnecting the three input NOR Gate with no additional logic blocks, resistors, or capacitors. At first glance, it appears that using only one type of logic block greatly increases the number of blocks required for the computer. But, by judiciously selecting and organizing the logic functions it is quickly apparent that few additional blocks are necessary. The few additional units required are greatly counterbalanced by the increased reliability gained during both the manufacturing of components and fabrication of the components into modules. [page 3]

It is possible to construct any digital logic function using nothing but NOR gates, because the NOR gate is one of two universal logic gate types (NAND being the other). The key to gate universality is the ability to function as an inverter (i.e. the NOT function) because inverting the input(s) and/or output of any logic function makes possible the transformation of that logic function into all other types. Any NOR gate will function as an inverter if the unused input(s) are fixed to "high" (1) logic states, the one remaining input controlling the gate's output. With a NOR gate such as the type used by NASA to build the Apollo Guidance Computers, the unused inputs may simply be left floating.

The following illustration demonstrates[1] how it is possible to use nothing but NOR gates to construct the other four basic logic functions (NOT, OR, AND, and NAND):

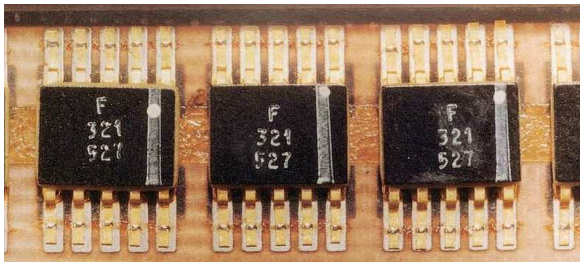


[1] This illustration itself, of course, does not actually *demonstrate* the universality of NOR gates. In order for a true demonstration to be complete, one must observe the system operating as intended. For this, it is left as an exercise to the reader to perform "thought experiments" on these four logic circuits, imagining the input terminals in their various possible states and following through to the consequent output states based on the truth table of a NOR function.

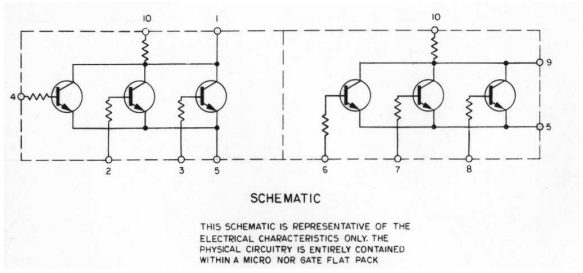
This next passage from the NASA document explains why NOR gates were chosen rather than NAND, and gives some technical specifications for the integrated circuits:

> The logic element utilized in the Apollo Guidance Computer is the three input NOR Gate as shown in Fig. 1. At the time that the decision was made to use integrated circuits, the NOR Gate, as shown, was the only device available in large quantities. The simplicity of the circuit allowed several manufacturers to produce interchangeable devices so that reasonable competition was assured. Because of recent process development in integrated circuits, the NOR Gate has been able to remain competitive on the basis of speed, power and noise immunity. This circuit is used at 3 V and 15 mW, but is rated at 8 V and 100 mW. Unpowered temperature rating is 150 ^{o}C. [page 4]

These NOR gate integrated circuits were enclosed in "flatpack" packages and soldered into printed circuit board assemblies. In the following photograph[2] we see three of these "flatpack" NOR gate ICs soldered into place:



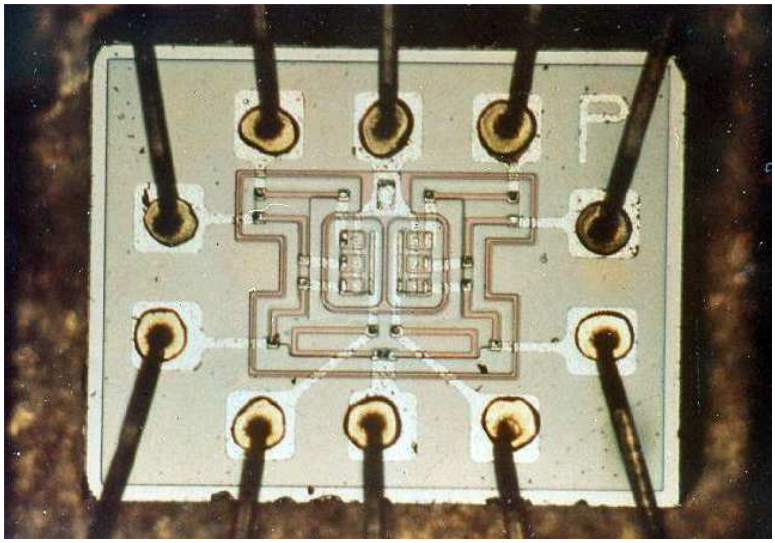
Each of these ten-terminal "flatpacks" contained two NOR gates, as shown in this NASA schematic[3]:



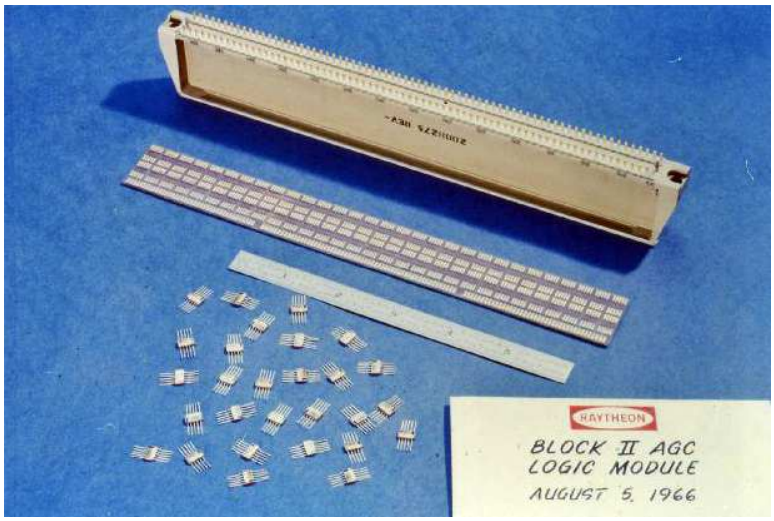
Note how, just as with modern multi-gate ICs, some of the terminals on this early example of an integrated circuit are shared in common with each of the internal gates. In this case we see terminal 10 (B+) common to both NOR gates, as well as terminal 5 (ground).

[2] This image was cropped from a public-domain photograph made courtesy of Grabert who posted it to Wikipedia.
[3] Another public-domain photograph courtesy of Grabert.

Here we see a close-up view[4] of the silicon wafer inside one of these dual-NOR gate IC packages:

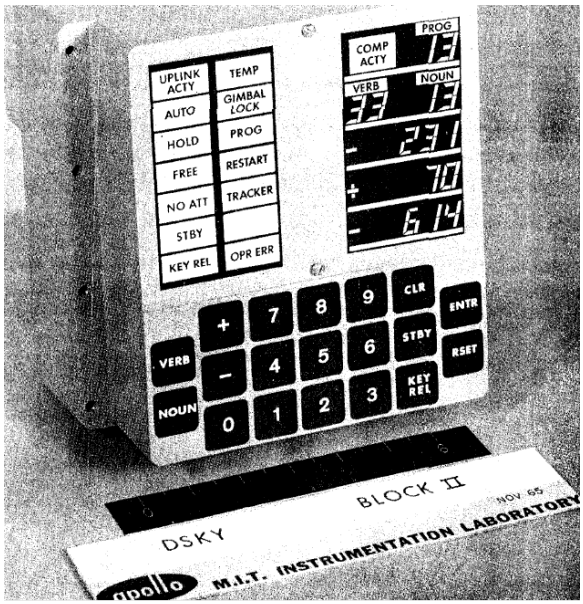


In another photograph[5] of the Apollo guidance computer assemblies, we see several of these NOR gate ICs lying on a table next to one of the module assemblies of the computer called "Block 2 AGC Logic":

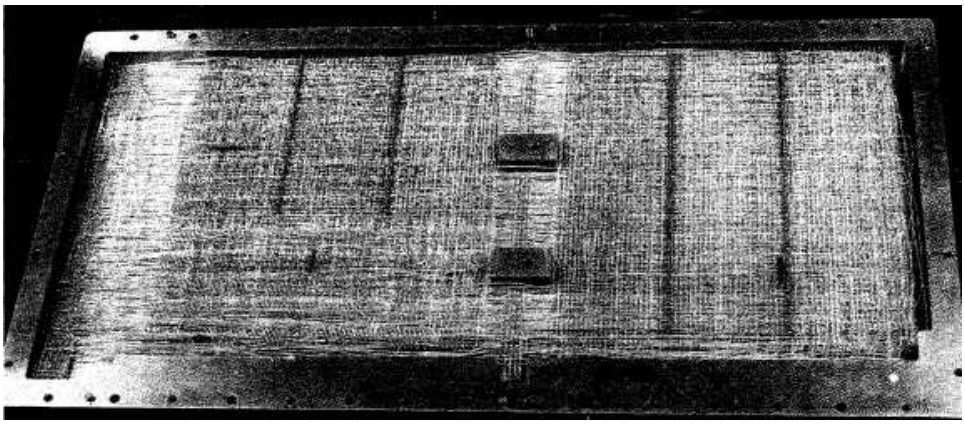


[4] Another photograph courtesy of Grabert, graciously released into the public domain.
[5] Another photograph courtesy of Grabert, graciously released into the public domain.

A photograph showing the operator console for this digital computer appears in the following photograph, from page 12 of the NASA *Case History* document:



Interconnections between NOR gates were quite extensive in this computer, the connections made by *wire-wrapping* on the side of the printed circuit board opposite of the ICs themselves. In other words, the ICs were soldered to one side of the circuit board but wire-wrap pins protruding through the other served as connection points for the wrapped wires connecting gates to each other. The following photograph, taken from page 11 of the NASA document shows wiring on the underside of the computer chassis. The quality of this photograph is too poor to see anything but a dense field of wire-wrap terminal pins and courses of thin wires interconnecting those pins:



Chapter 5

Derivations and Technical References

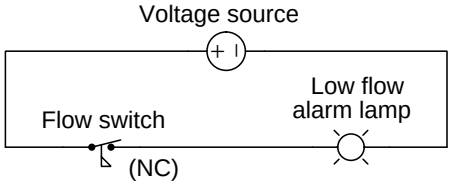
This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

5.1 Normal status of a switch contact

An perennial source of confusion among students new to electric switching circuits is the use of the word *normal* to refer to an electrical switch's default state. Switches, of course, are discrete devices capably only of two definite states: *open* (i.e. no electrical continuity) and *closed* (i.e. electrical continuity). *Toggle* switches are constructed in such a way that they may latch in either of these two states, which means they have no default condition. A great many switch types, however, are designed with a spring-return mechanism or equivalent functionality to make the switch return to a certain default state in the absence of any external stimulus. **This is called the "normal" state of the switch: its electrical state when at rest.**

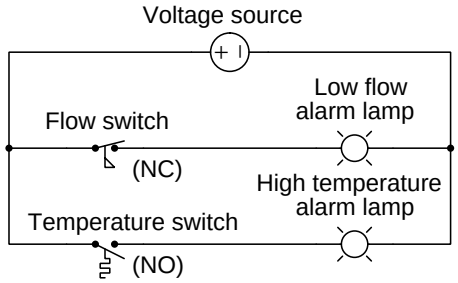
Where this becomes confusing is in applications where such a switch is *typically* found in an actuated state, such that ordinary operating conditions for the circuit maintain that switch in its *non-normal* state. Colloquial use of the word "normal" is synonymous with "typical" which makes it possible for someone to see a switch's "normal" status and mistakenly think this refers to its state in the circuit's normal operation rather than meaning its "normal" status as defined by its manufacturer.

A brief illustration is helpful here. Consider a *flow switch* used to detect the presence of liquid coolant flow through a pipe, carrying coolant to an operating engine. Engines, of course, dissipate heat as they run, and so a continuous flow of coolant to the engine is critical for maintaining safe operating temperature. A simple diagram shows how this flow switch would be connected to a warning lamp to alert personnel of any interruption in coolant flow to the engine:



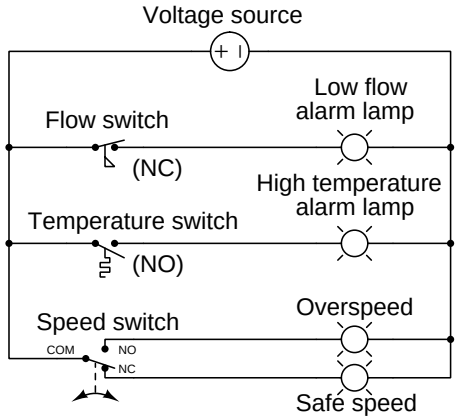
Since the purpose of this circuit is to energize the warning lamp in the event of *no coolant flow*, the flow switch's spring-return mechanism must be configured in such a way to *close* the switch contact in the absence of flow. In other words, this flow switch's contact will be closed when at rest – i.e. it will be a *normally-closed* flow switch. However, during typical operation when adequate coolant flow is present in the pipe, this switch will be held in its open state and the alarm lamp will be de-energized. Even though the flow switch is *normally-closed* (NC), in this application it will be *typically open* – the "normal" and "typical" states for this switch in this application are opposite.

