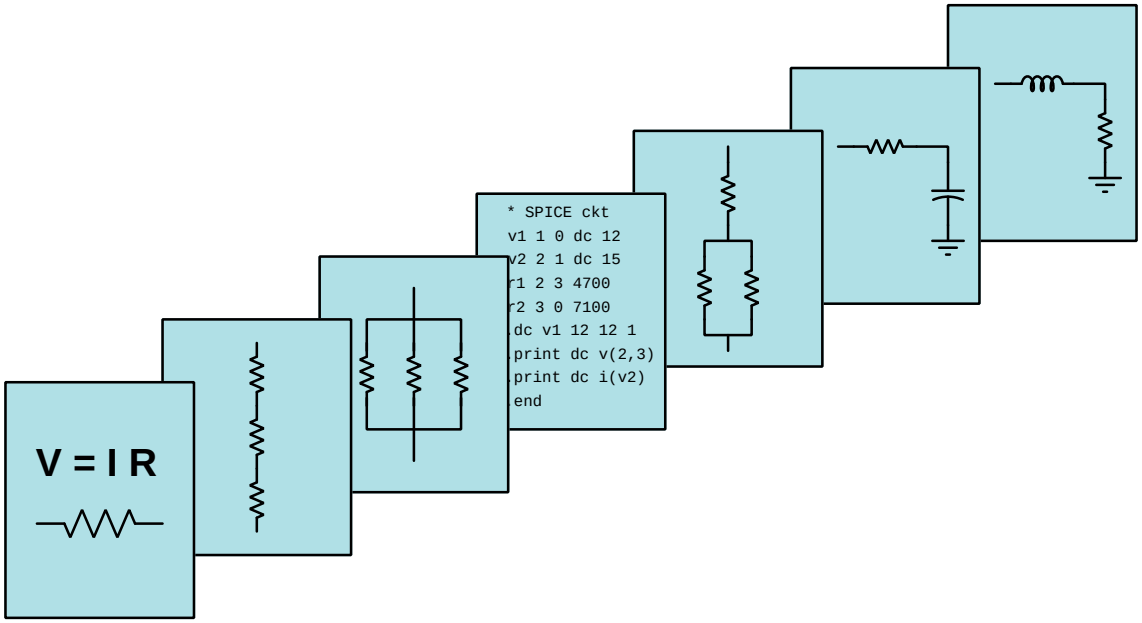
Modular Electronics Learning (ModEL) project



Programmable Logic ICs

Last update = 30 January 2024

Contents

Chapter 1

Introduction

Logic gates form the foundation of digital electronic circuits, and when these essential "building blocks" are interconnected to form more complex combinational and multivibrator (i.e. latching) networks we may use them to construct all manner of useful digital systems including binary arithmetic modules, memory devices, digital communication networks, and digital computers. *Programmable* logic devices are built in such a way that the interconnections between logic gates are determined by the end-user, and in many cases may be re-wired again and again as desired to create new digital functions.

Programmable logic shares many similarities with microprocessor-based systems, but they are different in many important ways as well. Microprocessors consist of fixed networks of logic gates, designed to read and execute *instructions* held in memory – the *program* executed by any microprocessor consists of these instructions directing the fixed-logic networks. Programmable logic, on the other hand, is where the logic gates are not fixed to one another at all, but may have their inputs and outputs redirected as desired after manufacture – in other words, the "program" inserted into a programmable logic device literally re-wires the gates to each other, after which those gates act directly on any data fed to them.

Applications for programmable logic are every bit as expansive as for microprocessors, but each has its own strengths. Microprocessors, given their instruction-centric design, excel at carrying out sequential and/or repetitive tasks. Programmable logic devices, given their connection-centric design, excel at tasks requiring many logic elements to act in parallel, even independently of each other.

Important concepts related to programmable logic include **gates**, **flip-flops**, **memory**, **addresses**, **data**, **Boolean algebra**, **SOP expressions**, **look-up tables**, **fuses**, **antifuses**, **volatile** versus **non-volatile** memory, **state machine**, **register**, **tri-state logic**, **RTL**, and **hardware description languages**.

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

- How might an experiment be designed and conducted to explore the concept of fuse-programmable logic? For example, what kind of simple circuit could you fabricate using fuses

to form disruptible connections between components, such that blowing certain fuses and not others would form specific logic functions?

- What are some practical applications of programmable logic ICs?
- What are some of the different ways in which we may create easily re-configurable digital logic networks?
- How may we translate a truth table into a Boolean SOP expression?
- How may we translate a truth table into a Boolean Negative-SOP expression?
- What factor(s) determine whether an SOP or a Negative-SOP expression would be more efficient in describing a given truth table?
- What is the difference between *behavioral* versus *structural* HDL code?
- What is a *testbench* in HDL and what is it useful for?
- How do CPLDs and FPGAs differ from one another?
- What is an *ASIC*, and how does this technology differ from CPLDs and FPGAs?
- Why are Hardware Description Languages useful for configuring programmable ICs?

Chapter 2

Tutorial

2.1 Programmable versus fixed logic

If a collection of logic gates, multivibrators (e.g. flip-flops), registers, and other digital logic elements are needed to construct a practical digital system, the designer has several options available to them. The first and most obvious of these is to interconnect a set of digital integrated circuit (IC) elements together on a printed circuit board (PCB), but this approach suffers from low density: i.e. most logic packages only have a few gates, flip-flops, or registers on a single IC, thus requiring a great many of them together on a rather large PCB to create any complex digital system. Another approach is to place all the necessary logic elements onto just a few high-density ICs, preferably on a single IC, but if the digital system in question is unique it will require custom ICs to be fabricated. As one might guess, building a custom integrated circuit from nothing is a highly complex and expensive. Some IC manufacturers offer *Application-Specific Integrated Circuit* (*ASIC*) fabrication as a service to multiple customers, but even with this economy of scale it is quite expensive.

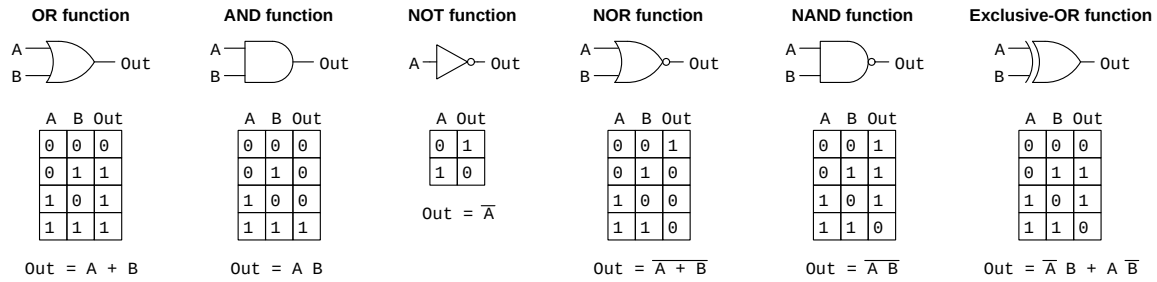
A practical alternative to designing and manufacturing customized hardware is to utilize digital components that are *programmable* in one way or another. In other words, if we create digital devices whose function may be specified after the time of manufacture by the writing of binary bit-states to memory elements within that device, we enable the creation of customized digital systems that don't require massive PCB layouts or one-of-a-kind ICs. A *microprocessor* is one such device, coupled together with memory circuits to retain this programming data. Microprocessors read programmed instructions from memory and then execute those instructions one step at a time. This is a versatile solution, but far slower than custom hardware because every function must be comprised of multiple steps carried out in sequence at finite speed.

A completely different type of programmable digital device is the focus of this Tutorial: a digital IC whose post-manufacture programming actually *re-wires* its internal circuitry to form the desired functions. In contrast to a microprocessor whose internal wiring is fixed, the type of digital IC we're about to explore contains a very large collection of logic gates as possibly other elements as well such as flip-flops and registers, whose interconnections are determined by the user of that device some time after manufacture. The principal advantage of this "programmed wiring" approach is that the programmed functions are free to operate simultaneously rather than to be restricted to one-step-at-a-time algorithms the way a microprocessor must do. In summary, microprocessors are fundamentally *sequential* devices whereas these programmable logic ICs are capable of *concurrent* operations. Whereas a microprocessor's ability to implement a digital function is principally limited by memory and by time, a programmable logic IC's ability to implement any function is principally limited by the number of logic elements it contains and the number of data lines it has inside to potentially connect those elements together in custom configurations.

Programmable IC technology is newer than that of microprocessors, exists in a diversity of forms lacking standardized nomenclature, with its programming generally considered to be more sophisticated and difficult to learn than that of microprocessors. To begin our exploration of this powerful technology, we will begin with a review of basic logic functions and then explore different ways of implementing those functions besides elementary logic gate circuits. Then, we will explore some of the modern forms of programmable logic ICs as well as the programming languages used to configure them.

2.2 Logic function review

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).

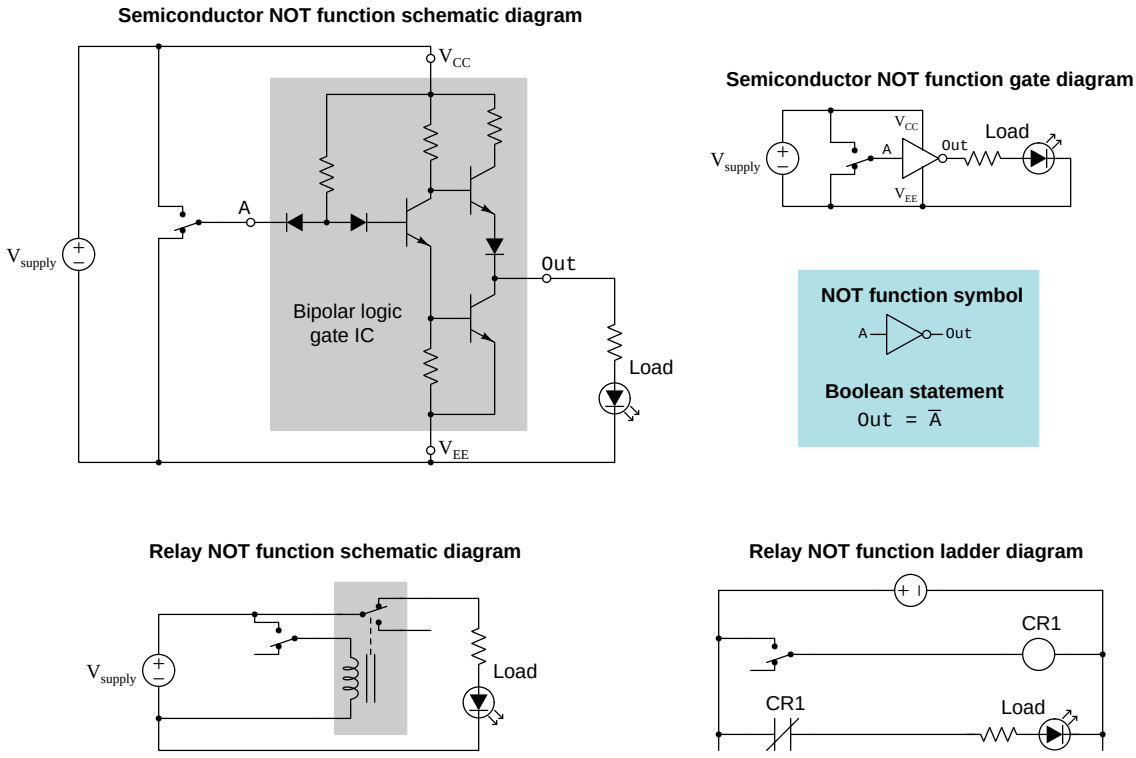
All of the two-input logic functions previously shown, with the exception of the Exclusive-OR (also called XOR), are available in versions having more than two inputs. A four-input OR function, for example, would have an expanded truth table with sixteen (2^4) rows, only the first of which has a 0 output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

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

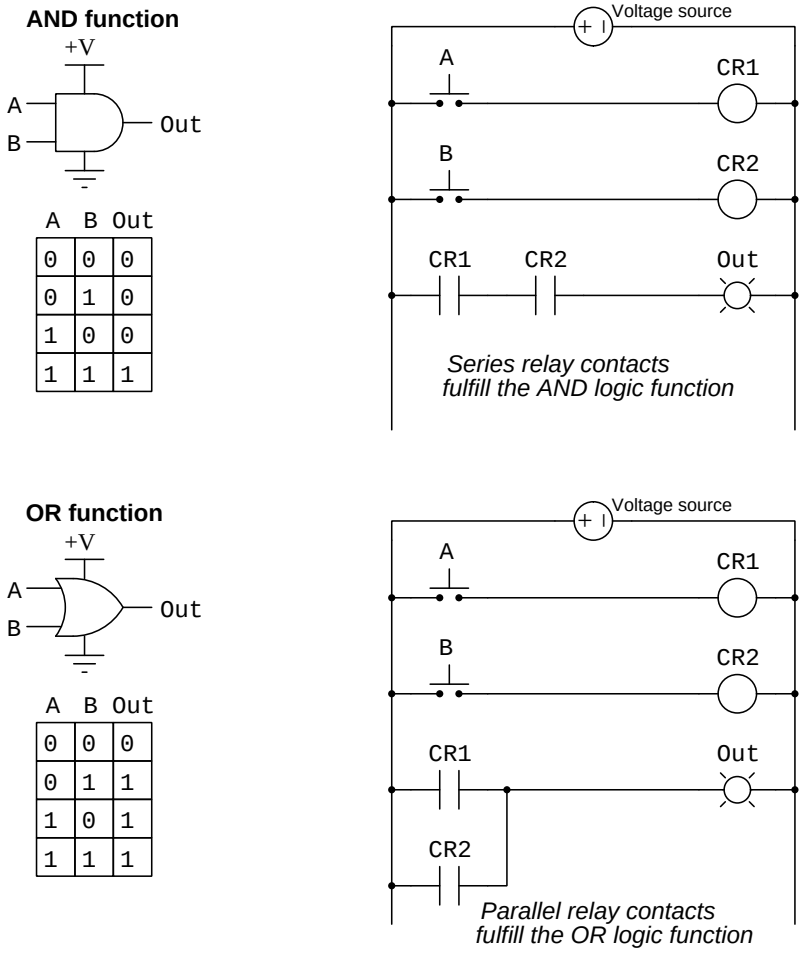
[1] When spoken, one generally says "A-bar" or "not-A" to represent the complement of A.

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.

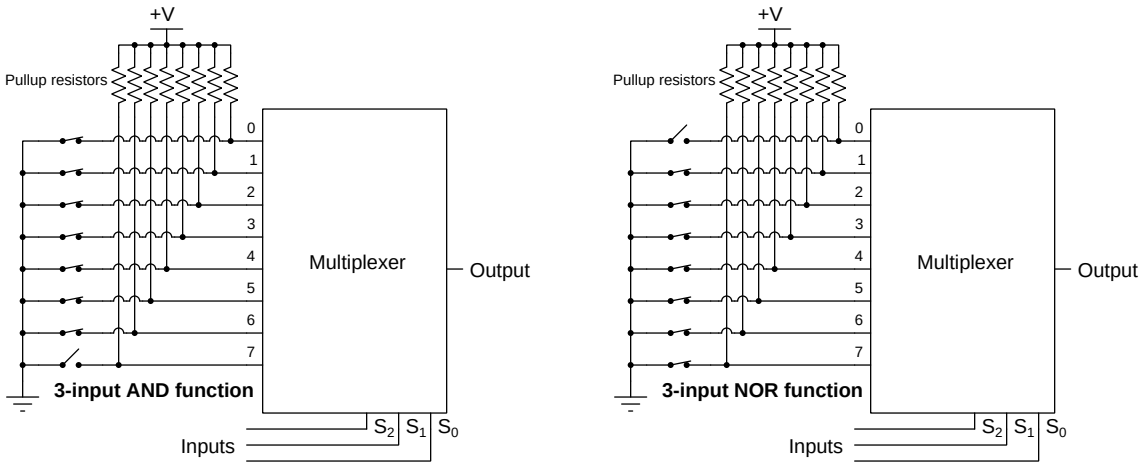
Basic logical functions such as AND and OR may be implemented using electromechanical relays just as they can using transistors. AND and OR functions in particular have direct relation to *series* and *parallel* contact connections, respectively. Please note that electrical power supply connections are 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.

2.3 Multiplexers as logic

One way to implement an arbitrary and re-configurable logic function is to use a *multiplexer* to "steer" arbitrary logic states to an output terminal at the command of a multi-bit code. If we consider the "select" lines to be the input lines of a logic function, and the single mux output line to be the output line of that same logic function, the mux input lines serve as connection points for "programmable" output states. Consider these simple examples:

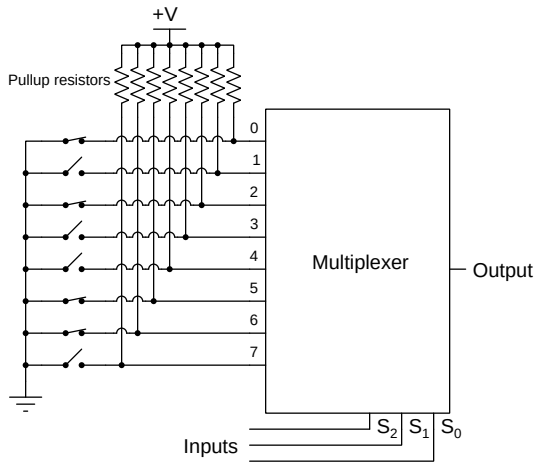


As we apply all possible high/low combinations to the three "select" input lines (S_2, S_1, and S_0) of these multiplexers, those binary values select one input channel at a time (e.g. when S_2 is low and S_1 is low, and S_0 is low, the mux selects the logic state applied to input terminal `0` to be "steered" to the output terminal.

The toggle switches connected to the multiplexer's input lines thus serve to *program* the desired logic function. Simply by setting these switches to different positions, we may specify any 3-input logic function desired.

To show just how arbitrary the logic function may be when implemented by a multiplexer, consider the following example. Here we see a truth table not adhering to any canonical logic function such as AND or NOR, but nevertheless implemented just as easily as any other logic function using the same multiplexer IC we saw in the two prior examples. The output states for each row of the truth table are "programmed" by the settings of the eight toggle switches connected to the multiplexer's input channel terminals:

S_2	S_1	S_0	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



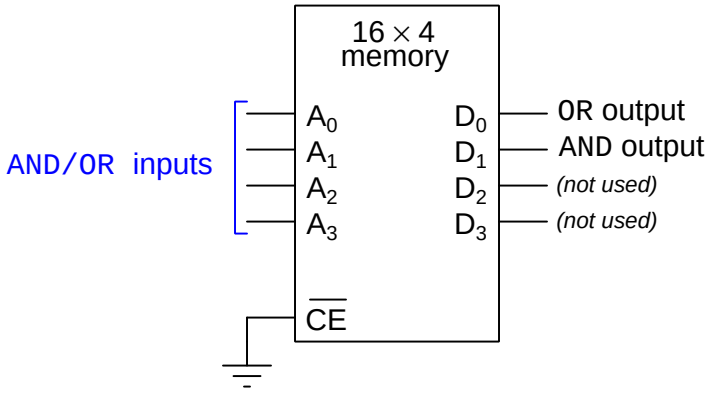
A simple thought experiment works well to explore how this circuit functions. Just imagine setting the three "select" input lines to various high (1) and low (0) states, one combination at a time, and see for yourself which input channel the mux selects for each combination. Remember that the three "select" input line states constitute bits of a three-bit binary number, the value of that number determining which input channel gets "steered" to the output:

- $S_2 = 0 \quad S_1 = 0 \quad S_0 = 0$ Output = 0 because channel 0's toggle switch is closed
- $S_2 = 0 \quad S_1 = 0 \quad S_0 = 1$ Output = 1 because channel 1's toggle switch is open
- $S_2 = 0 \quad S_1 = 1 \quad S_0 = 0$ Output = 0 because channel 2's toggle switch is closed
- $S_2 = 0 \quad S_1 = 1 \quad S_0 = 1$ Output = 1 because channel 3's toggle switch is open
- $S_2 = 1 \quad S_1 = 0 \quad S_0 = 0$ Output = 1 because channel 4's toggle switch is open
- $S_2 = 1 \quad S_1 = 0 \quad S_0 = 1$ Output = 0 because channel 5's toggle switch is closed
- $S_2 = 1 \quad S_1 = 1 \quad S_0 = 0$ Output = 0 because channel 6's toggle switch is closed
- $S_2 = 1 \quad S_1 = 1 \quad S_0 = 1$ Output = 1 because channel 7's toggle switch is open

2.4 Memory as logic

Another way to implement an arbitrary and re-configurable logic function is to use a *programmable memory IC* to generate the desired output state(s) at the command of a multi-bit code sent to the memory IC's address lines. Here we consider the "address" lines of the memory IC to be the input lines of a logic function and the "data" lines of that same memory IC to be the output lines of that same logic function. This is also known as a *look-up table* or *LUT* because the address word "looks up" whatever data word is stored in each memory location of the IC.

Consider the example of a simple 16 × 4 memory IC with four address lines and four data lines, programmed with the look-up table data shown in the following "hex dump" to implement both four-input AND as well as four-input OR functions:




```
   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00 0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3
```

According to this look-up table, the output pin D_0 will be in a "high" for any address but `0b0000`, which is to say that output goes "high" whenever *any* input (address line) is "high". This same table shows the next output pin (D_1) going to a "high" state only when all four of the input (address) lines are "high", representing a four-input AND function.

One of the strengths of a look-up table is that it may be programmed to represent any arbitrary logical function, not just canonical functions such as AND, OR, NAND, NOR, XOR, etc. If the memory array is a mask-type ROM then it means the look-up table must be programmed at the time of manufacture, which is ideal for mass-production applications where a great many logic devices must be produced, all having the exact same (fixed) functionality. If the memory array is EPROM in nature, then the IC leaves the manufacturer in a "blank" state ready to be programmed by the customer. Static RAM memory arrays are also suitable as look-up tables, and have the distinct advantage of being re-programmed on the fly, but must be initialized with the look-up table data upon every power-up cycle because their data is non-volatile.

2.5 Sum-of-Product logic expressions

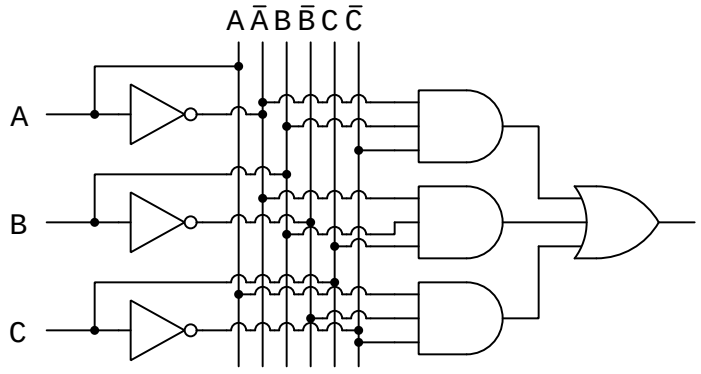
Any combinational logic function represented in truth-table form may be constructed of primitive logic gate types such as AND and OR using Boolean algebra sum-of-products (SOP) expressions. Take for example the following three-input logical function:

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

The output of this function is "high" (1) if ever A is low and B is high and C is low (010), or if ever A is low and B is high and C is high (011), or if ever A is high and B is low and C is low (100). The use of *and* and *or* verbiage is intentional here, as the point is to emphasize how each unique combination of A, B, and C logical states may be decoded using the AND function, and how the over-all summation of these combinations is really just an OR function. In Boolean algebra we represent the AND function as multiplication, and the OR function as addition, which means we may represent this logic function algebraically represented as follows:

$$\overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C}$$

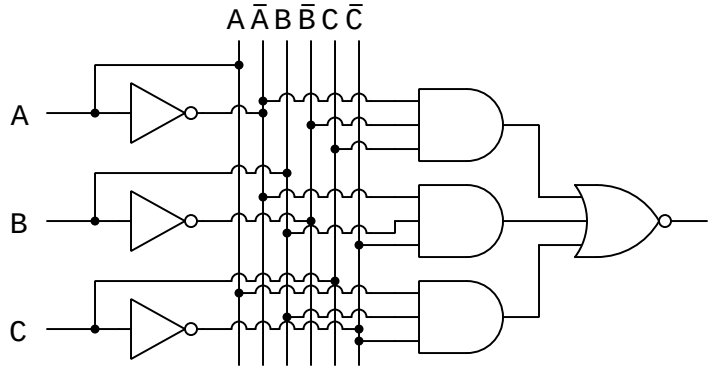
From here, it is a simple matter to sketch a combinational logic network of AND and OR gates to implement this truth table's function:



For functions having a greater number of "high" (1) output states than "low" (0) output states, we may efficiently apply the *negative-sum-of-products* strategy whereby we identify every unique combination of input states resulting in a low output, decode those states using AND (multiplication), and then sum them together using NOR (inverted addition). For example:

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

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



It should be noted that this same logical function could have been implemented using standard SOP form with AND gates feeding into a single OR gate (rather than into a single NOR gate), but doing so would have required *five* three-input AND gates rather than just three because the truth table contains five rows with "high" output states. The NOR-based approach is nothing more than a more gate-efficient method when the function in question has more "high" than "low" output states.