Now consider the addition of a different type of switch and alarm lamp to the circuit, with the new switch installed on the same heat-dissipating engine serving to warn personnel if the engine becomes too hot:



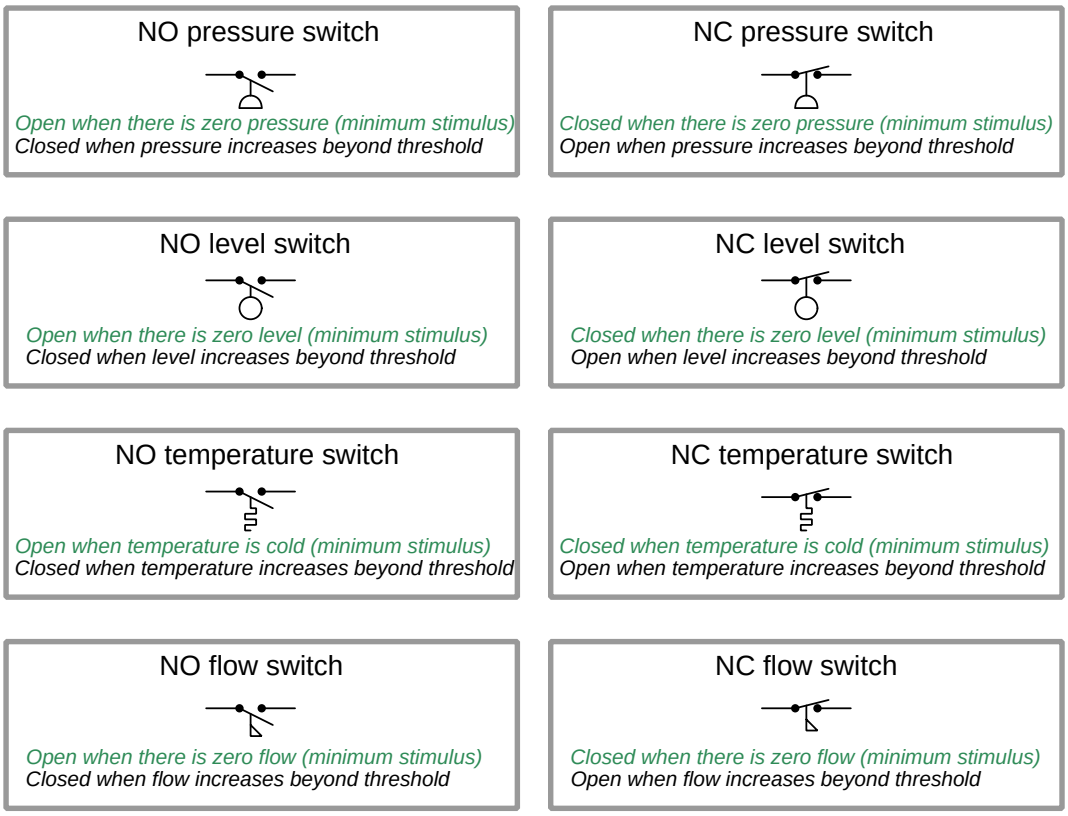
This new switch's purpose is to energize its warning lamp in the event the engine overheats, and its mechanism must be configured to *close* the switch in the presence of high temperature. This means the temperature switch's spring-return will force it open at rest, making it a *normally-open* temperature switch. During typical operation when the engine's temperature is within reasonable bounds, this switch will still be in its resting state, and so this *normally-open* (NO) temperature switch will also be *typically open* – a case where "normal" and "typical" states happen to be identical.

Let us consider one more switch application for this hypothetical engine, this time using a *single-pole, double-throw* (*SPDT*) speed switch to monitor the engine's shaft speed and trigger energization of two indicator lamps, one for "safe speed" and another for "overspeed":



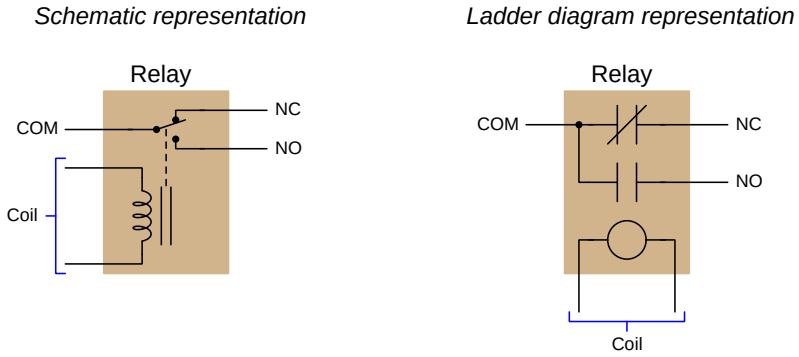
Note the COM, NO, and NC labeling of this switch's three terminals, denoting "Common", "Normally-Open", and "Normally-Closed", respectively. As with the other two switches, these contact labels as well as the switch symbol itself as drawn in the diagram represent the switch's state *when at rest*. This is strict convention in electrical switching circuits: the "normal" state of any switch is defined by a condition of minimal stimulus, and this is always how it is drawn.

A helpful tip to remember about sensing switches and their respective symbols is that the symbols are conventionally drawn in such a way that an *upward* motion of the movable switch element represents *increasing stimulus*. Here are some examples of this, showing various switch types and NO/NC contact configurations, comparing their states with no stimulus versus when the stimulus exceeds the each switch's threshold or "trip" setting. The *normal* status of each switch as defined by the manufacturer is labeled in green text:



Interestingly, the convention of upward motion representing the direction of stimulus is not maintained for hand-operated switches.

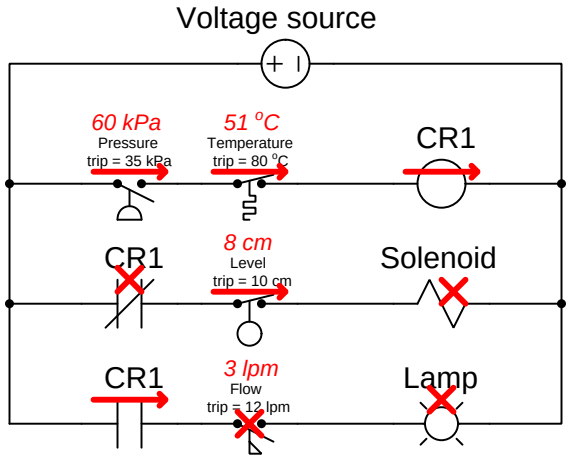
Switch contacts within electromechanical relays are also characterized as being either normally-open (NO) or normally-closed (NC), and in this case the stimulus in question is the energization of the relay's electromagnet coil. When the coil is de-energized, the contacts will all be in their resting (i.e. "normal") states which is also how the relay's contacts are drawn in diagrams. When the coil is energized, though, all contacts within the relay flip to their opposite states: all NO contacts *close* and all NC contacts *open*. The specific symbols used to represent relay coils and contacts differ according to the type of diagram, but their meaning is the same:



A normally-closed (NC) relay contact is one which will be in its closed state when the coil is de-energized, represented in diagram form by touching lines or by a slash mark between the two contact plates. A normally-open (NO) relay contact is one which will be in its open state when the coil is de-energized, represented in diagram form by an air gap between the contacting surfaces. Upon energization of the relay coil, all the contacts within that relay change state, *but their written symbols remain the same*[1] in order to represent their *resting* states.

[1] A bad habit some people adopt is to draw a slash mark through a relay contact symbol in order to annotate that relay contact's *closure* when analyzing the diagram for a relay-based circuit. This habit should avoided, as the symbols used to represent *normal* status should never be used to represent *present* status. There is enough confusion as it is surrounding the term "normal" without any more being added, so please do not contribute to the chaos!

When analyzing electrical switching circuits, a helpful problem-solving strategy is to annotate the diagram with symbols denoting the *actual* status of each switch contact in any given circuit condition, and not the *normal* status. Such annotations make it easier to determine which loads in a circuit will be energized, and which will not, for any given circuit condition. For this I recommend sketching an arrow or a line nearby a contact to show a closed state, and an "X" nearby a contact to show an open state. These annotations demonstrate real contact status without obscuring normal status. Consider these annotations used in the following example diagram:



In the upper "rung" of this ladder-style diagram we see the normally-open pressure switch is actuated (i.e. closed) because the applied pressure of 60 kPa exceeds the switch's trip setting of 35 kPa. The normally-closed temperature switch is unactuated (i.e. closed) because the applied temperature of 51 ^{o}C is less than the trip threshold of 80 ^{o}C. The red arrows annotating both switches show their closed statuses. Wired in series, these two closed switch contacts permit energizing current to the coil of relay CR1, and so another red arrow drawn there indicates that coil's energized status.

In the second and third rungs we see the present status of each CR1 contact. Since the CR1 relay coil is energized it places each CR1 relay contact into a state opposite of its "resting" or "normal" condition, therefore the normally-closed CR1 contact in rung 2 is open (shown with a red "X" annotation) and the normally-open CR1 contact in rung 3 is closed (shown with a red arrow annotation). The level switch's stimulus is less than its trip setting, and so that normally-closed contact remains closed and gets a red arrow. The flow switch's stimulus is also less than its trip setting, and so that normally-open switch remains open and gets a red "X" annotation. Neither rung 2 nor rung 3 is completed because one of the series-connected contacts in each rung is open thus preventing energization of its load. Therefore, both the solenoid coil and the lamp are de-energized, shown with red "X" annotations.

5.2 Logic families

Many possible circuit designs exist to create digital logic gates and associated logic circuitry. For example, one could design and build a NAND gate using nothing but bipolar (NPN and/or PNP) transistors; alternatively, one could make a NAND gate using nothing but MOSFETs. And, for each of these transistor types there are many variations of circuit design possible, such that any logic gate made according to a particular circuit design standard would have unique characteristics, some of which are listed here:

- DC power supply voltage range
- Acceptable "high" and "low" signal voltage levels at gate input terminals
- Guaranteed "high" and "low" signal voltage levels at gate output terminals
- Typical propagation delay times
- Typical output current limitations

When selecting logic ICs to form larger, more complex digital logic systems, it is important for the circuit designer to know that those logic components will function well with each other: that their DC power supply voltage ranges are compatible, that their output signal voltage levels will comply with their input signal voltage requirements, etc. In order to facilitate compatible device selection, logic IC manufacturers label their products with part numbers and codes designating each device's membership within different *families* of logic circuits. Each of these "families" is guaranteed to be interoperable within itself, meaning that any logic component from one family will be fully compatible with any other logic component belonging to the same family. Some families are interoperable between each other, too, but compatibility amongst members of a single IC logic family is the basic purpose of having these "family" classifications.

The early years of digital logic circuit manufacturing saw emergence of families such as Resistor-Transistor Logic (RTL), Diode-Transistor Logic (DTL), and Emitter-Coupled Logic (ECL), the first two now considered obsolete. Later emerged the Transistor-Transistor Logic (TTL) family based on NPN and PNP bipolar transistor circuitry, this family identified by part numbers beginning with either 54 or 74, the 54-series ICs having military-grade specifications (e.g. operating temperature limits) and the 74-series ICs having commercial-grade specifications. After that came the Complementary Metal-Oxide Semiconductor (CMOS) family based on N-channel and P-channel MOSFETs rather than bipolar transistors, this family identified by part numbers beginning with 4 or 14[2].

IC logic families soon developed into sub-families having different characteristics such as power consumption and switching speed, some of these sub-families designated by letters in the middle of the part number. For example, the classic 7411 is a triple 3-input AND gate IC based on TTL (bipolar transistor) technology, but in the years to follow the classic 54/74 TTL family's introduction there developed the "LS" sub-family (e.g. 74LS11 triple 3-input AND gate IC) using Schottky diodes within the internal circuitry to reduce power consumption, and then later the "HC" sub-family (e.g.

[2]The prepending of a "1" to the 4000-series part number was typical of ICs manufactured by Motorola, just to make things confusing!

74HC11 triple 3-input AND gate IC) using MOSFETs rather than bipolar transistors but otherwise designed to be backward-compatible with legacy 74 and 74LS sub-families.

More recent developments in IC logic have resulted in logic circuit designs optimized for lower and lower DC supply voltage ranges, this design trend addressing the need for more advanced consumer electronic products powered by chemical batteries. For any given amount of current, a lower operating voltage means less power dissipation, and this in turn means a battery of any given size will be able to energize that logic circuit for a longer period of time. Portable computers, mobile telephones, and other personal electronic devices naturally benefit from this technological trend.

An exhaustive list of IC logic families would be beyond the scope of this reference, and frankly is better left to the manufacturers themselves. However, here I will provide a listing of some of the more common digital IC logic families at the time of this writing (2023):

- **5400/7400 classic TTL family** – uses bipolar transistor technology; operates on 5 Volt DC power supply with a tight margin, typically 4.75 Volts minimum and 5.25 Volts maximum for the 7400 commercial-grade series, 4.5 Volts minimum and 5.5 Volts maximum for the 5400 military-grade series

- **4000 classic CMOS family** – uses complementary (N- and P-channel together) MOSFET technology; operates on a wide range of DC power supply voltages, typically 3 Volts minimum to 18 Volts maximum, often standardized at 5 Volts, 10 Volts, or 15 Volts; notably slower in switching speed than classic TTL but operates at a *far* lower power dissipation[3]

- **5400/7400 ALS, AS, S, and LS sub-families** – uses Schottky diodes within the internal TTL circuitry to help avoid transistor saturation and thereby increase maximum switching speeds; same DC power supply range as classic 5400/7400 TTL

- **5400/7400 F sub-family** – this is a "fast" variant of TTL logic designed for low propagation delay times and high-speed operation; limited to the same DC power supply voltage range as classic 5400/7400 TTL devices but with significantly higher current requirements and consequently higher power dissipation

- **5400/7400 HC sub-family** – uses MOSFETs rather than bipolar transistors internally, but designed to mimic the operation of classic TTL devices at much-reduced power dissipation; enjoys a wider DC power supply range than classic TTL, typically 2 Volts minimum and 6 Volts maximum

- **5400/7400 HCT sub-family** – similar to the HC sub-family in its use of MOSFETs rather than bipolar transistors, but designed to be fully interoperable with classic TTL devices; limited to the same 5400-series classic TTL power supply range of 4.5 Volts minimum and 5.5 Volts maximum

- **5400/7400 BCT sub-family** – uses a combination of bipolar transistors and MOSFETs internally (BiCMOS); limited to the same 5400-series classic TTL power supply range of 4.5 Volts minimum and 5.5 Volts maximum

[3] This trade-off between device speed versus device power dissipation is a common one in digital electronics. Often we need to sacrifice one to achieve superior performance in the other.