The fact that *any* combinational logic function whatsoever may be formed by simply combining primitive AND and OR gates (or NOR gates) suggests we might be able to create universal logic functions by simply connecting a large array of AND and OR gates together in the right manner. This fact is the basic concept behind simple *programmable* logic ICs, which we will explore next.

2.6 AND-OR programmable logic

A "simple" programmable logic IC is one containing an array of inverter and AND gates as well as either OR or NOR gates that are interconnectable in such a way as to implement arbitrary Boolean SOP expressions. What makes such an IC *programmable* is that these interconnections may be determined after the time of manufacture.

For many years circuit designers have had the option of purchasing *Application-Specific Integrated Circuits*, commonly referred to as *ASICs*, for custom digital logic needs. An ASIC consists of a completely custom IC, either outright produced by a single manufacturer large enough to operate its own semiconductor fabrication facilities or by a *silicon foundry* catering to highly knowledgeable customers producing their own designs. However, as one might guess ASICs are generally quite expensive because they are essentially *bespoke* (custom-designed) silicon chips. As such, ASICs only make sense when the application absolutely demands the most efficient use of silicon for a particular task, where unusual combinations of signals must be merged on the same IC[2], or the nature of the application demands a high level of secrecy[3] about the circuit's internal design, etc. For most other end-uses requiring custom digital functions, it makes more sense to use some form of integrated circuit containing a large array of gates that may be user-configured into networks rather than engineer and manufacture a novel IC design.

Programmable logic ICs were invented for just this purpose: to provide user-configurable digital logic functions in compact IC form. Many different types of programmable logic ICs have been invented and are currently available for use in digital design work, and several popular acronyms have been made to describe them including:

- **PLA** = Programmable Logic Array
- **PAL** = Programmable Array Logic
- **GAL** = Generic Array Logic
- **PLD** = Programmable Logic Device

Unfortunately, the definitions of these acronyms are not always consistent across manufacturers or in technical literature. For example, PLD (Programmable Logic Device) may be used to generically describe *any* digital logic IC that is user-programmable, or it may refer to a very specific category of programmable IC. Distinctions ostensibly made by these different acronyms are often blurred in practice as manufacturers update their product lines with more and better features while retaining legacy descriptors.

One of the most practical distinctions between different types of programmable logic ICs is whether or not they are *one-time programmable* (OTP) versus *re-programmable*. The earliest programmable devices utilized miniature semiconductor fuses or anti-fuses[4] to form connections

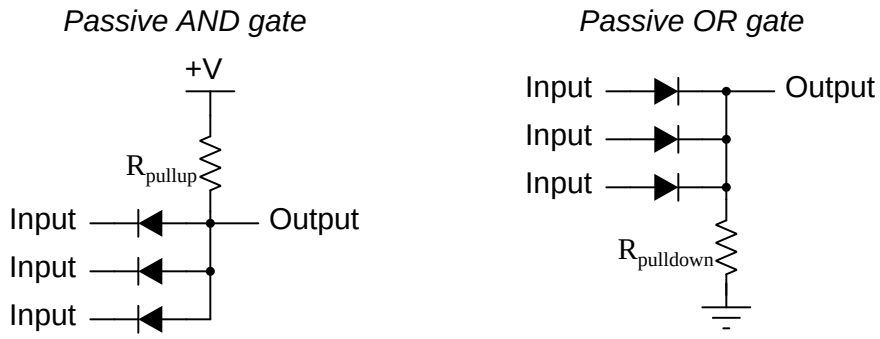
[2] For example, an IC that must process both digital and analog signals on one silicon die, not corresponding to any commonly-available mixed-signal IC.

[3] Consider ICs to be used in military hardware, where the design details could represent a security threat if reverse-engineered by any hostile party. It is not impossible to de-encapsulate an integrated circuit and study its internal design using tools such as scanning electron microscopes, but such tasks are far from easy. Certainly, reverse-engineering a digital system built from standardized ICs whose inner functions are already well-documented would be much easier!

[4] One popular way to create an anti-fuse on a silicon die is to form a special MOSFET with an insulating metal-oxide layer intended to break down and become conductive with sufficient voltage applied across it.

between the outputs of some gates and the inputs of others, making them programmable just once. After a fuse has been blown, it cannot be re-formed again; after an anti-fuse has been shorted, it cannot be opened again. Re-programmable devices, by contrast, use floating-gate MOSFETs or static RAM arrays to store the interconnection bits, and so may be configured and re-configured many times – a valuable feature for prototyping as well as firmware-upgradable products.

We may explore the concept of one-time programmable (OTP) logic using passive networks of diodes and resistors to form AND and OR functions, each of these primitive elements shown in the following schematics:

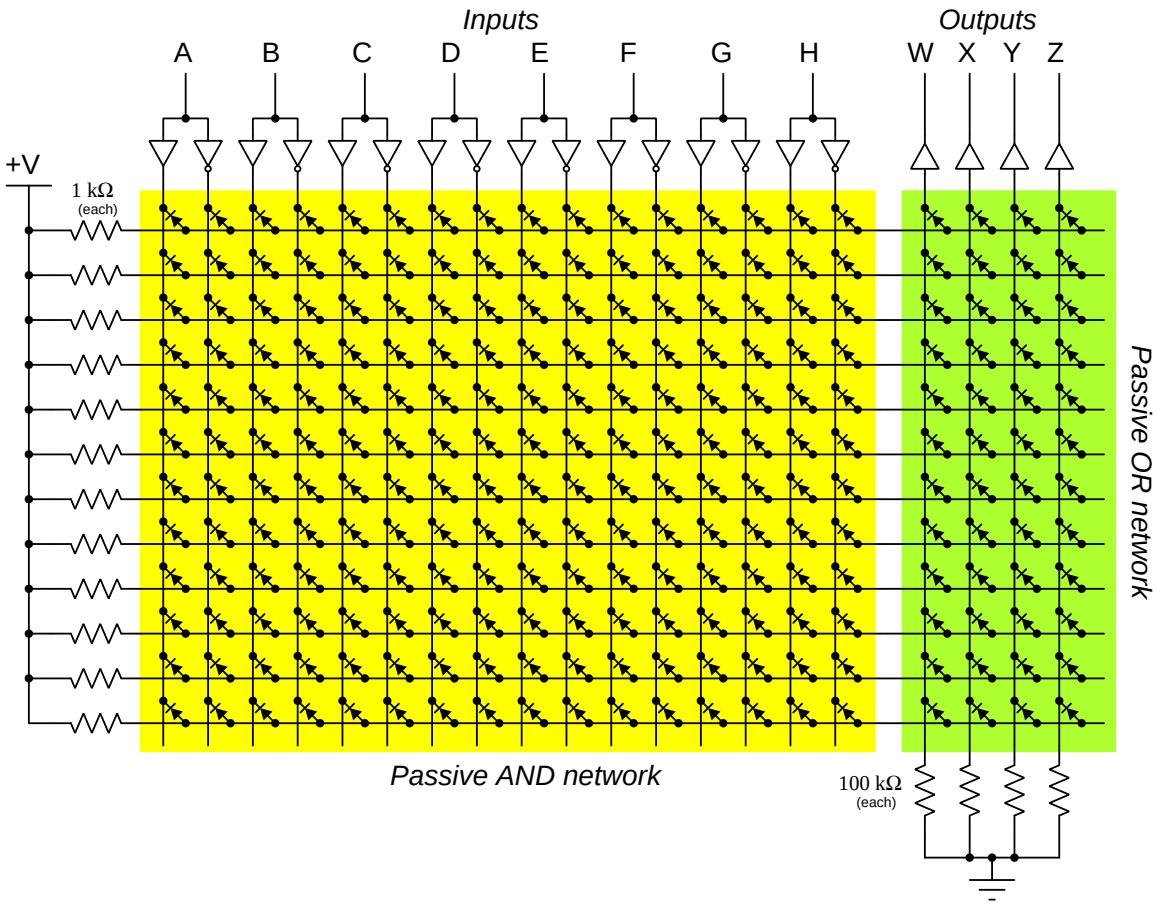


The passive AND gate outputs a "high" signal only if none of the inputs are "low". If one or more inputs go to the "low" (grounded) state, the output will assume a potential just one forward-voltage drop removed from zero. For example, operating on a 5 Volt DC source voltage, a "high" logic level will be +5 Volts to ground while a "low" logic level will be approximately 0.7 Volts to ground if standard silicon PN junction diodes are used.

Similarly, the passive OR gate outputs a "high" signal if one or more of its inputs are "high", and in that state the high-level output will be 0.7 Volts less than +V. The only way for the OR network to output a "low" signal is if every one of its inputs is "low", in which case the low-level output will be zero Volts to ground.

Adding digital buffer gates to the input and output lines makes both of these gate circuits practical in the sense that their inputs will not electrically load down any gates driving them and they will be better able to source and sink current to any loads connected to their outputs.

Shown here is an eight-input, four-output programmable logic array using passive diode-resistor networks. In the form shown the circuit is unprogrammed, and needs to have diodes removed (i.e. blown open as fuses) in order to leave only the intended connections in the AND and OR networks:



The 1 kiloOhm pullup resistors provide a default "high" state to each of the horizontal lines unless and until one or more of the input buffers drive it "low" by sinking current through an intact diode. The 100 kiloOhm pulldown resistors provide a default "low" state on each of the vertical output lines unless and until one or more of the horizontal lines drive it "high" by sourcing current through an intact diode. Vastly disparate resistance values were chosen for these pullup and pulldown resistors in order to ensure a minimal voltage-divider effect when one of the pullup resistors needs to assert a "high" state on one or more of the output lines connected to pulldown resistors. Even with the 1 kΩ and 100 kΩ values specified in this design, the typical "high" state output voltage would be 4.25 Volts assuming a +5 Volt DC source and standard 0.7 Volt forward-drop silicon diodes.