- **5400/7400 LVC sub-family** – this "low-voltage CMOS" sub-family operates with a considerably lower DC power supply range than previous 5400/7400 digital logic sub-families, typically 2 Volts minimum and 3.6 Volts maximum, often standardized at 3.3 Volts
- **5400/7400 LVT sub-family** – this "low-voltage BiCMOS" sub-family uses a combination of bipolar transistors and MOSFETs internally and operates on a DC power supply voltage range of 2.7 Volts minimum and 3.6 Volts maximum, often standardized at 3.3 Volts
- **5400/7400 AVC sub-family** – this "advanced low-voltage CMOS" sub-family extends the operating DC power supply voltage to even lower levels with 1.4 Volts being minimum, often standardized at 3.3 Volts, 2.5 Volts, or 1.8 Volts
- **5400/7400 AUC sub-family** – this "advanced ultra-low voltage CMOS" sub family pushes the DC power supply voltage envelope down even further, with 0.8 Volts minimum and 3.6 Volts maximum, often standardized at 2.5 Volts, 1.8 Volts, and 1.2 Volts

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 combinational logic using C++

Here is an example C++ program intended to display a crude representation of a four-function combinational logic circuit and then compute its output state, using the `bool` data type defined in the C++ language:

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

int main (void)
{
  bool A, B, C, D, Out;

  A = 1;
  B = 0;
  C = 0;
  D = 1;

  cout << "A ---|                                " << endl;
  cout << "     |NOR|-----|                     " << endl;
  cout << "B ---|         |                     " << endl;
  cout << "               |AND|---|             " << endl;
  cout << "C ---|NOT|-----|       |             " << endl;
  cout << "                       |NAND|--- Out " << endl;
  cout << "D --------------------|             " << endl;

  cout << endl;

  cout << "A = " << A << endl;
  cout << "B = " << B << endl;
  cout << "C = " << C << endl;
  cout << "D = " << D << endl;

  Out = !(!(A || B) && !C && D);

  cout << "Out = " << Out << endl;

  return 0;
}
```

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

```
A ---|
     |NOR|-----|
B ---|         |
               |AND|---|
C ---|NOT|-----|       |
                       |NAND|--- Out
D ---------------------|

A = 1
B = 0
C = 0
D = 1
Out = 1
```

Let's analyze how this program works, exploring the following programming principles along the way:

- Preprocessor directives, namespaces
- The `main` function
- Delimiter characters (e.g. { } ;)
- Variable types (`bool`), names, and declarations
- Variable assignment/initialization (`=`)
- Printing text output (`cout`)
- Boolean operators (`!`, `||`, `&&`)
- Accepting user input (`cin`)

Beginning at the top of the source code listing, we see `#include` and `namespace` directives instructing the compiler how to interpret some of the other instructions (e.g. `cout`) found in the source code. The `main` function contains all the code we wish to run; i.e. all the lines of code following the opening-brace symbol ({) and preceding the closing-brace symbol (}).

Variables `A` through `D` 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. These four Boolean-type variables are assigned values, then a sequence of `cout` lines print text to its console: text enclosed within quotation marks printed verbatim and mathematical expressions printed as their logical values. `endl` control characters force a line of text to end and a new line of text to begin.

As you can see, most of the `cout` instructions exist only to print explanatory text to the console for the user's benefit. This program could be re-written to be *much* simpler by omitting these lines and focusing solely on the calculation of the `Out` variable.

Note the use of the Boolean NOT operator (`!`) preceding each inverted variable. This is similar the to legacy use of an apostrophe ("prime" symbol) to denote inversion in typeset Boolean

expressions, except the apostrophe symbol always came immediately *after* the variable, while the C++ inversion symbol must come immediately *prior to* the variable. Parentheses are necessary to properly group the variables together for these inversions.

One improvement we can make to this program is to give the user ability to enter their own values for `A` through `D`, and this is easily done using the `cin` instruction as shown below. Furthermore, we can streamline the text presentation by showing the variable states at the left-hand edge of the diagram instead of listing them separately:

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

int main (void)
{
  bool A, B, C, D, Out;

  cout << "A = ";
  cin >> A;
  cout << "B = ";
  cin >> B;
  cout << "C = ";
  cin >> C;
  cout << "D = ";
  cin >> D;

  Out = !(!(A || B) && !C && D);

  cout << endl;

  cout << "A " << A << " ---| " << endl;
  cout << "       |NOR|-----| " << endl;
  cout << "B " << B << " ---|         | " << endl;
  cout << "                 |AND|---| " << endl;
  cout << "C " << C << " ---|NOT|-----|       | " << endl;
  cout << "                           |NAND|--- Out " << Out << endl;
  cout << "D " << D << " ----------------------| " << endl;

  return 0;
}
```

The gate-symbol characters shown in the `cout` statements do not appear as well-aligned as in the first version of the program, because some of these *cout* statements instruct the computer to print variable values to the console while others merely display static text.