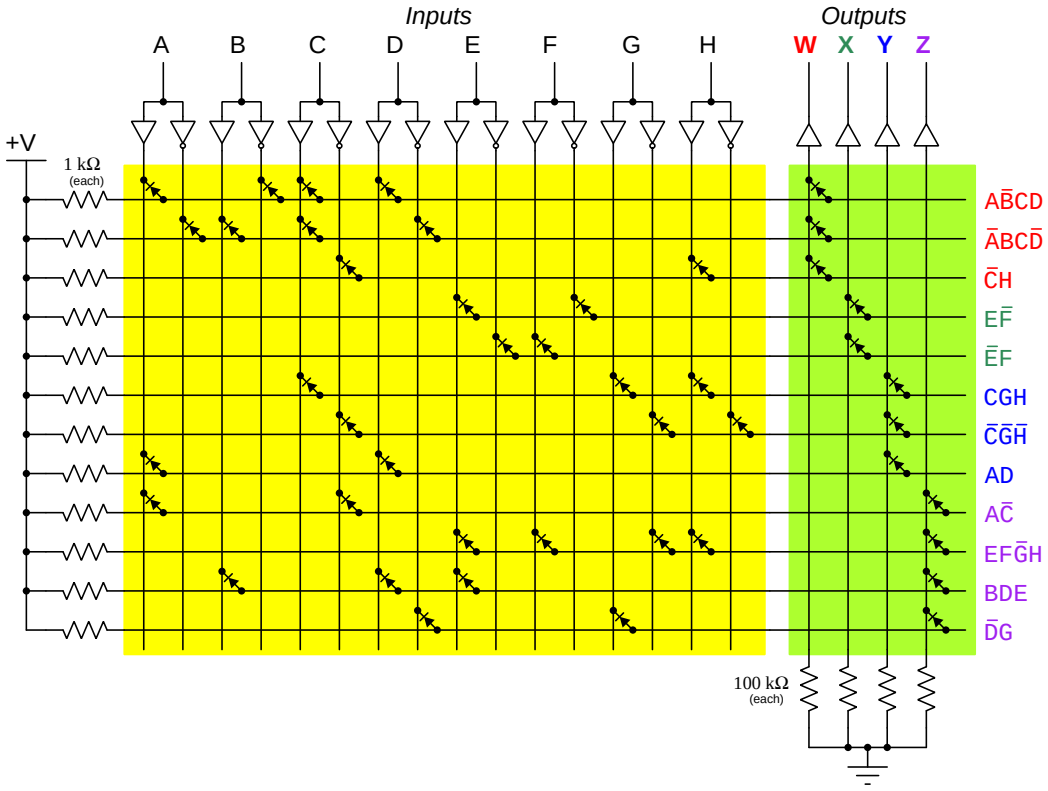
As an example of this circuit's use, consider the diode connections necessary to fulfill the following Boolean SOP expressions:

$$W = A\overline{B}CD + \overline{A}BC\overline{D} + \overline{C}H$$

$$X = E\overline{F} + \overline{E}F$$

$$Y = CGH + \overline{C}\,\overline{G}\,\overline{H} + AD$$

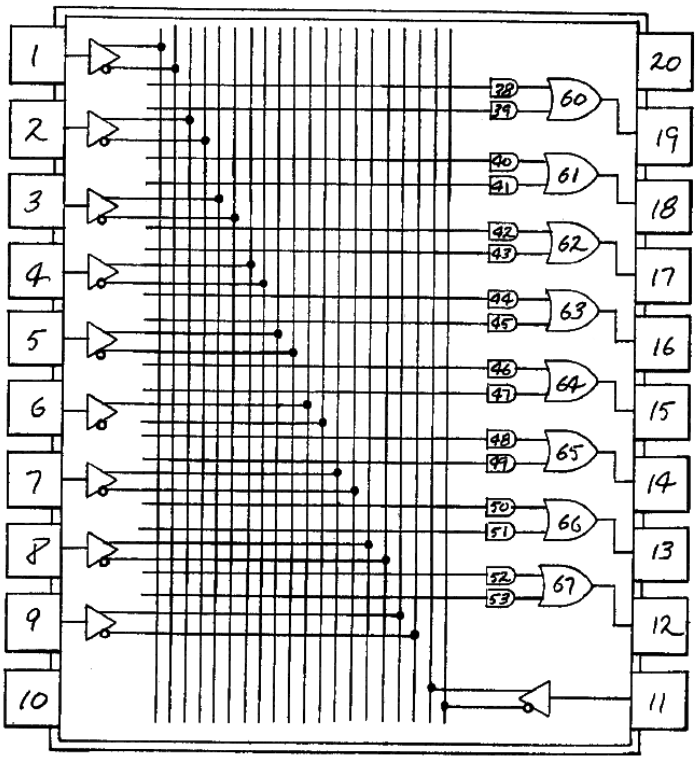
$$Z = A\overline{C} + EF\overline{G}H + BDE + \overline{D}G$$



Having twelve horizontal lines means this logic array has a maximum total of twelve Boolean products for all of its output functions, and in this particular example we used all twelve of them: three for the W function, two for the X function, three for the Y function, and four for the Z function.

This particular form of programmable logic, where both the AND functions as well as the OR functions are programmable, is generally known as a **Programmable Logic Array** or **PLA**.

One of the disadvantages of utilizing both configurable AND arrays and configurable OR (or NOR) arrays to form PLA devices was the relatively large size of the silicon die necessary to implement all the fuse or anti-fuse links connecting input lines to AND gates and AND gate outputs to OR gate inputs, respectively. Engineers working at Monolithic Memories Incorporated designed and patented a simpler form of programmable logic IC in the late 1970's using permanently-connected OR gates and programmable AND gates to reduce the amount of die area needed for fuse/anti-fuse links. The reduction in space more than made up for the reduced flexibility of the logic because it allowed a greater number of gates to occupy the silicon die. A schematic from their patent (US Patent number 4,124,899) shows one possible realization of the concept:



This schematic omits all the links connecting input lines to the AND gates for simplicity.

It is common to refer to this form of programmable logic IC as a **Programmable Array Logic** device or **PAL**, in order to differentiate it from the PLA with its programmable OR as well as programmable AND networks.

The inventors of the PAL also cited the possibility of connecting flip-flops to the OR/NOR gate outputs to allow for synchronous control of data flow through the device, as well as controlled feedback of output signals back to AND gate inputs to permit the construction of *state-based* logic such as finite state machines (FSM). The general concept is shown in this partial schematic diagram showing one of the device's OR gates passing data through a D-type flip-flop, the output of which becomes another set of complemented input signals accessible to any of the AND gates in the device:

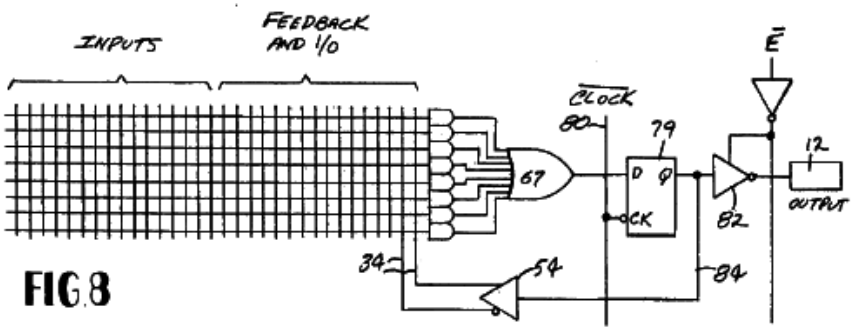
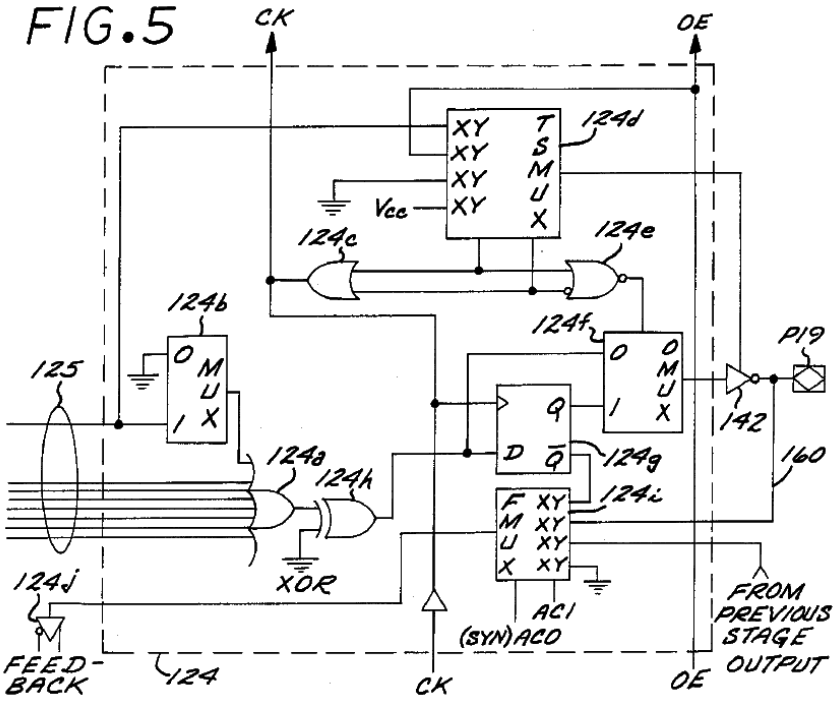


FIG.8

In this diagram, the D-type flip-flop's (79) Q output is accessible at one of the IC's output pins (12) whenever the tri-state buffer (82) is enabled by the $\overline{E}$ line, but is always available as an additional set of potential AND gate inputs (34). Moreover, *every* OR gate is potentially equipped with a flip-flop to recycle its output back to the AND array's input lines (specifically, the group of vertical lines labeled *feedback and I/O*) so that each SOP expression is able to incorporate the output state of other SOP expressions in the device.

This *signal feedback* makes possible all kinds of interesting state-based logic functionality such as sequencers, counters, etc. For example, imagine one of these programmable devices being configured such that one product term on each of the OR gates contained variables driven by the preceding OR gate's registered output. Programmed correctly, it would mean each OR gate could generate a "high" output only when the preceding gate's output went high *and* other input conditions specified in the product term were met, thus creating a set of discrete states advancing one to the next only when the right conditions were met.

In the mid-1980's engineers at Lattice Semiconductor Corporation designed and patented yet another variation on the theme of programmable AND and OR networks introducing two innovations: the use of floating-gate MOSFETs as programmable links for the AND gate input array as well as a standardized logical module called an *Output Logic Macro Cell* (OLMC) following each of the OR gate outputs with multiplexers for signal routing and flip-flops for "registered" (i.e. synchronous) gating of signals through the device. The use of floating-gate MOSFETs made possible non-volatile re-programming of the device, rather than relying on fuses or anti-fuses which could only be programmed once. A simplified diagram for each OLMC appears below:



This one OLMC contains a single OR gate constituting the sum of an SOP expression. Each of the product terms for this SOP expression comes from AND gates outside of the OLMC, those AND gate output lines represented by the label 125 in the diagram.

Each of the multiplexers ("MUX") as well as the Exclusive-OR (XOR) gate are controlled by digital bits contained in a register called the *architecture control word*. The multiplexer selection bits act to select specified signal sources for the OLMC logic. For example, MUX 124i in this diagram is controlled by architecture control bits AC0 and AC1, causing the feedback signal (124j) to either be the flip-flop's (124g) $\overline{Q}$ output, the I/O pin's state (P19), the output of the previous OLMC, or constantly "low" (ground potential) as determined by the end-user. The XOR gate (124h) takes the OR gate's (124a) output signal and either passes it through unchanged (default) or inverts it to emulate a NOR gate (if the other XOR input is made "high" rather than "low").

This basic design of programmable logic IC became known as the **Generic Array Logic** or **GAL** and remains popular to this day (2023).

Modern variations on the theme of AND-OR programmable logic include the so-called **Complex Programmable Logic Device** or **CPLD** which is nothing more than a multitude of PLA- or PAL-style AND-OR gate networks and macrocell modules associated with one or more busses providing flexible interconnection options between I/O pins and these internal structures. For example, a typical CPLD may have a *global bus* consisting of lines connected to the external I/O pins and to macrocell flip-flop outputs as well as separate *regional busses* acting as feedback networks for specific clusters of macrocells. In a typical CPLD every input pin is available to every macrocell, but each macrocell has its own dedicated output pin.

Modern CPLDs are re-programmable, and their memory elements are typically non-volatile. Once a CPLD has been programmed, it is ready to operate immediately upon every power-up, behaving identically to an equivalent set of gates and flip-flops hard-wired together. Being re-programmable, though, means that "wiring" may be easily changed if desired, making CPLDs excellent for prototype design work as well as end-use applications amenable to revision with firmware updates.

2.7 Field-Programmable Gate Arrays

In the mid-1980's a new company named Xilinx was launched with its flagship product being a new type of programmable logic IC that came to be known as a **Field-Programmable Gate Array** or **FPGA**. In contrast to prior programmable logic ICs using vast arrays of AND and OR gates to implement Boolean SOP expressions, the FPGA utilized simpler logic elements consisting of just a few gates and flip-flops with a much more complex "fabric" of interconnecting bus lines to support the creation of complex logic functions. Static RAM memory elements programmed with *look-up tables* replaced AND-OR gate networks with fusible or programmable interconnects to form arbitrary combinational logic functions.

An important distinction between FPGAs and prior programmable logic IC technologies is the extensive use of volatile memory elements in the former versus non-volatile storage in the latter (whether fuses or antifuses in one-time programmable PLAs and PALs or floating-gate MOSFETs in later devices). This difference means CPLDs and similar devices are ready to operate immediately upon power-up, but FPGAs must first be *booted* (i.e. initialized with configuration data) by either on-board flash memory storage or from some external device with its own non-volatile store of configuration data. However, the advantage of using volatile memory within the FPGA means those memory elements may be made much smaller – and therefore any given size of silicon die may contain more of them – than comparable technology based on non-volatile memory elements.

An FPGA contains a multitude[5] of configurable logic blocks (CLBs), sometimes called configurable logic elements (CLEs) or logic cells (LCs) or logic array blocks (LABs) rather than *macrocells* as customary for CPLDs. Sometimes the term "configurable" is dropped entirely since it is well-understood that these logic blocks must be configurable to be useful within an FPGA, shortening acronyms such as CLE and CLB to *LE* and *LB*, respectively. Just to be confusing, some manufacturers refer to these logic blocks, or to specific sections within each block, as *slices*. In any case, FPGA logic blocks typically contain at least one look-up table (LUT), one flip-flop, one binary adder, and multiplexers or "pass" transistors necessary to steer signals in particular directions between these basic elements. As FPGA technology evolves, we find more specialized function blocks included on the same silicon die, including dedicated network interface circuits, random-access memory arrays, specialized digital math units such as multipliers or dividers, and digital signal processing (DSP) circuits such as digital filters that would otherwise require chaining together large numbers of logic blocks that could be used more efficiently for other tasks.

No standard presently exists for the internal workings of FPGA logic blocks, each manufacturer offering their own unique configurations and capabilities.

[5] At the time of this writing (2023), one may purchase FPGAs with *over half a million* of these logic blocks on a single IC!

An illustration from the first patent (United States patent number 4,870,302 by Ross Freeman of Xilinx Incorporated) for an FPGA shows the basic concept of surrounding a multitude of relatively simple logic blocks (logic elements, or L.E.'s as referenced by Freeman's patent) with a "fabric" of interconnecting bus lines to form a useful *array* of digital logic suitable for implementing a wide range of functions:

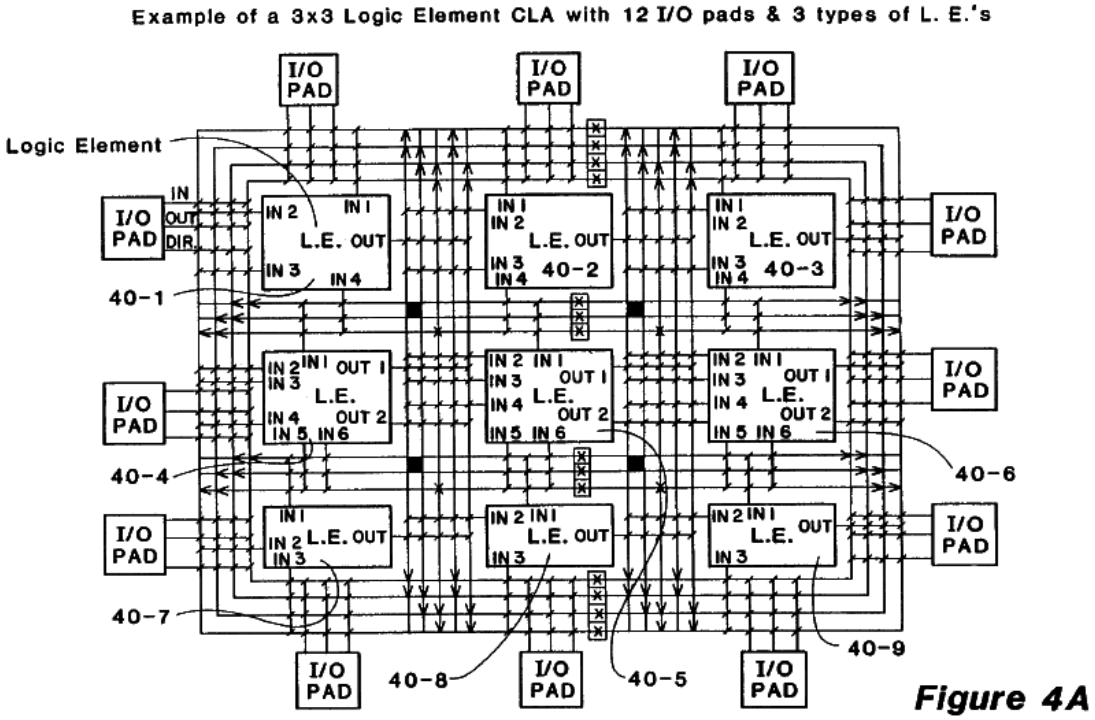


Figure 4A

Note the interesting symbols Freeman used to show "interchanges" between some of the bus lines, many of which had the capability not only to join certain lines together but also to re-route or even sever some of them if desired.

2.8 Hardware description languages

Simple programmable ICs based on configurable AND-OR gate networks are possible to directly program at the level of gate interconnections, determining which connections to make (or break) in order to form the desired Boolean expressions, as the device structures are relatively simple. In a similar manner, simple microprocessor systems are programmable at the "machine code level" of binary 1's and 0's by anyone with access to the IC's instruction set, opcode list, and a map of available registers. However, programming either a programmable logic IC or a microprocessor at such a low level of abstraction quickly becomes impractically confusing and error-prone for mere mortals. For extremely dense devices such as modern CPLDs and FPGAs the task of "bare metal" programming becomes flatly impossible for any human being to manage, just as it is impossibly complex for any human to write a program with modern features in machine code for any processor IC.

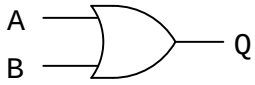
For microprocessor programming the solution to this problem is the use of a "higher-level" programming language such as C, C++, or Python which is more easily understood by human beings, that becomes translated into the microprocessor's native "machine code" language by a piece of software (running on a different computer) called a *compiler* or an *interpreter*. Compilers and interpreters make the task of writing effective programs simpler by allowing the human programmer to focus on "big-picture" concepts and tasks instead of getting bogged down by attention to trivial details. Compilers and interpreters also make it possible for a program written in one language to execute on different microprocessor hardware, that compiler/interpreter software translating the high-level instructions into each microprocessor's unique and specific machine-code dialect.

With programmable logic ICs such as CPLDs and FPGAs the problem and the solution is much the same: let some piece of software determine all the detailed interconnections necessary to configure the IC to do the desired task, while letting the human designer focus on high-level concerns. In order to do this, there needs to be some intermediate form of language through which the human designer can express to that software what it is they intend the IC to do, and then that software must be aware of the IC's specific capabilities and architecture in order to accurately translate the designer's wishes into a set of physical interconnections that will yield the intended result(s). In the world of programmable logic ICs, this is generically known as a *Hardware Description Language* or *HDL*.

Procedural programming languages such as C, C++, or Python fundamentally describe a series of steps for a computer to follow (i.e an *algorithm*), but for many digital systems it makes more sense to describe the network as a concurrent set of inputs, outputs, and logical functions. Concurrent code may appear strange[6] to anyone familiar with procedural (algorithmic) code because concurrent code describes multiple functions happening simultaneously, while procedural code describes one action happening at a time. One of the complexities of hardware description languages is that they may support programming at multiple levels of abstraction: for example, it may be possible to describe a digital network in algorithmic terms or in concurrent terms, all using the same language!

[6] Readers familiar with this collection of instructional modules have no doubt seen examples of *SPICE netlists* used to simulate electrical networks. This is another example of a concurrent programming language, where the code describes the configuration of a system where many things happen simultaneously.

Modern HDL programming usually focuses on one of two competing languages: *VHDL* or *Verilog*. Here we will explore both of these hardware description languages as they might apply to a trivial case of modeling a single OR gate at the *Register Transfer Level* (RTL) of abstraction:



VHDL code example for a single OR gate (RTL abstraction)

```
library IEEE;
use IEEE.STD_Logic_1164.all;
entity MyGate is
  port(A, B: in std_logic;
       Q: out std_logic);
end;

architecture implement of MyGate is
begin
  Q <= A or B;
end;
```

Verilog code example for a single OR gate (RTL abstraction)

```
module MyGate(A, B, Q);
input A, B;
output Q;

assign Q = A | B;
endmodule
```

As we can see from these two examples, VHDL defines the inputs and outputs of a digital function as an *entity* and the input/output behavior of that function as an *architecture*. Verilog lumps the two together into a single *module*. Other subtle differences distinguish these two languages from each other as well.

HDL languages such as VHDL and Verilog support hardware description at multiple levels of abstraction. The examples shown above are both at the Register Transfer Level, but it's also possible to use either language to describe the intended digital network's behavior in more procedural terms (where is it easiest for the computer to simulate) as well as in lower-level terms where actual wire connections between elements are specified. HDL code specifying signal or wire connections between

individual logic elements is often referred to as *structural* code, as contrasted against *behavioral* code such as RTL and higher abstraction levels where the HDL merely states the intended output conditions for given inputs.

The process of converting behavioral HDL code into bit-states to instruct a CPLD or FPGA which interconnections to make requires multiple steps, analogous to the *compiling* and *linking* steps required to convert a procedural programming language such as C into something a microprocessor may directly execute:

- **Synthesis** – here the RTL code gets converted into a *netlist* describing which logical elements must connect to each other to form a functional network
- **Mapping, placing, routing** – here the netlist is mapped to actual elements and signal bus routes available on the particular target CPLD/FPGA device, usually done with software from the IC vendor
- **Bitfile generation** – here the software writes a device-specific binary file with all the configuration date necessary to select the necessary logical elements as well as designate specific signal routes to take between those elements on the target CPLD/FPGA device

A powerful feature of both the VHDL and Verilog hardware description languages is their ability to *simulate* encoded digital logic in order to verify the correctness of that code prior to writing it to a programmable IC. This is really an essential utility for any complex digital system design where so much can go wrong. In both VHDL and Verilog languages the portions of code used to apply simulated input conditions and monitor output states for correctness are called *testbenches*.

Chapter 3

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.

3.1 PAL patent

John Birkner and Hua-Thye Chua of Monolithic Memories Incorporated filed a United States patent (number 4,124,899) for a programmable array logic (PAL) IC in May of 1977, which gives a lucid description of how programmable AND/OR networks function. Their specific innovation, of a programmable array of AND gates feeding in to fixed rather than programmable OR or NOR arrays, is what we now generically refer to as a PAL type of programmable logic IC, but the lessons we may draw from this patent apply to a wide range of programmable logic technologies.