When compiled and executed, though, the result is much more useful than before. The first four lines are the prompts for my input, revealing that I happened to enter zero values for `A` through `C` and one for `D`:

```
A = 0
B = 0
C = 0
D = 1

A 0 ---|
       |NOR|-----|
B 0 ---|         |
                 |AND|---|
C 0 ---|NOT|-----|       |
                         |NAND|--- Out 0
D 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.

Thought experiments as a problem-solving strategy

Discrete signal

Logic function

Truth table

Boolean algebra

OR function

AND function

NOT function

NOR function

NAND function

XOR function

Logic state

Logic gate

Relay ladder diagram

Combinational function

Universal function

DeMorgan's Theorem

Double-negation

Converting circuit to Boolean

Annotating relay states

Normal state of a switch

Converting Boolean to circuit

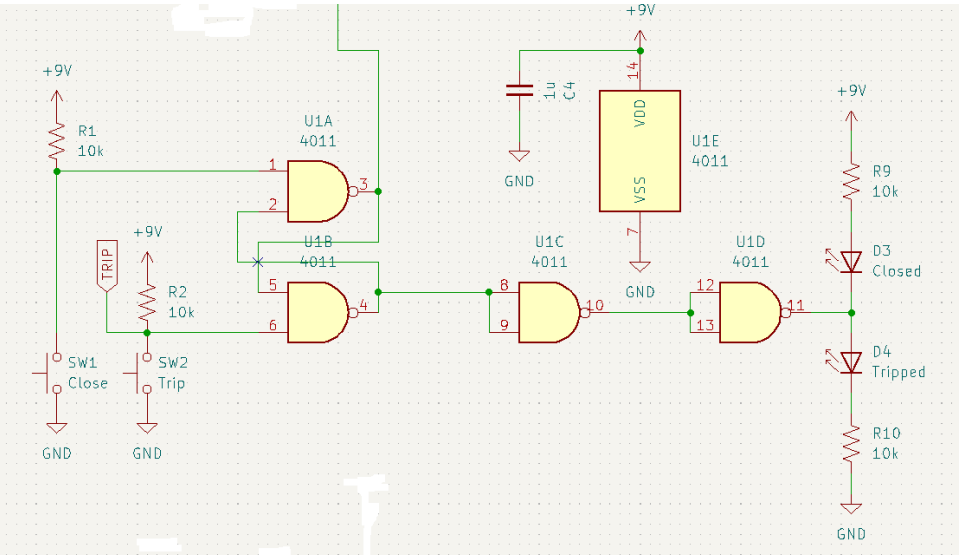
Converting truth table to circuit

Simplifying Boolean expressions

Simplifying relay circuits

7.1.3 Use of spare NAND gates

Digital logic gates are often manufactured in sets packaged inside of one housing. An example of this is the CD4011 quad NAND gate integrated circuit (IC), called "quad" because it contains *four* two-input NAND gates inside of one 14-pin "DIP" package. In the following schematic diagram we see all four NAND gates of one IC used in this circuit:



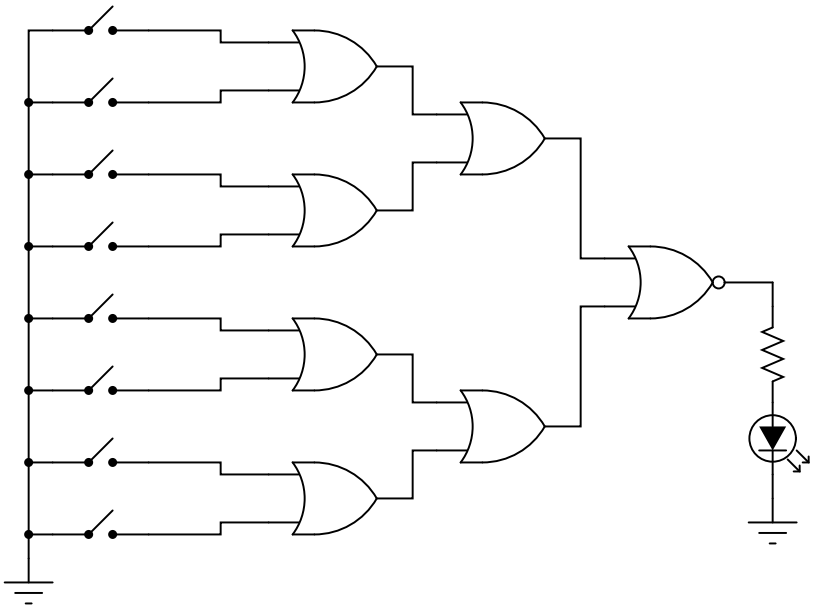
What purpose do the "C" and "D" gates of this one CD4011 IC serve, and why do you suppose the designer of this circuit chose to use these two NAND gates rather than some other type of logic gate?

Challenges

- Which terminals of the CD4011 IC accept DC power, and how can you tell from this schematic diagram?

7.1.4 Unanimous vote detector circuit

The digital circuit shown here is a unanimous-yea vote detector. Votes are cast by eight different voters by the setting of switches in either the closed (yea) or open (nay) positions. According to the logic function provided by the TTL gates, the LED will energize if and only if all switches are closed:



As is common in digital circuit schematics, the power supply (V_{CC}) is omitted for the sake of simplicity. This is analogous to the omission of power supply connections in many operational amplifier circuit schematics.

If we were to draw a truth table for this circuit, how large (number of rows and columns) would the table have to be?

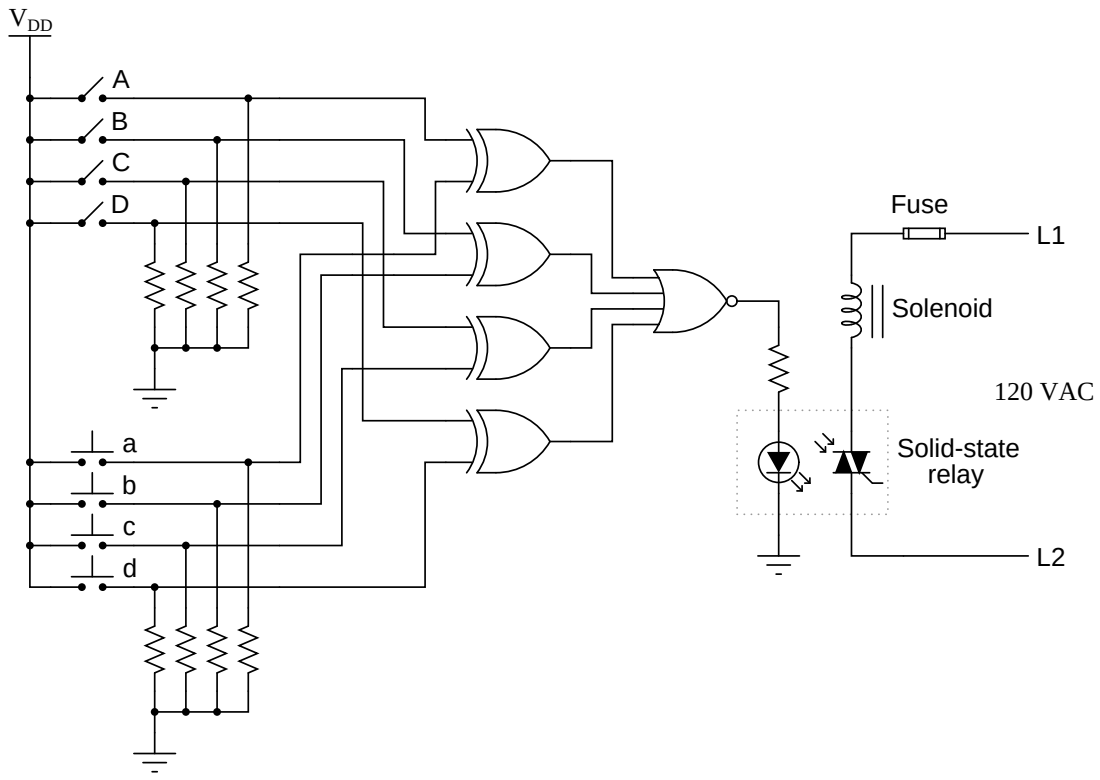
Suppose we wished to modify this circuit, such that an electromechanical bell would ring whenever a unanimous-yea vote was cast, rather than merely lighting a small LED. The bell we have in mind to use is rather large, its solenoid coil drawing 3 amps of current at a voltage of 12 volts DC: well beyond the final gate's ability to source. How could we modify this circuit so that the final gate is able to energize this bell instead of just an LED?

Challenges

- Explain why pullup resistors are not required in the input lines to this logic circuit.
- Redesign this circuit so that it performs the same unanimous vote detection function without the use of any logic gates.

7.1.5 Combination lock circuit

The following schematic diagram shows a simple electronic combination lock, controlling power to a door lock solenoid:



The four pushbutton switches (a, b, c, and d) are accessible to the person wishing to enter the door. The four toggle switches (A, B, C, and D) are located behind the door, and are used to set the code necessary for entering.

Explain how this system is supposed to work. What are the logic states of the respective gate outputs when a matching code is entered through the pushbutton switches? How about when a non-matching code is entered?

Do you see any security problems with this door lock circuit? How easy would it be for someone to enter, who does not know the four-bit code? Do you have any suggestions for improving this lock design?

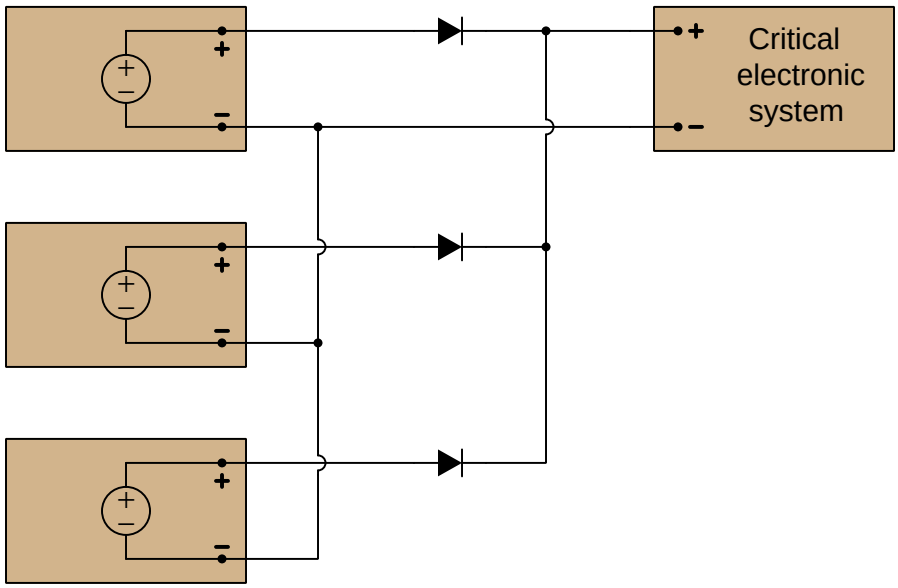
Challenges

- Explain how to properly size the banks of resistors at the gate inputs.

- We often see *commutating diodes* installed in parallel with inductive loads, to protect the semiconductor switching device(s) controlling that load's energization. Here, however, we do not. Explain why.

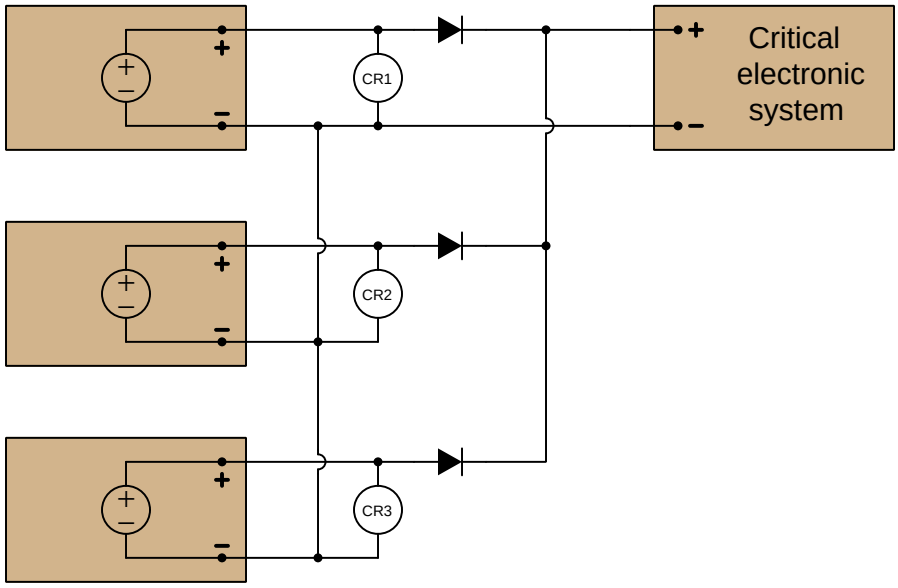
7.1.6 Triple-redundant power supply

A critical electronic system receives DC power from three power supplies, each one feeding through a diode, so that if one power supply develops an internal short-circuit, it will not cause the others to overload:

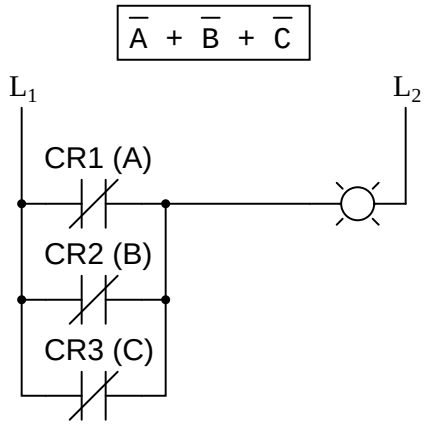


The only problem with this system is that we have no indication of trouble if just one or two power supplies do fail. Since the diode system routes power from any available supply(ies) to the critical system, the system sees no interruption in power if one or even two of the power supplies stop outputting voltage. It would be nice if we had some sort of alarm system installed to alert the technicians of a problem with any of the power supplies, long before the critical system was in jeopardy of losing power completely.

An engineer decides that a relay could be installed at the output of each power supply, prior to the diodes. Contacts from these relays could then be connected to some sort of alarm device (flashing light, bell, etc.) to alert maintenance personnel of any problem:



The first solution to this problem is to connect a warning lamp such that it will illuminate if *any* of the three power supplies fails:



However, it is soon discovered that this circuit generates "nuisance alarms" whenever a technician powers down any of the redundant power supplies for routine maintenance. Re-design the ladder logic circuit so that the lamp illuminates only if *two out of three* power supplies fail.

Design an alternate solution whereby the lamp illuminates if any of the power supplies fail, but there is provision for an alarm *bypass* switch that the maintenance technician could use to suppress the alarm during routine work.

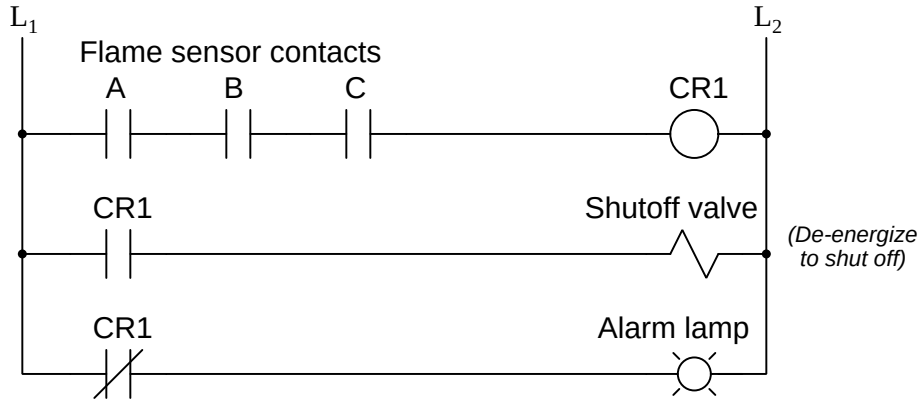
Challenges

- What would be the result of one of the diodes failing open?

7.1.7 Chemical weapons incinerator

A chemical weapons agent incinerator uses high-temperature combustion to neutralize toxic chemicals, in an effort to safely dispose of unused chemical weapons. Maintaining an ever-present flame is obviously important for this system to function safely because if the flame ever went out the toxic chemicals could pass through the incinerator unaltered and pose an immediate threat to life and health near the incinerator's exhaust vent.

Three independent flame sensors monitor the incinerator's combustion, and under typical operating conditions all three sensors must indicate a "good" flame to keep the master relay energized. If any of the sensors lose flame signal, the corresponding relay contact will open which will cause the master relay to de-energize, and this in turn shuts down the flow of chemical agent into the incinerator and also activates an alarm:



However, this system has a problem. If ever a maintenance technician performs a routine service test on any of the flame sensors, the system interprets that test as a failed signal and shuts everything down. Somehow, we must find a solution allowing routine testing yet providing redundancy of flame detection so that we never rely on just *one* of the sensors for safety. Unlikely thought it may be, it is possible for a sensor to fail with a false-positive indication of flame, which is why we have multiple flame sensors.

An engineer suggests installing a "Maintenance Bypass" switch forcing the system into a two-out-of-three (2oo3) redundancy mode instead of the typical three-out-of-three (3oo3). This way a maintenance technician could flip the Bypass switch, then perform routine checks of any one of the sensors (at a time), and the system would still operate if the remaining two sensors provided indication of good flame in the incinerator.

Implement this switchable 2oo3/3oo3 system in relay ladder logic:

Challenges

- Why do you suppose it is wise to build this system using a de-energize-to-shut-off valve?

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[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] 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.

[5] 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*[6] 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.

[6] 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[7] 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[8] 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[9] 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.

[7] 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.

[8] 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.

[9] 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[10] – 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.

[10] 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 combinational 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 eight commands typed at the prompt (`>>>`) of a Python interpreter[11] demonstrating the truth table for a three-input combinational function where two inputs feed into an OR function, then the output of that and a third input feed into an AND 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) and False
False
>>> (False or False) and True
False
>>> (False or True) and False
False
>>> (False or True) and True
True
>>> (True or False) and False
False
>>> (True or False) and True
True
>>> (True or True) and False
False
>>> (True or True) and True
True
>>> quit()
```

[11] 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.

Once you have Python installed and working on your computer[12], demonstrate the following:

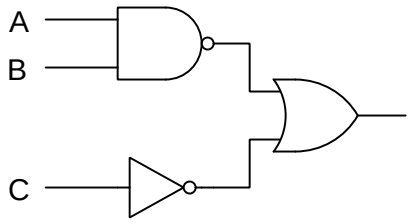
- Demonstrate the truth table for a four-input combinational function, where two inputs feed into an AND function, two more inputs feed into an OR function, and the outputs of the AND and OR functions feed into a two-input OR function.

Challenges

- How many commands would you need to enter at the Python prompt to fully explore the truth table of a five-input combinational function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of a six-input combinational function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of an n-input combinational function?

7.2.4 Using Python to evaluate a combinational function diagram

Examine the following combinational logic diagram, and then use Python (the computer programming language) to compute its output for the input conditions $A = 1$, $B = 0$, and $C = 1$:



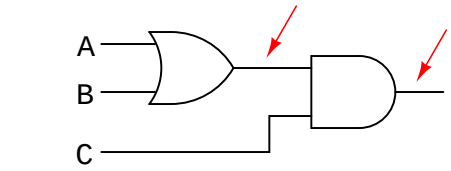
Challenges

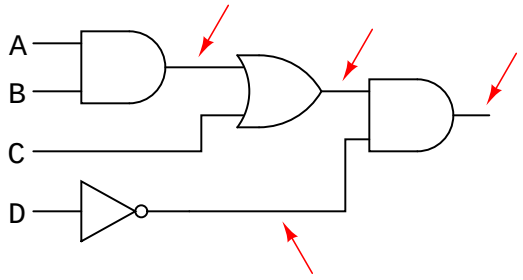
- How many commands would you need to enter at the Python prompt to fully explore the truth table of a five-input combinational function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of a six-input combinational function?
- How many commands would you need to enter at the Python prompt to fully explore the truth table of an n-input combinational function?

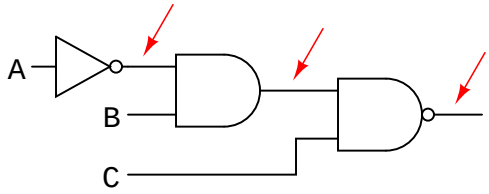
[12] One option for Microsoft Windows users is to install `Cygwin` which is a Unix shell (terminal) application, and included in the "Development" package installation of `Cygwin` is Python. Another option uses the online interpreter available at `https://python.org/shell`.

7.2.5 Boolean expressions from gate circuits

Write Boolean expressions for each of the following logic gate circuits, also writing Boolean sub-expressions next to each gate output in the diagrams:





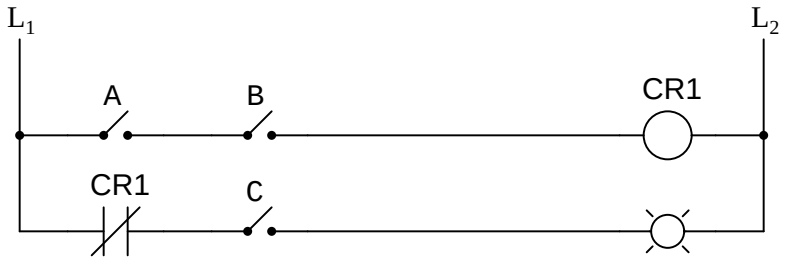


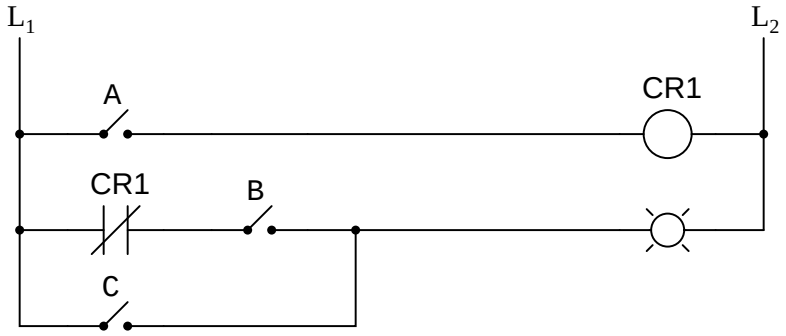
Challenges

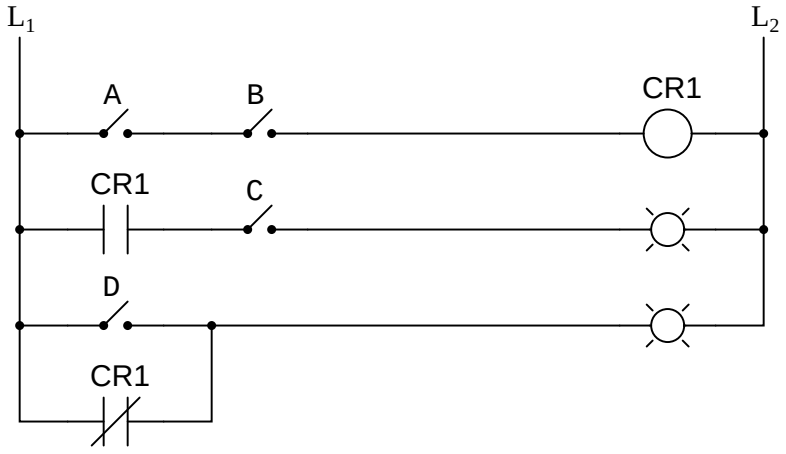
- Explain why it is helpful to write the sub-expressions rather than just the final expression.