The patent's "Background of the invention" section serves well as a brief history of programmable logic technologies at that time:

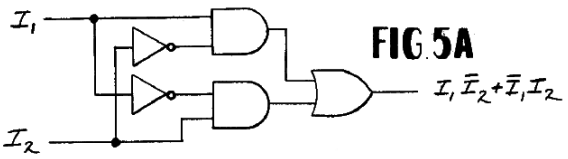
> Fusable links used in bipolar PROMS (Programmable Read-Only Memories) have given the digital systems designer the ability to "write on silicon." In little more than a few seconds, an algorithm, a process, or a boolean transfer function can be permanently provided in the regular structure of an integrated circuit (IC) read only memory.
>
> PROMs are useful for many purposes including microprogram stores for high speed processors and controllers, non-volatile program stores for minicomputers and microprocessors, and high speed character generation and look up tables.
>
> More recently, programmable integrated circuits have been extended to logic circuit arrays. These are sometimes referred to as PLAs (Programmable Logic Arrays) and FPLAs (Field Programmable Logic Arrays). FPLAs, in contrast to earlier mask-programmable circuits, can be programmed away from the place they are manufactured. Any problems in a programmed design that are discovered can be corrected simply by programming a new FPLA and discarding the old one. If the particular application has high enough volumes to cost justify it, a mask can be designed subsequently so that mask-programmable arrays can be made.
>
> PLAs are used in the implementation of random logic networks, data routing, code converters, instruction decoders, state sequences, and a variety of other functions. For a general discussion of PLAs and FPLAs, reference is made to Electronic Design, Vol. 18, Sept. 1, 1976, "PLAs or μPs? At Times They Compete, and At Other Times They Cooperate", pp. 24-30.
>
> Existing FPLAs comprise an array of logical AND and OR gates which can be programmed for a specific function. Each output function is the sum (logical OR) of selected products (logical ANDs) where each product is the product of selected polarities of selected inputs.
>
> FPLAs can be programmed so that (1) any input line can be connected to any AND gate input and (2) any of the products (ANDs) can be summed by any of the OR gates. This is accomplished by providing a programmable array or matrix (1) between the circuit inputs and the AND gate inputs and (2) between the output of the AND gates and the inputs of the OR gates, respectively. The FPLA is then programmed by blowing or not blowing the fusible links connecting the conductors of the two arrays much the same way as PROMs are programmed. Examples of such FPLAs are Signetic Models 82S100 and 82S101.

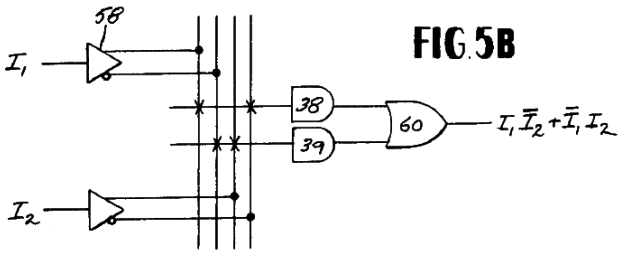
> Existing FPLAs as described above, while useful in many applications, have certain disadvantages. First, the size of the IC chip is quite large, due to the use of two programmable arrays per FPLA. This means lower yields, greater costs, and larger IC packages.
>
> Secondly, the flexibility of such FPLAs is limited. They are limited as to the number of inputs, speed, and perhaps most importantly, architecture. Existing FPLAs are very limited in terms of the logical and arithmetical operations they can perform. [page 15]

In their "Summary of the invention" section of the patent, the authors proceed to describe what sets their particular design apart from prior-art designs:

> In accordance with the present invention, an improved FPLA, hereinafter referred to as a programmable array logic (PAL), comprises a single programmable array or matrix of circuit inputs and the inputs to a plurality of AND gates (product terms). Outputs from subgroups of AND gates, in turn, are nonprogrammably connected as inputs to individual, specified OR gates (sum of the products).
>
> By making the AND gate inputs programmable, i.e. selectable by the designer, while having the OR gate inputs nonprogrammable, some design flexibility is sacrificed. However, the reduction in IC chip size for the PAL more than makes up for the slight reduction in flexibility. Smaller chip size means greater yields and hence lower costs. Smaller chip size also means that more convenient packaging can be used. For example, a package size of approximately 300 mils wide by wide by 1000 mils long with 20 pins is easily accomplished. This compares with the 600 mils by 1400 mils package size, and 28 pins, for existing FPLAs of comparable circuit components and function. [page 15]

Figures 5A and 5B show a simple combinational logic example of two AND gates feeding into a single OR gate to form a two-term sum-of-products (SOP) function. Figure 5A shows this combinational circuit using standard gate notation while figure 5B shows it as implemented in a fuse-programmed PAL. "X" symbols shown in figure 5B designate locations where fuse links have been intentionally "blown" to disconnect certain inputs from certain AND gates:





Device 58 is a dual-output logical buffer providing both inverted and non-inverted versions of the input signal to a set of bus lines within the PAL device. Every input to the PAL IC passes through a buffer such as this, in order to provide both inverted and non-inverted versions of the input signal to the AND arrays as desired. Connections to those bus lines are by default, and may be severed by intentionally blowing fusible links between the buffer outputs and the respective bus lines. At each junction where a programming fuse has been blown, the disconnected input is designed to default to the logical "high" state so that its respective AND gate may still function on the states of the other (still-connected) inputs.

Devices 38 and 39 are both four-input AND gates, the four inputs shown as a single bus line for simplicity. As figure 5B shows, AND gate 38 has inputs connecting only to $\overline{I_1}$ and I_2 while AND gate 39 has inputs connecting only to I_1 and $\overline{I_2}$. Both AND gate outputs connect in a fixed (i.e. non-programmable) manner to the single OR gate labeled 60. This combinational function happens to emulate an Exclusive-OR (XOR) gate.

Any practical PAL device would of course contain AND arrays accepting far more than two programmable inputs. In figure 6B of the patent the inventors provide an example of an unprogrammed 20-pin IC supporting 10 inputs and 8 outputs:

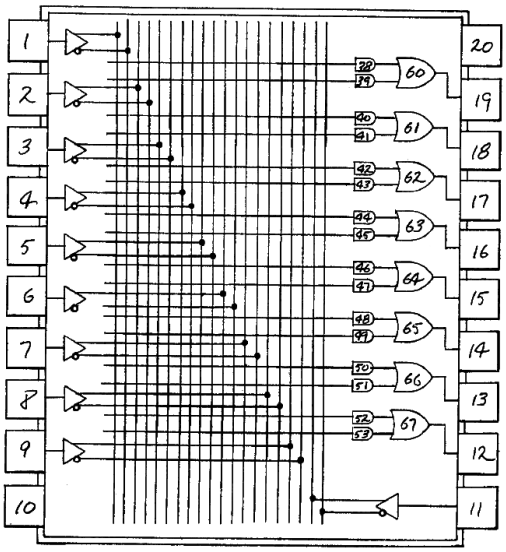


FIG 6B

A variant of this design using NOR gates rather than OR gates is shown in figure 6A, useful for implementing negative-sum-of-products expressions:

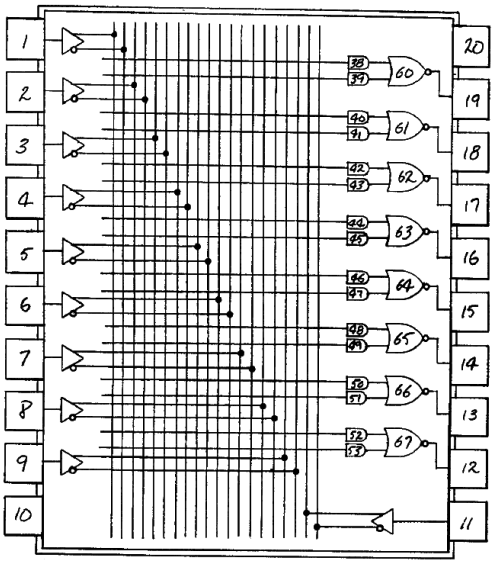
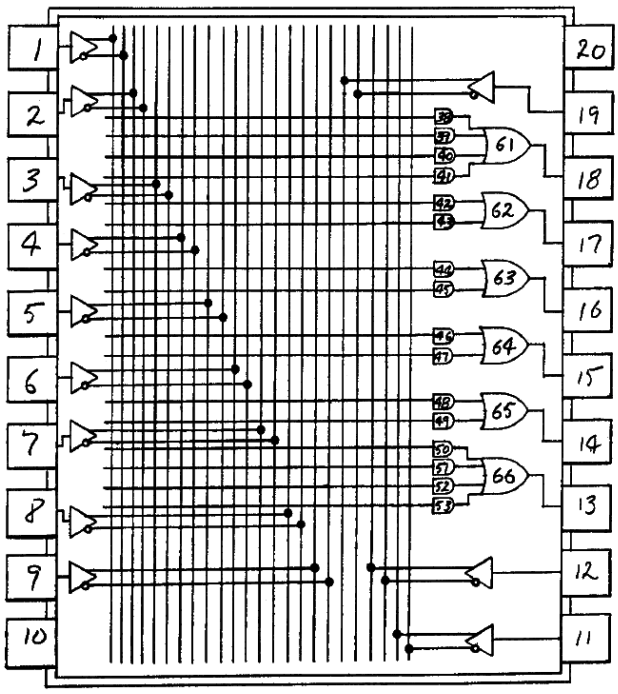


FIG 6A

To clarify, each of the PALs shown here are capable of implementing eight independent SOP or Negative-SOP expressions, each of those expressions having two terms and as many as ten inputs.

A variation on this theme mixes two-term and four-term SOP expressions on the same PAL:

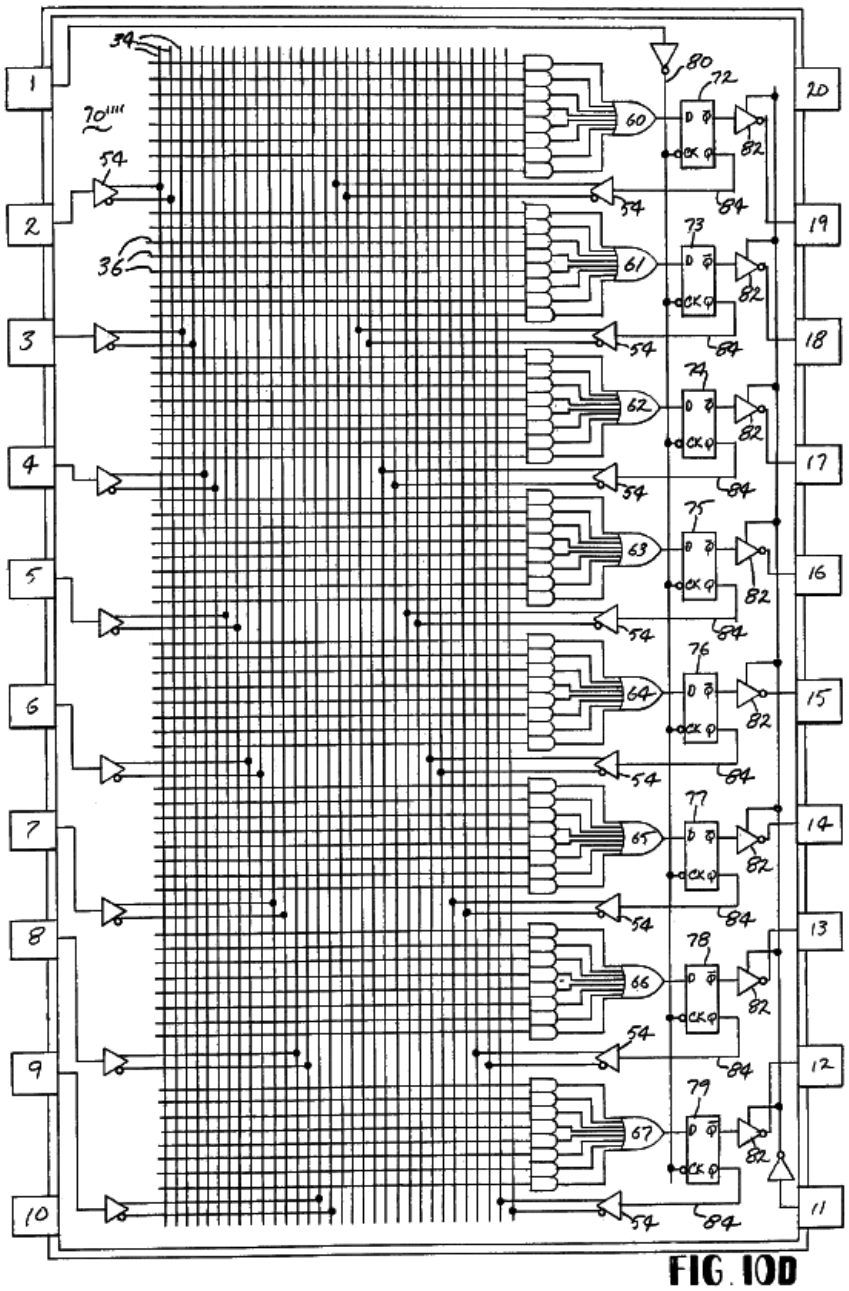


Here, OR gates 61 and 66 provide the four-term SOP functions, output on pins 18 and 13, respectively. The other outputs are all two-term SOP functions.

The inventors of this PAL extend functionality beyond combinational logic by adding latching logic to their device in the form of flip-flops which they call *registers*. Continuing the "Summary of the invention" section of the patent:

> In accordance with another aspect of the present invention, PALs are provided having greater design and operational flexibility than existing FPLAs. This is accomplished through improved architectural design.
>
> One improved architectural feature is the use of registered outputs with feedback. Registers are provided at OR gate outputs which allows temporary storage of the OR gate outputs. Additionally, a feedback path from each such register to the AND gate array is provided. This combination forms a state sequencer which can be programmed to execute elementary sequences such as count up, count down, shift, skip, and branch. [page 15]

One possible implementation of "registered" outputs within a PAL is shown in figure 10D:



In this version we see D-type flip-flops following every one of the eight-input OR gates, the $\overline{Q}$ outputs of those flip-flops passing through a tri-state buffer to the output pins, while the Q outputs

feed back to the AND array input bus where they may be enabled (i.e. fuses left intact) as state variables for state-based logic or disabled (i.e. fuses blown) to retain purely combinational logic functionality.

3.2 FPGA patent

Ross Freeman of Xilinx Incorporated filed a United States patent (number 4,870,302) for a programmable IC having both configurable logic elements and configurable interconnects between those logic elements, which later became known as a **Field-Programmable Gate Array** or **FPGA**. Like PALs the preceded this invention, the FPGA features re-programmable MOSFET elements allowing its configuration to be changed many times after manufacture.

At the heart of Freeman's invention was the *configurable logic element*, or simply *logic element* (also known as *configurable logic blocks* or CLBs in later FPGA designs), shown here as Figure 2 of his patent:

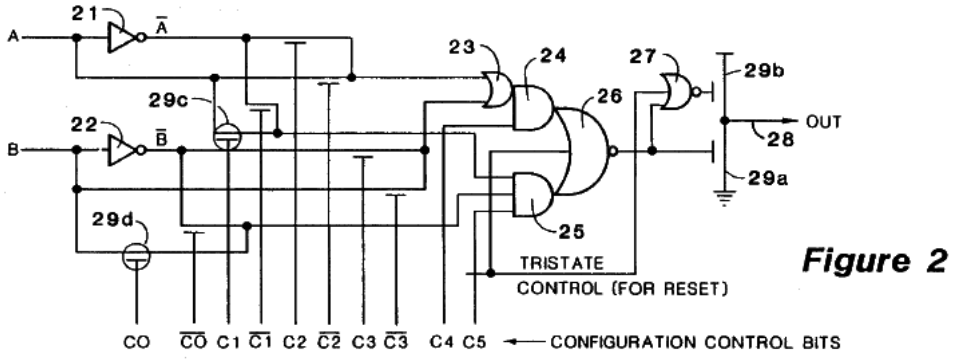


Figure 2

This illustration mixes standard logic-gate symbols with generic MOSFET symbols (e.g. devices 29a, 29b, 29c, and 29d) to show a network that may be configured with dedicated *configuration control bits* to perform various functions.

Freeman briefly outlines existing PLA and PAL technology in the "Prior Art" section of his patent description, then follows with a summary of how his invention differs:

> 2. Prior Art
>
> Gate arrays are well known in the prior art. Typically a gate array is produced by interconnecting a plurality of active devices in a base array in any one of a number of ways to achieve a desired logic function. As gate arrays become more complex, the simulation of the logic to be achieved from a given interconnection of the active devices in the base array becomes more difficult and is typically carried out using a computer program. The layout of the actual interconnections for the active devices in the base array to yield a finished gate array is then derived using a computer aided design program of a type well known in the art. The process of designing such a structure is complex and reasonably expensive requiring the use of logic simulation and verification programs and semiconductor device layout programs. Accordingly, a need exists for an alternative approach which significantly simplifies the obtaining of a given logic function from a base array.
>
> SUMMARY
>
> In accordance with this invention, I provide a structure which I denote as a configurable logic array which allows changing the configuration of the finished integrated circuit from

> time-to-time (even when the integrated circuit is installed in a system) to provide any one of a plurality of logical functions from the same integrated circuit. In accordance with my invention, by providing a number of "configurable logical elements" (also referred to herein as "logic elements") in the base array, a new type of integrated circuit is achieved which is capable of being configured to provide any one of a plurality of logic functions depending upon the tasks which the system of which it is a part is called upon to perform. By "configurable logical element" I mean a combination of devices which are capable of being electrically interconnected by switches operated in response to control bits to perform any one of a plurality of logical functions. [page 18]

Freeman does not limit the contents of the IC's configurable logic elements to just those shown in figure 2, however. Later in the patent he describes how figure 2 is merely one possible form of logic element and that other forms may be devised containing such devices as three-input gates, SR latches, D-type flip-flops, static RAM look-up tables, etc. Freeman offers a standard four-by-four row-and-column matrix of storage elements as a static RAM for implementing look-up tables (in Figure 3A, not shown here), and then proposes alternative to standard column and row decoders used in static RAM networks by using an array of MOSFETs useful for selecting one of sixteen pre-configured bits using a four-bit selection word (ABCD):

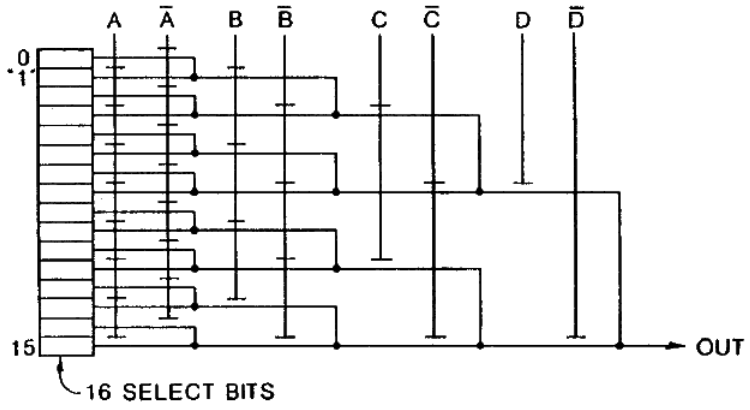


Figure 3B

In order to make these relatively simple logic elements truly useful, they must be interconnectable to form larger digital networks. In figure 4A Freeman illustrates an *array* of configurable logic elements surrounded by interconnection busses:

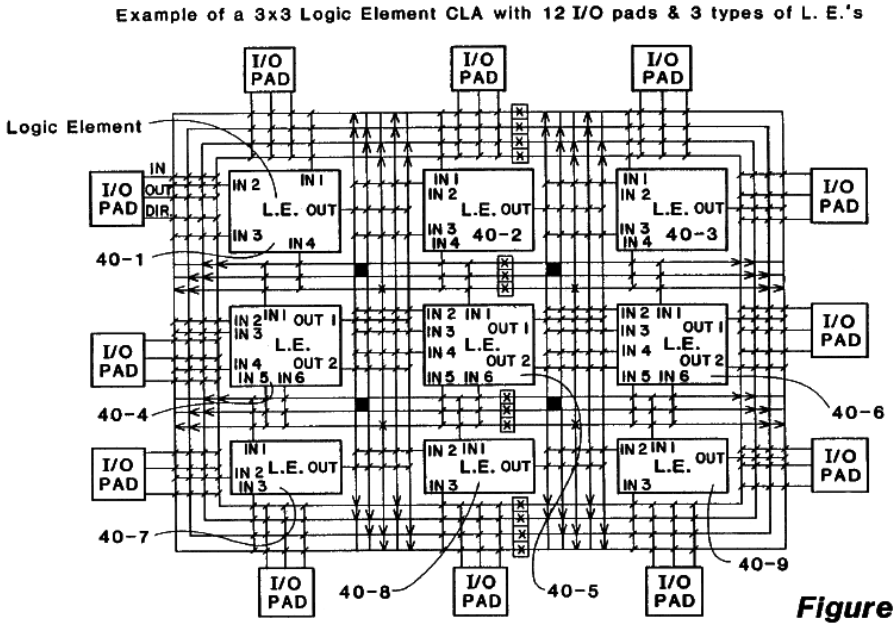


Figure 4A

Freeman describes how these interconnections function:

> FIGS. 4A illustrates an embodiment of a configurable logic array of this invention containing nine configurable logical elements. As shown in FIG. 4A, nine logical elements are placed on an integrated circuit chip together with interconnects and variable switches for connecting various leads to other leads. Each of logic elements 40-1 through 40-9 represents a collection of circuitry such as that shown in FIG. 2 or some similar structure capable of being configured as described above with respect to FIG. 2 to perform any one of a number of logic functions. To program the circuitry, selected signals of a logic element such as shown in FIG. 2 are applied to input leads of the configurable logic element identified as configuration control input leads from a source such as the RAM of FIG. 3A or 3B described above thereby to generate a desired logical function in each of the logic elements. In FIG. 4A, no specific I/O pad has been identified as an input lead for applying the configuration control signals to the logic elements. However, any particular I/O pad can be selected for this purpose. The configuration control bits can be input into the configurable logic array of FIG. 4A either in series or in parallel depending upon design considerations. Input of configuration control bits is described later in conjunction with FIGS. 5, 8A, and 8B. In addition, another I/O pad will be used on input clock signals to clock the logic elements both for the shifting in of the configuration control signals to each configurable logic element and for controlling the operation of each logic element during the functioning of the integrated circuit chip in its intended manner. The combination of logic elements 40-1 through 40-9 as configured

by the configuration control bits plus the interconnect structure of FIG. 4A yields the desired logical output for the Configurable Logic Array. FIG. 4B illustrates the meaning of the interconnect symbols used in FIG. 4A.

In order to express the complexity of interconnection options, Freeman provides legends to interpret some of his custom symbols shown in figure 4A:

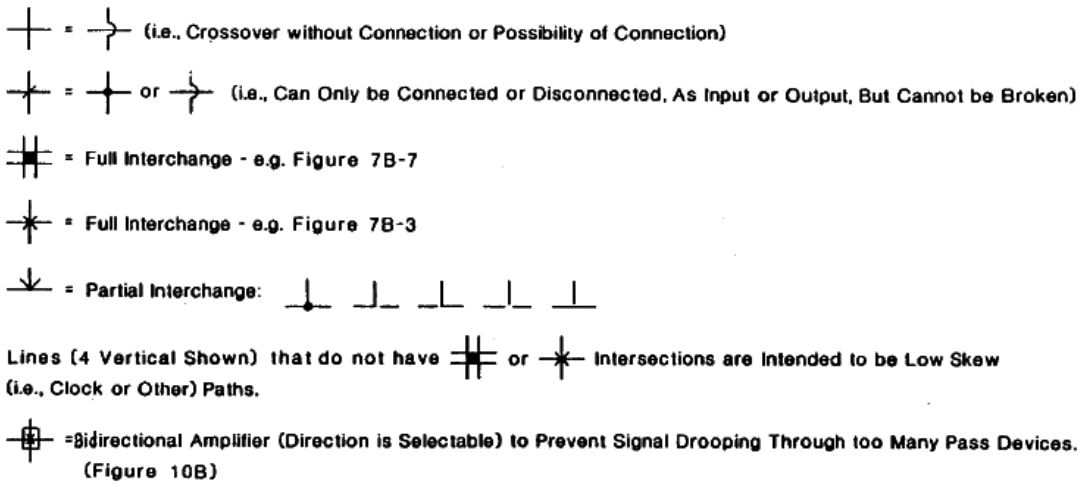


Figure 4B

As the legend shows, there are far more options than simply “connected” versus “not connected”, permitting connections to be made or broken between the busses and logic elements as well as between bus lines at intersections.