7.2.6 Boolean expressions from relay circuits

Write Boolean expressions for each of the following relay ladder-logic circuits, also writing Boolean sub-expressions next to each rung in the diagrams:







Challenges

- Explain why it is helpful to write the sub-expressions rather than just the final expression.

7.2.7 Truth tables from Boolean expressions

Complete truth tables for the following Boolean expressions:

$$\text{Output} = \overline{A} + \overline{B} + C$$

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

$$\text{Output} = A(B + AC + \overline{A})$$

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Challenges

- Explain how it is possible to conclude, at first glance, that the truth table for the second Boolean expression must have "0" output states for the first four rows.

7.2.8 Gate circuits from Boolean expressions

Design logic gate circuits to fulfill each of the following Boolean expressions, each circuit driving a heavy DC load requiring more current than the final gate is able to either source or sink:

$$A\overline{B} + \overline{C}(A + B)$$

$$(\overline{A} + B)(AB + \overline{C})$$

Challenges

- An important mathematical principle is *order of operations*, instructing us as to which arithmetic operation needs to be performed first, next, etc. Explain how this fundamental principle applies to this particular problem.
- Determine the output states for each of these combinational circuits for a condition where $A = 0$ and $B = 0$ and $C = 0$.
- Determine the output states for each of these combinational circuits for a condition where $A = 0$ and $B = 1$ and $C = 0$.
- Determine the output states for each of these combinational circuits for a condition where $A = 1$ and $B = 1$ and $C = 0$.
- How may we ensure that the transistor is fully "saturated" when turned on by the gate's output?

7.2.9 Relay circuits from Boolean expressions

Design relay ladder-logic circuits to fulfill each of the following Boolean expressions:

$$A\overline{B}C + (B + \overline{A}C)$$

$$(A + B\overline{C})(\overline{A} + B + C)$$

Challenges

- An important mathematical principle is *order of operations*, instructing us as to which arithmetic operation needs to be performed first, next, etc. Explain how this fundamental principle applies to this particular problem.

7.2.10 Circuits from two-input truth tables

Design a Boolean algebra expression, a logic gate circuit, and a relay ladder logic circuit implementing the following truth tables:

Example #1:

A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

Example #2:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Example #3:

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Challenges

- Can any of the direct implementations of the truth table's function be simplified to require fewer Boolean variables, or fewer circuit components?
- Check your answers by plugging in all possible combinations of the input variable states and verifying that the expression and circuits all result in the same output states.

7.2.11 SOP and POS expressions from the same truth table

Examine this truth table and then write both SOP and POS Boolean expressions describing the Output:

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

Challenges

- Which of these two expressions is most concise?
- How would the equivalent logic gate circuits differ?

7.2.12 Circuits from three-input truth tables

Design a Boolean algebra expression, a logic gate circuit, and a relay ladder logic circuit implementing the following truth tables:

Example #1:

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Example #2:

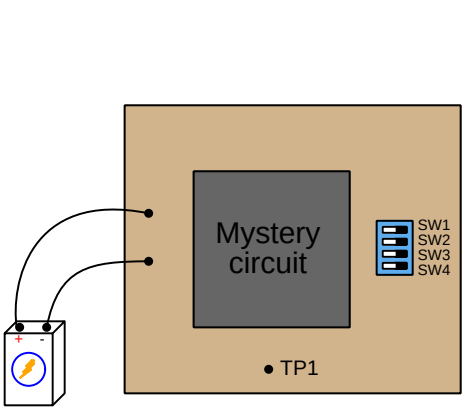
A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Challenges

- Can any of the direct implementations of the truth table's function be simplified to require fewer Boolean variables, or fewer circuit components?
- Check your answers by plugging in all possible combinations of the input variable states and verifying that the expression and circuits all result in the same output states.

7.2.13 Boolean expression for an undocumented logic circuit

Suppose you were faced with the task of writing a Boolean expression for a logic circuit, the internals of which are unknown to you. The circuit has four inputs – each one set by the position of its own micro-switch – and one output. By experimenting with all the possible input switch combinations, and using a logic probe to "read" the output state (at test point TP1), you were able to write the following truth table describing the circuit's behavior:



SW1	SW2	SW3	SW4	TP1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Write a Boolean expression based on this truth table "description" of the circuit.

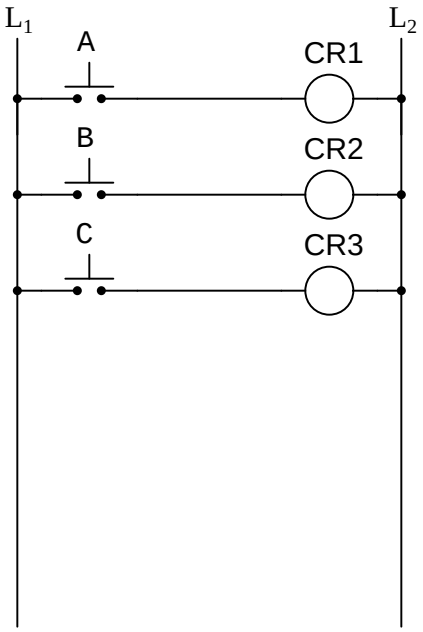
Challenges

- Sketch a logic gate diagram equivalent in function to this circuit.

7.2.14 SOP expression and ladder logic from a truth table

Write an SOP expression for this truth table, and then draw a ladder logic (relay) circuit diagram corresponding to that SOP expression:

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



Implement the SOP logic function using contacts of relays CR1, CR2, and CR3. A partial ladder logic diagram has been provided for you.

Challenges

- Sketch a logic gate circuit to implement the same truth table.

7.2.15 Gate circuit from a truth table

Sketch a logic gate circuit to implement this truth table, showing how the circuit may be used to drive a heavy DC load having current requirements exceeding that of the final gate's sinking or sourcing ability:

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

Challenges

- Explain why a standard SOP expression would be very cumbersome describing this circuit.
- How may we ensure that the transistor is fully "saturated" when turned on by the gate's output?

7.2.16 Relay circuit from a truth table

Sketch a relay ladder-logic circuit to implement this truth table:

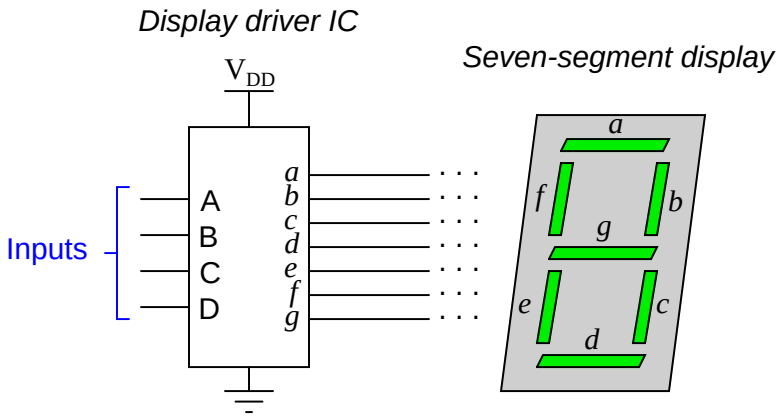
A	B	C	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Challenges

- Explain why a standard SOP expression would be very cumbersome describing this circuit.

7.2.17 Seven-segment decoder

A *seven segment decoder* is a digital circuit designed to drive a very common type of digital display device: a set of LED (or LCD) segments that render numerals 0 through 9 at the command of a four-bit code:



The behavior of the display driver IC may be represented by a truth table with seven outputs: one for each segment of the seven-segment display (a through g). In the following table, a "1" output represents an active display segment, while a "0" output represents an inactive segment:

D	**C**	**B**	**A**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	Display
0	0	0	0	1	1	1	1	1	1	0	"0"
0	0	0	1	0	1	1	0	0	0	0	"1"
0	0	1	0	1	1	0	1	1	0	1	"2"
0	0	1	1	1	1	1	1	0	0	1	"3"
0	1	0	0	0	1	1	0	0	1	1	"4"
0	1	0	1	1	0	1	1	0	1	1	"5"
0	1	1	0	1	0	1	1	1	1	1	"6"
0	1	1	1	1	1	1	0	0	0	0	"7"
1	0	0	0	1	1	1	1	1	1	1	"8"
1	0	0	1	1	1	1	1	0	1	1	"9"

Write the unsimplified SOP or POS expressions (choose the most appropriate form) for outputs a, b, c, and e.

Challenges

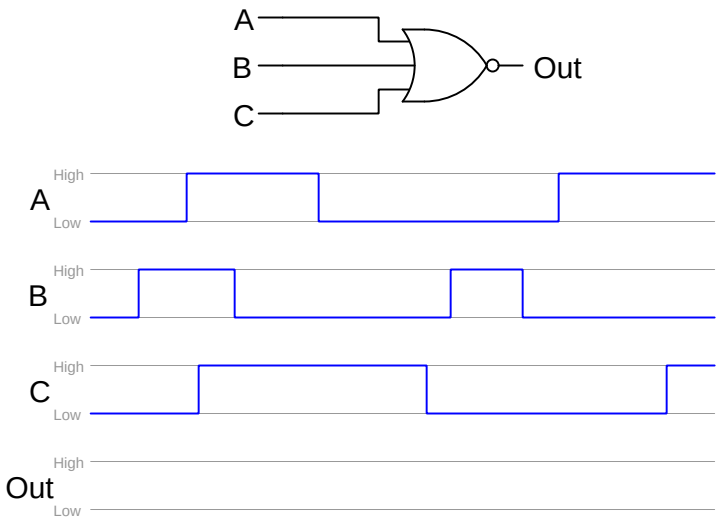
- Use the laws of Boolean algebra to simplify each of the above expressions into their simplest forms.

7.2.18 Timing diagrams for gate circuits

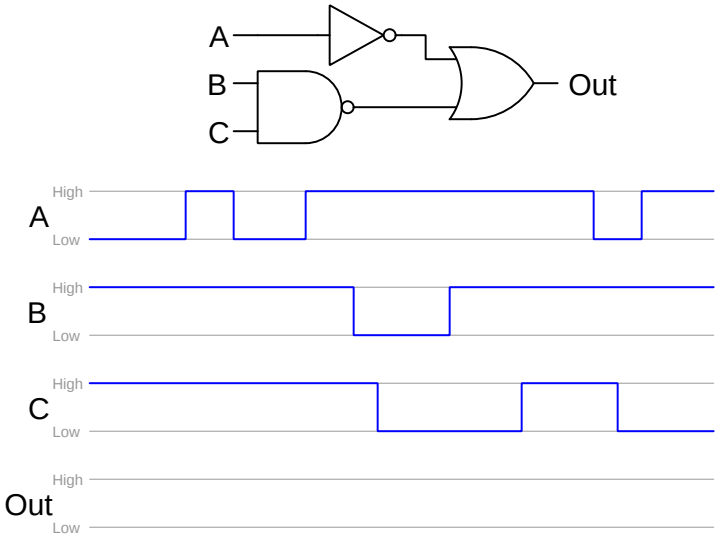
A *timing diagram* is a time-domain illustration of high/low logic signals, useful for showing how a digital logic system responds to changing inputs over a period of time. Most digital electronic systems have pulsing data states in at least some portions of the circuit, and timing diagrams show the relationships between these pulsing signals.

Complete the following timing diagrams by sketching the Output waveforms these digital logic gate circuits will produce given the input signals (A, B, and C) shown:

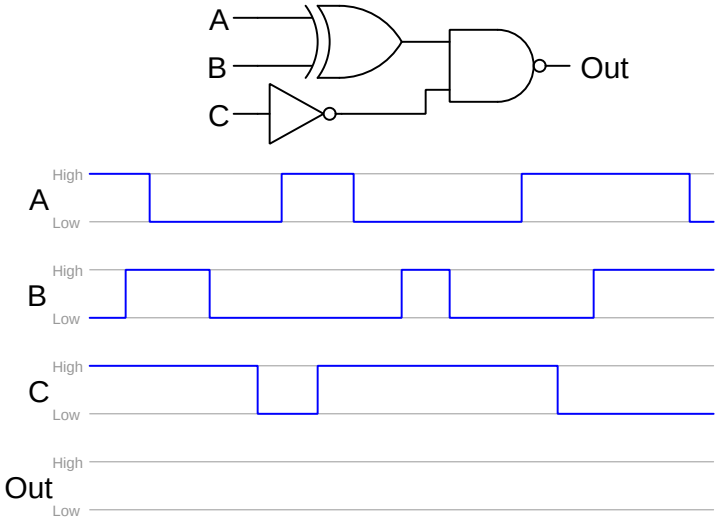
Example #1:



Example #2:



Example #3:



Challenges

- What 3-input logic gate is the Example #2 circuit equivalent to?

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.

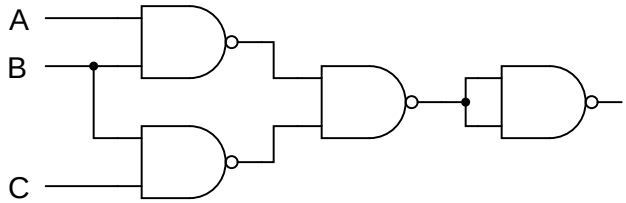
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of 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 Effect of gate fault on Boolean expression

Write the Boolean expression for the following logic gate circuit:



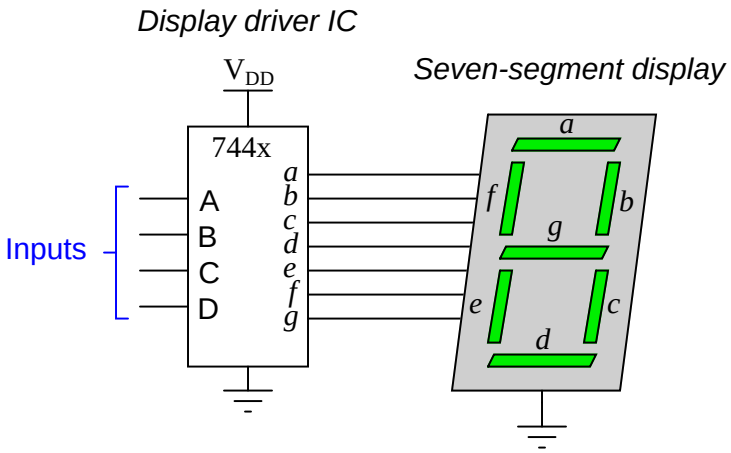
Now suppose the lower-left gate (the one connected to the C input) fails with a "high" output status regardless of input status. How does this fault affect the Boolean expression describing this circuit's functionality?

Challenges

- Identify a single fault that would continually force the output to a "high" state.
- Identify a single fault that would continually force the output to a "low" state.
- Simply this circuit to implement the same logical function using fewer gates.
- Describe a good reason for building this circuit with identical gates, rather than building a logically equivalent circuit with a mix of gate types.

7.3.2 Seven-segment decoder/driver problem

Two electronics students attempt to build 7-segment display circuits, one using a 7447 decoder/driver IC and the other using a 7448. Both students connect their ICs to common-cathode 7-segment displays as such:



The student using the 7448 notices the LED segments glowing faintly, but the patterns are not correct for the digits that are supposed to be displayed. The student using the 7447 has an even worse problem: no light at all! Both have checked and re-checked their wiring, to no avail. It seems as though all the connections are in the right place.

What do you think the problem is?

Challenges

- Redesign the circuit so that it will work properly.

Chapter 8

Projects and Experiments

The following project and experiment descriptions outline things you can build to help you understand circuits. With any real-world project or experiment there exists the potential for physical harm. *Electricity can be very dangerous in certain circumstances, and you should follow proper safety precautions at all times!*

8.1 Recommended practices

This section outlines some recommended practices for all circuits you design and construct.

8.1.1 Safety first!

Electricity, when passed through the human body, causes uncomfortable sensations and in large enough measures[1] will cause muscles to involuntarily contract. The overriding of your nervous system by the passage of electrical current through your body is particularly dangerous in regard to your heart, which is a vital muscle. Very large amounts of current can produce serious internal burns in addition to all the other effects.

Cardio-pulmonary resuscitation (CPR) is the standard first-aid for any victim of electrical shock. This is a very good skill to acquire if you intend to work with others on dangerous electrical circuits. You should never perform tests or work on such circuits unless someone else is present who is proficient in CPR.

As a general rule, any voltage in excess of 30 Volts poses a definitive electric shock hazard, because beyond this level human skin does not have enough resistance to safely limit current through the body. "Live" work of any kind with circuits over 30 volts should be avoided, and if unavoidable should only be done using electrically insulated tools and other protective equipment (e.g. insulating shoes and gloves). If you are unsure of the hazards, or feel unsafe at any time, stop all work and distance yourself from the circuit!

A policy I strongly recommend for students learning about electricity is to *never come into electrical contact*[2] *with an energized conductor, no matter what the circuit's voltage*[3] *level!* Enforcing this policy may seem ridiculous when the circuit in question is powered by a single battery smaller than the palm of your hand, but it is precisely this instilled habit which will save a person from bodily harm when working with more dangerous circuits. Experience has taught me that students who learn early on to be careless with safe circuits have a tendency to be careless later with dangerous circuits!

In addition to the electrical hazards of shock and burns, the construction of projects and running of experiments often poses other hazards such as working with hand and power tools, potential

[1] Professor Charles Dalziel published a research paper in 1961 called "The Deleterious Effects of Electric Shock" detailing the results of electric shock experiments with both human and animal subjects. The threshold of perception for human subjects holding a conductor in their hand was in the range of 1 milliampere of current (less than this for alternating current, and generally less for female subjects than for male). Loss of muscular control was exhibited by half of Dalziel's subjects at less than 10 milliamperes alternating current. Extreme pain, difficulty breathing, and loss of all muscular control occurred for over 99% of his subjects at direct currents less than 100 milliamperes and alternating currents less than 30 milliamperes. In summary, it doesn't require much electric current to induce painful and even life-threatening effects in the human body! Your first and best protection against electric shock is maintaining an insulating barrier between your body and the circuit in question, such that current from that circuit will be unable to flow through your body.

[2] By "electrical contact" I mean either directly touching an energized conductor with any part of your body, or indirectly touching it through a conductive tool. The only physical contact you should ever make with an energized conductor is via an electrically insulated tool, for example a screwdriver with an electrically insulated handle, or an insulated test probe for some instrument.

[3] Another reason for consistently enforcing this policy, even on low-voltage circuits, is due to the dangers that even some low-voltage circuits harbor. A single 12 Volt automobile battery, for example, can cause a surprising amount of damage if short-circuited simply due to the high current levels (i.e. very low internal resistance) it is capable of, even though the voltage level is too low to cause a shock through the skin. Mechanics wearing metal rings, for example, are at risk from severe burns if their rings happen to short-circuit such a battery! Furthermore, even when working on circuits that are simply too low-power (low voltage and low current) to cause any bodily harm, touching them while energized can pose a threat to the circuit components themselves. In summary, it generally wise (and *always* a good habit to build) to "power down" *any* circuit before making contact between it and your body.

contact with high temperatures, potential chemical exposure, etc. You should never proceed with a project or experiment if you are unaware of proper tool use or lack basic protective measures (e.g. personal protective equipment such as safety glasses) against such hazards.

Some other safety-related practices should be followed as well:

- All power conductors extending outward from the project must be *firmly* strain-relieved (e.g. "cord grips" used on line power cords), so that an accidental tug or drop will not compromise circuit integrity.
- All electrical connections must be sound and appropriately made (e.g. soldered wire joints rather than twisted-and-taped; terminal blocks rather than solderless breadboards for high-current or high-voltage circuits). Use "touch-safe" terminal connections with recessed metal parts to minimize risk of accidental contact.
- Always provide overcurrent protection in any circuit you build. *Always.* This may be in the form of a fuse, a circuit breaker, and/or an electronically current-limited power supply.
- Always ensure circuit conductors are rated for more current than the overcurrent protection limit. *Always.* A fuse does no good if the wire or printed circuit board trace will "blow" before it does!
- Always bond metal enclosures to Earth ground for any line-powered circuit. *Always.* Ensuring an equipotential state between the enclosure and Earth by making the enclosure electrically common with Earth ground ensures no electric shock can occur simply by one's body bridging between the Earth and the enclosure.
- Avoid building a high-energy circuit when a low-energy circuit will suffice. For example, I always recommend beginning students power their first DC resistor circuits using small batteries rather than with line-powered DC power supplies. The intrinsic energy limitations of a dry-cell battery make accidents highly unlikely.
- Use line power receptacles that are GFCI (Ground Fault Current Interrupting) to help avoid electric shock from making accidental contact with a "hot" line conductor.
- Always wear eye protection when working with tools or live systems having the potential to eject material into the air. Examples of such activities include soldering, drilling, grinding, cutting, wire stripping, working on or near energized circuits, etc.
- Always use a step-stool or stepladder to reach high places. Never stand on something not designed to support a human load.
- When in doubt, *ask an expert.* If anything even seems remotely unsafe to you, do not proceed without consulting a trusted person fully knowledgeable in electrical safety.

8.1.2 Other helpful tips

Experience has shown the following practices to be very helpful, especially when students make their own component selections, to ensure the circuits will be well-behaved:

- Avoid resistor values less than 1 kΩ or greater than 100 kΩ, unless such values are definitely necessary[4]. Resistances below 1 kΩ may draw excessive current if directly connected to a voltage source of significant magnitude, and may also complicate the task of accurately measuring current since any ammeter's non-zero resistance inserted in series with a low-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Resistances above 100 kΩ may complicate the task of measuring voltage since any voltmeter's finite resistance connected in parallel with a high-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Similarly, AC circuit impedance values should be between 1 kΩ and 100 kΩ, and for all the same reasons.

- Ensure all electrical connections are low-resistance and physically rugged. For this reason, one should avoid *compression splices* (e.g. "butt" connectors), solderless breadboards[5], and wires that are simply twisted together.

- Build your circuit with **testing** in mind. For example, provide convenient connection points for test equipment (e.g. multimeters, oscilloscopes, signal generators, logic probes).

- Design permanent projects with **maintenance** in mind. The more convenient you make maintenance tasks, the more likely they will get done.

- **Always document and save your work**. Circuits lacking schematic diagrams are more difficult to troubleshoot than documented circuits. Similarly, circuit construction is simpler when a schematic diagram precedes construction. Experimental results are easier to interpret when comprehensively recorded. Consider modern videorecording technology for this purpose where appropriate.

- **Record your steps** when troubleshooting. **Talk to yourself** when solving problems. These simple steps clarify thought and simplify identification of errors.

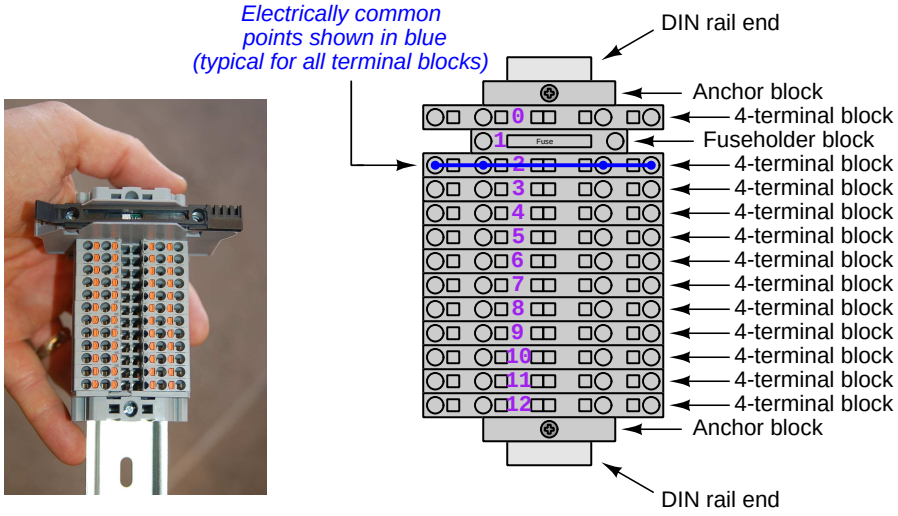
[4] An example of a necessary resistor value much less than 1 kΩ is a *shunt resistor* used to produce a small voltage drop for the purpose of sensing current in a circuit. Such shunt resistors must be low-value in order not to impose an undue load on the rest of the circuit. An example of a necessary resistor value much greater than 100 kΩ is an electrostatic *drain resistor* used to dissipate stored electric charges from body capacitance for the sake of preventing damage to sensitive semiconductor components, while also preventing a path for current that could be dangerous to the person (i.e. shock).

[5] Admittedly, solderless breadboards are very useful for constructing complex electronic circuits with many components, especially DIP-style integrated circuits (ICs), but they tend to give trouble with connection integrity after frequent use. An alternative for projects using low counts of ICs is to solder IC sockets into prototype printed circuit boards (PCBs) and run wires from the soldered pins of the IC sockets to terminal blocks where reliable temporary connections may be made.

8.1.3 Terminal blocks for circuit construction

Terminal blocks are the standard means for making electric circuit connections in industrial systems. They are also quite useful as a learning tool, and so I highly recommend their use in lieu of solderless breadboards[6]. Terminal blocks provide highly reliable connections capable of withstanding significant voltage and current magnitudes, and they force the builder to think very carefully about component layout which is an important mental practice. Terminal blocks that mount on standard 35 mm DIN rail[7] are made in a wide range of types and sizes, some with built-in disconnecting switches, some with built-in components such as rectifying diodes and fuseholders, all of which facilitate practical circuit construction.

I recommend every student of electricity build their own terminal block array for use in constructing experimental circuits, consisting of several terminal blocks where each block has at least 4 connection points all electrically common to each other[8] and at least one terminal block that is a fuse holder for overcurrent protection. A pair of anchoring blocks hold all terminal blocks securely on the DIN rail, preventing them from sliding off the rail. Each of the terminals should bear a number, starting from 0. An example is shown in the following photograph and illustration:



Screwless terminal blocks (using internal spring clips to clamp wire and component lead ends) are preferred over screw-based terminal blocks, as they reduce assembly and disassembly time, and also minimize repetitive wrist stress from twisting screwdrivers. Some screwless terminal blocks require the use of a special tool to release the spring clip, while others provide buttons[9] for this task which may be pressed using the tip of any suitable tool.

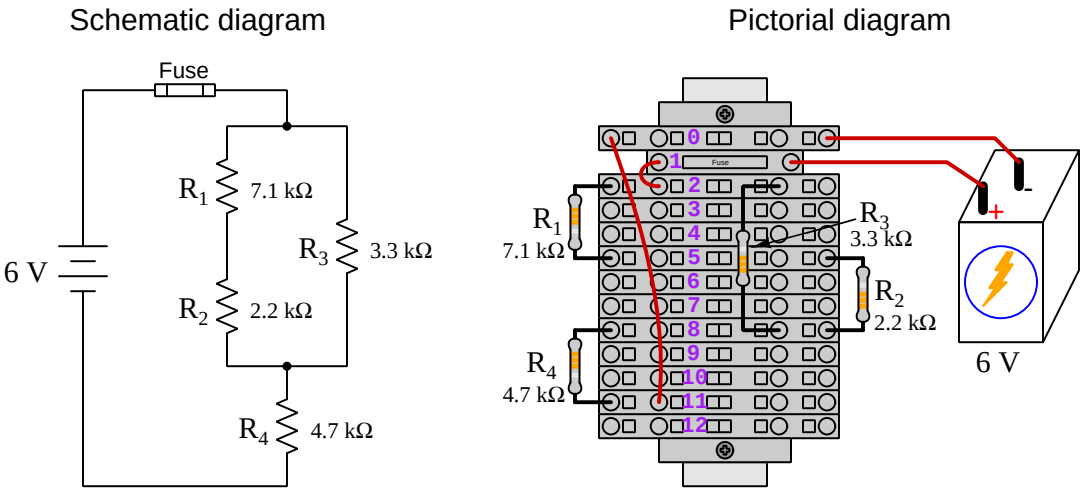
[6] Solderless breadboard are preferable for complicated electronic circuits with multiple integrated "chip" components, but for simpler circuits I find terminal blocks much more practical. An alternative to solderless breadboards for "chip" circuits is to solder chip sockets onto a PCB and then use wires to connect the socket pins to terminal blocks. This also accommodates *surface-mount* components, which solderless breadboards do not.

[7] DIN rail is a metal rail designed to serve as a mounting point for a wide range of electrical and electronic devices such as terminal blocks, fuses, circuit breakers, relay sockets, power supplies, data acquisition hardware, etc.

[8] Sometimes referred to as *equipotential*, *same-potential*, or *potential distribution* terminal blocks.

[9] The small orange-colored squares seen in the above photograph are buttons for this purpose, and may be actuated by pressing with any tool of suitable size.

The following example shows how such a terminal block array might be used to construct a series-parallel resistor circuit consisting of four resistors and a battery:



Numbering on the terminal blocks provides a very natural translation to SPICE[10] netlists, where component connections are identified by terminal number:

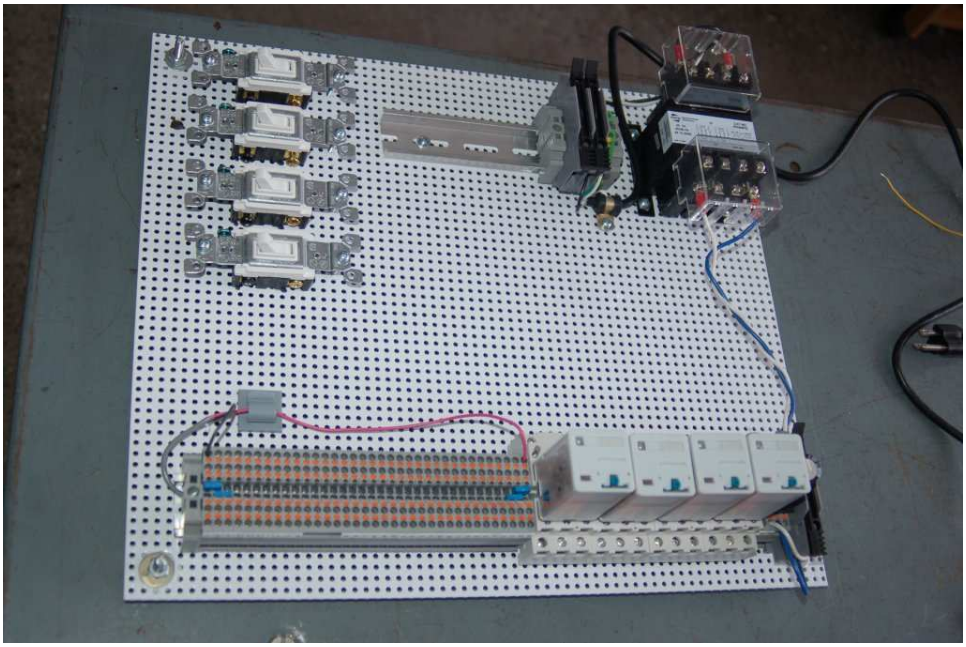
```
* Series-parallel resistor circuit
v1 1 0 dc 6
r1 2 5 7100
r2 5 8 2200
r3 2 8 3300
r4 8 11 4700
rjmp1 1 2 0.01
rjmp2 0 11 0.01
.op
.end
```

Note the use of "jumper" resistances `rjmp1` and `rjmp2` to describe the wire connections between terminals 1 and 2 and between terminals 0 and 11, respectively. Being resistances, SPICE requires a resistance value for each, and here we see they have both been set to an arbitrarily low value of 0.01 Ohm realistic for short pieces of wire.

Listing all components and wires along with their numbered terminals happens to be a useful documentation method for any circuit built on terminal blocks, independent of SPICE. Such a "wiring sequence" may be thought of as a *non-graphical description* of an electric circuit, and is exceptionally easy to follow.

[10] SPICE is computer software designed to analyze electrical and electronic circuits. Circuits are described for the computer in the form of *netlists* which are text files listing each component type, connection node numbers, and component values.

An example of a more elaborate terminal block array is shown in the following photograph, with terminal blocks and "ice-cube" style electromechanical relays mounted to DIN rail, which is turn mounted to a perforated subpanel[11]. This "terminal block board" hosts an array of thirty five undedicated terminal block sections, four SPDT toggle switches, four DPDT "ice-cube" relays, a step-down control power transformer, bridge rectifier and filtering capacitor, and several fuses for overcurrent protection:



Four plastic-bottomed "feet" support the subpanel above the benchtop surface, and an unused section of DIN rail stands ready to accept other components. Safety features include electrical bonding of the AC line power cord's ground to the metal subpanel (and all metal DIN rails), mechanical strain relief for the power cord to isolate any cord tension from wire connections, clear plastic finger guards covering the transformer's screw terminals, as well as fused overcurrent protection for the 120 Volt AC line power and the transformer's 12 Volt AC output. The perforated holes happen to be on $\frac{1}{4}$ inch centers with a diameter suitable for tapping with 6-32 machine screw threads, their presence making it very easy to attach other sections of DIN rail, printed circuit boards, or specialized electrical components directly to the grounded metal subpanel. Such a "terminal block board" is an inexpensive[12] yet highly flexible means to construct physically robust circuits using industrial wiring practices.

[11] An electrical *subpanel* is a thin metal plate intended for mounting inside an electrical enclosure. Components are attached to the subpanel, and the subpanel in turn bolts inside the enclosure. Subpanels allow circuit construction outside the confines of the enclosure, which speeds assembly. In this particular usage there is no enclosure, as the subpanel is intended to be used as an open platform for the convenient construction of circuits on a benchtop by students. In essence, this is a modern version of the traditional *breadboard* which was literally a wooden board such as might be used for cutting loaves of bread, but which early electrical and electronic hobbyists used as platforms for the construction of circuits.

[12] At the time of this writing (2019) the cost to build this board is approximately $250 US dollars.

8.1.4 Conducting experiments

An *experiment* is an exploratory act, a test performed for the purpose of assessing some proposition or principle. Experiments are the foundation of the *scientific method*, a process by which careful observation helps guard against errors of speculation. All good experiments begin with an *hypothesis*, defined by the American Heritage Dictionary of the English Language as:

> An assertion subject to verification or proof, as (a) A proposition stated as a basis for argument or reasoning. (b) A premise from which a conclusion is drawn. (c) A conjecture that accounts, within a theory or ideational framework, for a set of facts and that can be used as a basis for further investigation.

Stated plainly, an hypothesis is an *educated guess* about cause and effect. The correctness of this initial guess matters little, because any well-designed experiment will reveal the truth of the matter. In fact, *incorrect* hypotheses are often the most valuable because the experiments they engender lead us to surprising discoveries. One of the beautiful aspects of science is that it is more focused on the process of *learning* than about the status of *being correct*[13]. In order for an hypothesis to be valid, it must be testable[14], which means it must be a claim possible to refute given the right data. Hypotheses impossible to critique are useless.

Once an hypothesis has been formulated, an experiment must be designed to test that hypothesis. A well-designed experiment requires careful regulation of all relevant variables, both for personal safety and for prompting the hypothesized results. If the effects of one particular variable are to be tested, the experiment must be run multiple times with different values of (only) that particular variable. The experiment set up with the "baseline" variable set is called the *control*, while the experiment set up with different value(s) is called the *test* or *experimental*.

For some hypotheses a viable alternative to a physical experiment is a *computer-simulated experiment* or even a *thought experiment*. Simulations performed on a computer test the hypothesis against the physical laws encoded within the computer simulation software, and are particularly useful for students learning new principles for which simulation software is readily available[15].

[13] Science is more about clarifying our view of the universe through a systematic process of error detection than it is about proving oneself to be right. Some scient*ists* may happen to have large egos – and this may have more to do with the ways in which large-scale scientific research is *funded* than anything else – but scientific *method* itself is devoid of ego, and if embraced as a practical philosophy is quite an effective stimulant for humility. Within the education system, scientific method is particularly valuable for helping students break free of the crippling fear of *being wrong*. So much emphasis is placed in formal education on assessing correct retention of facts that many students are fearful of saying or doing anything that might be perceived as a mistake, and of course making mistakes (i.e. having one's hypotheses disproven by experiment) is an indispensable tool for learning. Introducing science in the classroom – *real* science characterized by individuals forming actual hypotheses and testing those hypotheses by experiment – helps students become self-directed learners.

[14] This is the principle of *falsifiability*: that a scientific statement has value only insofar as it is liable to disproof given the requisite experimental evidence. Any claim that is unfalsifiable – that is, a claim which can *never* be disproven by any evidence whatsoever – could be completely wrong and we could never know it.

[15] A very pertinent example of this is learning how to analyze electric circuits using simulation software such as SPICE. A typical experimental cycle would proceed as follows: (1) Find or invent a circuit to analyze; (2) Apply your analytical knowledge to that circuit, predicting all voltages, currents, powers, etc. relevant to the concepts you are striving to master; (3) Run a simulation on that circuit, collecting "data" from the computer when complete; (4) Evaluate whether or not your hypotheses (i.e. predicted voltages, currents, etc.) agree with the computer-generated results; (5) If so, your analyses are (provisionally) correct – if not, examine your analyses and the computer simulation again to determine the source of error; (6) Repeat this process as many times as necessary until you achieve mastery.

Thought experiments are useful for detecting inconsistencies within your own understanding of some subject, rather than testing your understanding against physical reality.

Here are some general guidelines for conducting experiments:

- The clearer and more specific the hypothesis, the better. Vague or unfalsifiable hypotheses are useless because they will fit *any* experimental results, and therefore the experiment cannot teach you anything about the hypothesis.
- Collect as much data (i.e. information, measurements, sensory experiences) generated by an experiment as is practical. This includes the time and date of the experiment, too!
- *Never* discard or modify data gathered from an experiment. If you have reason to believe the data is unreliable, write notes to that effect, but never throw away data just because you think it is untrustworthy. It is quite possible that even "bad" data holds useful information, and that someone else may be able to uncover its value even if you do not.
- Prioritize *quantitative* data over *qualitative* data wherever practical. Quantitative data is more specific than qualitative, less prone to subjective interpretation on the part of the experimenter, and amenable to an arsenal of analytical methods (e.g. statistics).
- Guard against your own bias(es) by making your experimental results available to others. This allows other people to scrutinize your experimental design and collected data, for the purpose of detecting and correcting errors you may have missed. Document your experiment such that others may independently replicate it.
- Always be looking for sources of error. No physical measurement is perfect, and so it is impossible to achieve *exact* values for any variable. Quantify the amount of uncertainty (i.e. the "tolerance" of errors) whenever possible, and be sure your hypothesis does not depend on precision better than this!
- Always remember that scientific confirmation is provisional – no number of "successful" experiments will prove an hypothesis true for all time, but a single experiment can disprove it. Put into simpler terms, *truth is elusive but error is within reach.*
- Remember that scientific method is about *learning*, first and foremost. An unfortunate consequence of scientific triumph in modern society is that science is often viewed by non-practitioners as an unerring source of truth, when in fact science is an ongoing process of challenging existing ideas to probe for errors and oversights. This is why it is perfectly acceptable to have a failed hypothesis, and why the only truly failed experiment is one where nothing was learned.

The following is an example of a well-planned and executed experiment, in this case a physical experiment demonstrating Ohm's Law.

```
Planning Time/Date = 09:30 on 12 February 2019

HYPOTHESIS: the current through any resistor should be exactly proportional
to the voltage impressed across it.

PROCEDURE: connect a resistor rated 1 k Ohm and 1/4 Watt to a variable-voltage
DC power supply.  Use an ammeter in series to measure resistor current and
a voltmeter in parallel to measure resistor voltage.

RISKS AND MITIGATION: excessive power dissipation may harm the resistor and/
or pose a burn hazard, while excessive voltage poses an electric shock hazard.
30 Volts is a safe maximum voltage for laboratory practices, and according to
Joule's Law a 1000 Ohm resistor will dissipate 0.25 Watts at 15.81 Volts
(P = V^2 / R), so I will remain below 15 Volts just to be safe.

Experiment Time/Date = 10:15 on 12 February 2019

DATA COLLECTED:
  (Voltage)    (Current)         (Voltage)   (Current)
   0.000 V   =  0.000 mA          8.100    =  7.812 mA
   2.700 V   =  2.603 mA          10.00 V  =  9.643 mA
   5.400 V   =  5.206 mA          14.00 V  =  13.49 mA

Analysis Time/Date = 10:57 on 12 February 2019

ANALYSIS: current definitely increases with voltage, and although I expected
exactly one milliAmpere per Volt the actual current was usually less than
that.  The voltage/current ratios ranged from a low of 1036.87 (at 8.1 Volts)
to a high of 1037.81 (at 14 Volts), but this represents a variance of only
-0.0365% to +0.0541% from the average, indicating a very consistent
proportionality -- results consistent with Ohm's Law.

ERROR SOURCES: one major source of error is the resistor's value itself.  I
did not measure it, but simply assumed color bands of brown-black-red meant
exactly 1000 Ohms.  Based on the data I think the true resistance is closer
to 1037 Ohms.  Another possible explanation is multimeter calibration error.
However, neither explains the small positive and negative variances from the
average.  This might be due to electrical noise, a good test being to repeat
the same experiment to see if the variances are the same or different.  Noise
should generate slightly different results every time.
```

The following is an example of a well-planned and executed *virtual* experiment, in this case demonstrating Ohm's Law using a computer (SPICE) simulation.

```
Planning Time/Date = 12:32 on 14 February 2019

HYPOTHESIS: for any given resistor, the current through that resistor should be
exactly proportional to the voltage impressed across it.

PROCEDURE: write a SPICE netlist with a single DC voltage source and single
1000 Ohm resistor, then use NGSPICE version 26 to perform a "sweep" analysis
from 0 Volts to 25 Volts in 5 Volt increments.

    * SPICE circuit
    v1 1 0 dc
    r1 1 0 1000
    .dc v1 0 25 5
    .print dc v(1) i(v1)
    .end

RISKS AND MITIGATION: none.

DATA COLLECTED:
       DC transfer characteristic  Thu Feb 14 13:05:08  2019
    ------------------------------------------------------------
    Index   v-sweep         v(1)            v1#branch
    ------------------------------------------------------------
    0       0.000000e+00    0.000000e+00    0.000000e+00
    1       5.000000e+00    5.000000e+00    -5.00000e-03
    2       1.000000e+01    1.000000e+01    -1.00000e-02
    3       1.500000e+01    1.500000e+01    -1.50000e-02
    4       2.000000e+01    2.000000e+01    -2.00000e-02
    5       2.500000e+01    2.500000e+01    -2.50000e-02

Analysis Time/Date = 13:06 on 14 February 2019

ANALYSIS: perfect agreement between data and hypothesis -- current is precisely
1/1000 of the applied voltage for all values.  Anything other than perfect
agreement would have probably meant my netlist was incorrect.  The negative
current values surprised me, but it seems this is just how SPICE interprets
normal current through a DC voltage source.

ERROR SOURCES: none.
```

As gratuitous as it may seem to perform experiments on a physical law as well-established as Ohm's Law, even the examples listed previously demonstrate opportunity for real learning. In the physical experiment example, the student should identify and explain why their data does not perfectly agree with the hypothesis, and this leads them naturally to consider sources of error. In the computer-simulated experiment, the student is struck by SPICE's convention of denoting regular current through a DC voltage source as being *negative* in sign, and this is also useful knowledge for future simulations. Scientific experiments are most interesting when things *do not* go as planned!

Aside from verifying well-established physical laws, simple experiments are extremely useful as educational tools for a wide range of purposes, including:

- Component familiarization (e.g. *Which terminals of this switch connect to the NO versus NC contacts?*)
- System testing (e.g. *How heavy of a load can my AC-DC power supply source before the semiconductor components reach their thermal limits?*)
- Learning programming languages (e.g. *Let's try to set up an "up" counter function in this PLC!*)

Above all, the priority here is to inculcate the habit of hypothesizing, running experiments, and analyzing the results. This experimental cycle not only serves as an excellent method for self-directed learning, but it also works exceptionally well for troubleshooting faults in complex systems, and for these reasons should be a part of every technician's and every engineer's education.

8.1.5 Constructing projects

Designing, constructing, and testing projects is a very effective means of practical education. Within a formal educational setting, projects are generally chosen (or at least vetted) by an instructor to ensure they may be reasonably completed within the allotted time of a course or program of study, and that they sufficiently challenge the student to learn certain important principles. In a self-directed environment, projects are just as useful as a learning tool but there is some risk of unwittingly choosing a project beyond one's abilities, which can lead to frustration.

Here are some general guidelines for managing projects:

- Define your goal(s) before beginning a project: what do you wish to achieve in building it? What, exactly, should the completed project *do*?
- Analyze your project prior to construction. Document it in appropriate forms (e.g. schematic diagrams), predict its functionality, anticipate all associated risks. In other words, *plan ahead*.
- Set a reasonable budget for your project, and stay within it.
- Identify any deadlines, and set reasonable goals to meet those deadlines.
- Beware of *scope creep*: the tendency to modify the project's goals before it is complete.
- Document your progress! An easy way to do this is to use photography or videography: take photos and/or videos of your project as it progresses. Document failures as well as successes, because both are equally valuable from the perspective of learning.

8.2 Experiment: Relay circuit implementation of an arbitrary truth table

Conduct an experiment demonstrating how a circuit using toggle switches (inputs) and at least one electromechanical relay may be used to implement an arbitrary truth table for a three-input combinational logic function. A truth table template is given here for your use, to arbitrarily write "1" and "0" states in the output column:

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

EXPERIMENT CHECKLIST:

- Prior to experimentation:

  ☑ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.

  ☑ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).

  ☑ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.

- During experimentation:

  ☑ Safe practices followed at all times (e.g. no contact with energized circuit).

  ☑ Correct equipment usage according to manufacturer's recommendations.

  ☑ All data collected, ideally quantitative with full precision (i.e. no rounding).

- After each experimental run:

  ☑ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.

  ☑ Identify any uncontrolled sources of error in the experiment.

- After all experimental re-runs:
  - ☑ Save all data for future reference.
  - ☑ Write an analysis of experimental results and lessons learned.

Challenges

- Identify a truth table function possible to implement with no relay at all.
- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- It is possible to implement your chosen logic function using *only* toggle switches and no relays?
- Describe a different relay-based circuit that would implement the exact same logic function.

8.3 Project: Combinational gate circuit driving 120 VAC load

Design a combinational logic circuit using toggle switches (inputs) and integrated circuit (IC) logic gate "chips" to implement an arbitrary truth table, and power a small 120 VAC load with the function's output state. No electromechanical relays should be used in this project to perform logic functionality. If any relay is to be used, it should strictly be for interposing to the load. A truth table template is given here for your use, to arbitrarily write "1" and "0" states in the output column:

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

PROJECT CHECKLIST:

- Prior to construction:
  - ☑ Prototype diagram(s) and description of project scope.
  - ☑ Risk assessment/mitigation plan.
  - ☑ Timeline and action plan.
- During construction:
  - ☑ Safe work habits (e.g. no contact made with energized circuit at any time).
  - ☑ Correct equipment usage according to manufacturer's recommendations.
  - ☑ Timeline and action plan amended as necessary.
  - ☑ Maintain the originally-planned project scope (i.e. avoid adding features!).
- After completion:
  - ☑ All functions tested against original plan.
  - ☑ Full, accurate, and appropriate documentation of all project details.

- [√] Complete bill of materials.
- [√] Written summary of lessons learned.

Challenges

- Describe a different logic gate circuit that would implement the exact same logic function.

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

"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.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

- 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[1] 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[2] 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[3] 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

[1] 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.

[2] 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.

[3] 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 characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told *again* how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to *require* those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

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.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students' thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this "inverted" format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the "inverted" session instructor in order that students never feel discouraged by having their errors exposed. *Everyone* makes mistakes from time to time, and learning is a lifelong process! Part of the instructor's job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.

To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, *the fundamental goal of education is for each student to learn to think clearly and independently.* This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student's responsibility to do that hard work ultimately limits that student's potential and thereby does real harm.

- It is not only permissible, but encouraged, to answer a student's question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.

- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one's own reasoning.

- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an "Outline and reflections" question as well as a "Foundational concepts" question in the Conceptual reasoning section, to prompt reflective reading.

- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to *variations* of that question. This is why module questions typically contain multiple "Challenges" suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own "Challenges" as they are able, in order to have a multitude of ways ready to probe students' understanding.

- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done "live" before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.

Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples.*
- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.
- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.
- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.
- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.
- 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[4] effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge[5] one another.

To high standards of education,

Tony R. Kuphaldt

[4] 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.

[5] Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one's life through the improvement of clear and independent thought, literacy, expression, and various practical skills.

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

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

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.

6 May 2025 – minor edits to the Tutorial, and some edits made to instructor notes as well.

4 May 2025 – changed the "Universality" Tutorial section title to "Universal logic functions", and also edited some of that section's text for readability.

5 December 2024 – added more bullet-points to the Introduction chapter list of recommendations for instructors.

6 November 2024 – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors.

14 April 2024 – added example numbers to the "Circuits from two-input truth tables" and "Circuits from three-input truth tables" Quantitative Reasoning question, added more timing diagram sub-questions to the "Timing diagrams for gate circuits" Quantitative Reasoning question, and fixed a minor typographical error in this Version history chapter.

11 December 2023 – minor edits to the Tutorial.

14-16 September 2023 – added a Case Tutorial section showcasing a gallery of practical logic gate application circuits.

25 April 2023 – Added a Case Tutorial section on timing diagrams applied to simple combinational logic circuits.

19 April 2023 – fixed typographical error where the text said "division" and should have said "addition". This fix courtesy of Daniel Wing.

15 January 2023 – added a Technical Reference section on IC logic families.

15 December 2022 – altered the SOP table-to-circuit example to avoid confusion, based on a recommendation by Nolan Call. The original SOP expression was $\overline{A}B\overline{C} + AB\overline{C}$, which was potentially confusing because the two terms' bit states were opposite which meant a reader might not properly associate 0 and 1 states with each term's variable complementation. The new truth table and circuit diagrams now reflect the Boolean expression $\overline{A}B\overline{C} + ABC$ where there exists no such ambiguity. Also corrected an unrelated typographical error where I described an AND gate as being "NAND" instead.

12 December 2022 – edited image_1316 to make all inverter gate symbols consistent.

8-9 December 2022 – added more references in the Tutorial to the equivalence of AND = Boolean multiply = series contacts, OR = Boolean add = parallel contacts, and NOT = Boolean inversion = normally-closed contacts. Added numerical labels to image_1306 as well. Also added a new Conceptual Reasoning question based on a schematic capture image showing a creative use of spare NAND gates.

28 November 2022 – placed questions at the top of the itemized list in the Introduction chapter prompting students to devise experiments related to the tutorial content.

25 May 2022 – renamed the Tutorial section "Gate universality" to "Universality" because this concept relates to abstract logic functions and is not specific to logic *gates*.

5 May 2022 – added Historical Reference section on NASA's Apollo guidance computer which was made of just NOR gates.

20 April 2022 – added more questions to the Introduction chapter.

13 December 2021 – added more explanatory text describing why we might use SOP or POS to translate a truth table into a working circuit.

26 November 2021 – added section title to the logic function review at the start of the Tutorial.

17-19 November 2021 – shortened some of the Quantitative Reasoning question names (e.g. "Boolean expressions and circuits from..." is now "Circuits from..."). Also divided the "Truth tables into circuits" section of the Tutorial into subsections. Finally, added a Case Tutorial chapter with examples showing translation from truth tables into actual circuits.

9 May 2021 – commented out or deleted empty chapters.

9 December 2020 – minor additions to the Introduction chapter.

5 December 2020 – added a requirement to the "Gate circuits from Boolean expressions" and "Gate circuit from a truth table" questions that the combinational logic circuit be able to drive a heavy DC load.

15 October 2020 – replaced `+` and `*` operators in C++ programming examples with `||` and `&&`.

12 October 2020 – minor additions to the Introduction chapter.

1 October 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.

15 May 2020 – added more questions.

1 April 2020 – added a "frame" box around the C++ code for the combinational logic circuit simulation that should have been there originally. My formatting policy for coding examples is to print source code as verbatim text inside of a frame, and to print the output of a program's execution as just verbatim text (with no frame).

23 March 2020 – added more challenge questions to "Gate circuits from Boolean expressions" Quantitative Reasoning problem.

14 March 2020 – minor edits to the Tutorial, mostly dealing with the alternative method to POS (using SOP when the truth table contains a majority of 1 states). Also correct some typographical errors.

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

2 January 2020 – moved C++ demonstration program of a combinational logic circuit from mod_boolean to mod_comblogic. Also deleted frames from text output, reserving those just for source code listings.

23 December 2019 – minor edit to a question.

13 November 2019 – added more questions.

20 April 2019 – clarified the use of logic gate ICs to perform logical function (not relays) in one of the projects.

15 April 2019 – added Quantitative Reasoning questions centered around deriving Boolean, gate, and ladder logic circuits from truth tables.

11 April 2019 – added two Quantitative Reasoning questions centered around the use of Python to evaluate combinational logic functions.

27 March 2019 – added questions to Conceptual, Quantitative, and Diagnostic Reasoning sections. Also added Claude Shannon's work on Boolean algebra to the Historical References chapter.

14 March 2019 – added more content to the Tutorial, corrected a few typographical errors, edited two images.

13 March 2019 – added an experiment, to build a circuit fulfilling some arbitrary logic function defined by a truth table. Also, corrected a typo in the module's title (it was "Combinational Logic module" when it should have been "Combinational Logic")!

10 March 2019 – continued writing Tutorial chapter, finished Foundational Concepts list.

5 March 2019 – continued writing Tutorial chapter.

3 March 2019 – document first created.

Index