Some of these "interchanges" deserve further elaboration, which Freeman provides in figure 7B:

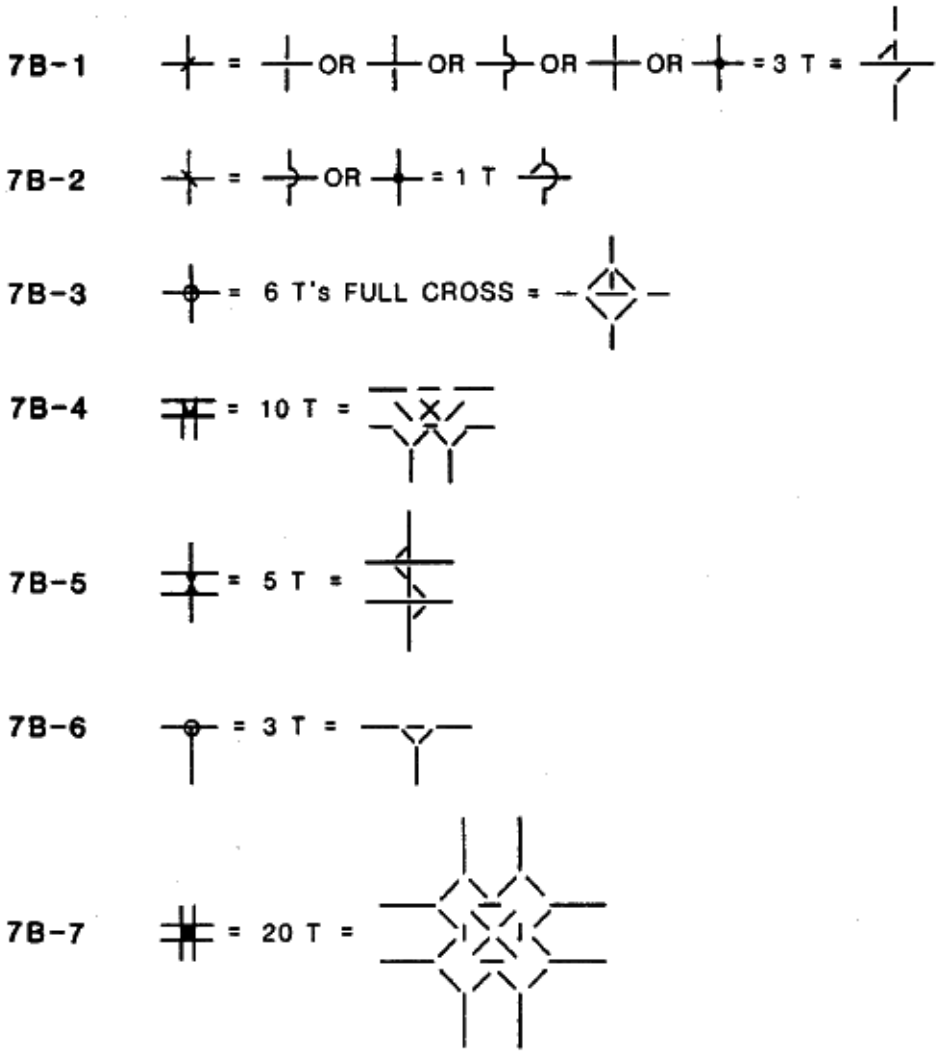


Figure 7B

For each interchange, Freeman cites the number of transistors necessary to implement the intended interconnection options, from one transistor (1 T) in figure 7B-2 to twenty transistors (20 T) in figure 7B-7.

These configurable interchanges must be implemented using actual transistors on the IC, and to describe how this may be done Freeman provides some schematics showing generic MOSFETs arranged in bridge-like configurations to make or break connections between bus lines. Take for example the following illustration showing six MOSFETs controlling one intersection between a vertical bus line and a horizontal bus line, symbolized by a circle around an intersection of bus lines shown (previously) in figure 7B-3:

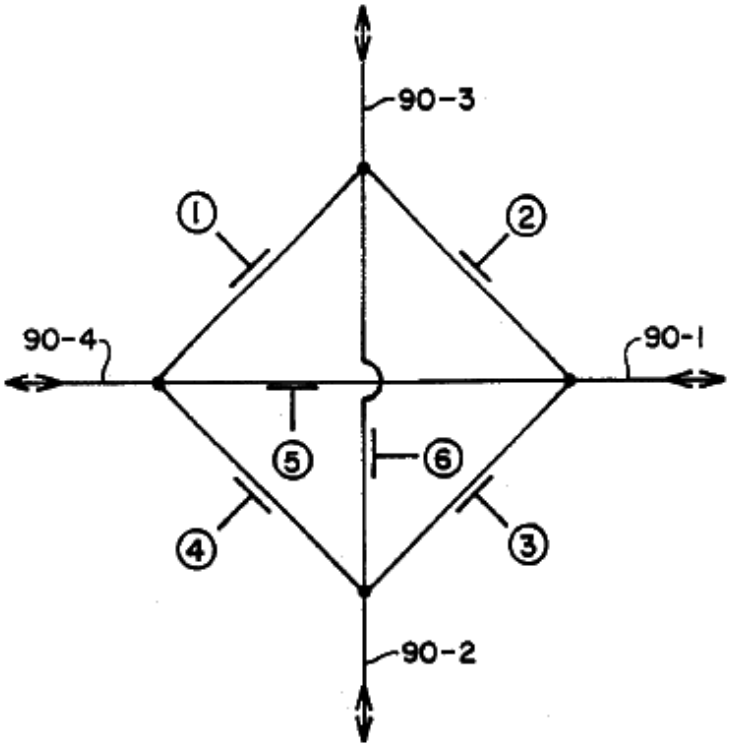


Figure 9A

For example, to make the vertical bus line complete and the horizontal bus line complete, but *not* form a connection between the two complete lines, one would activate "pass" transistors 5 and 6 while leaving all the other MOSFETs off. If, by contrast, one wished to make these two lines complete but also electrically common to each other, "pass" transistors 1, 2, 3, and 4 would need to be turned on while the states of transistors 5 and 6 could be either on or off.

For more complex interchanges involving two vertical and two horizontal bus lines, many more transistors are needed to provide a rich array of interconnection options. In the following figure Freeman shows such an interchange, using circled numbers to indicate which of the bus lines lie at either end of a "pass" transistor (the actual MOSFETs not shown for simplicity):

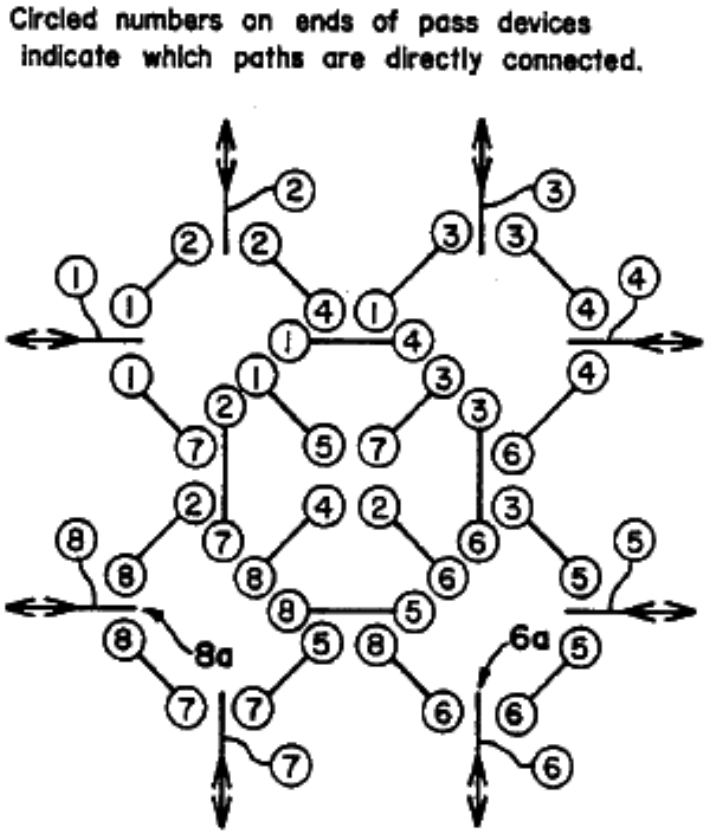


Figure 9D

Here there are eight bus lines terminating at the interchange, labeled 1 through 8. The twenty pass transistors positioned at various locations are shown simply as line-segments in this illustration, each of those transistors drain and source terminals terminating at two of the eight bus lines.

Just some of the interconnection patterns possible with such an interchange are shown in figures 9E, 9F, and 9G:

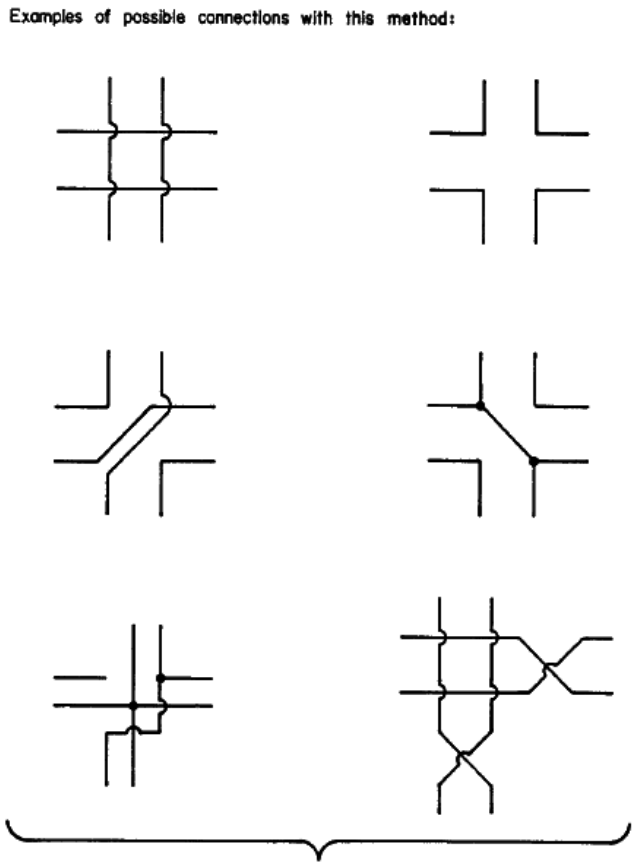


Figure 9E

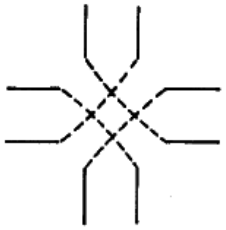


Figure 9F

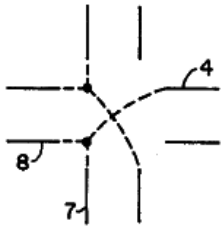


Figure 9G

Chapter 4

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?

4.1 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.

4.1.1 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, write their own outline and 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 OUTLINE THE TEXT, as though you were writing a detailed Table of Contents. Feel free to rearrange the order if it makes more sense that way. Prepare to articulate these points in detail and to answer questions from your classmates and instructor. Outlining is a good self-test of thorough reading because you cannot outline what you have not read or do 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.

4.1.2 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.

Truth table

Boolean algebra

Logic function

Logic state

AND function

OR function

NOT function

NAND function

NOR function

XOR function

Logic gate

Pullup/pulldown resistor

Passive AND gate

Passive OR gate

Multiplexing

Bus

Latch

Register

Flip-flop

Bistable

SR latch

D latch

JK flip-flop

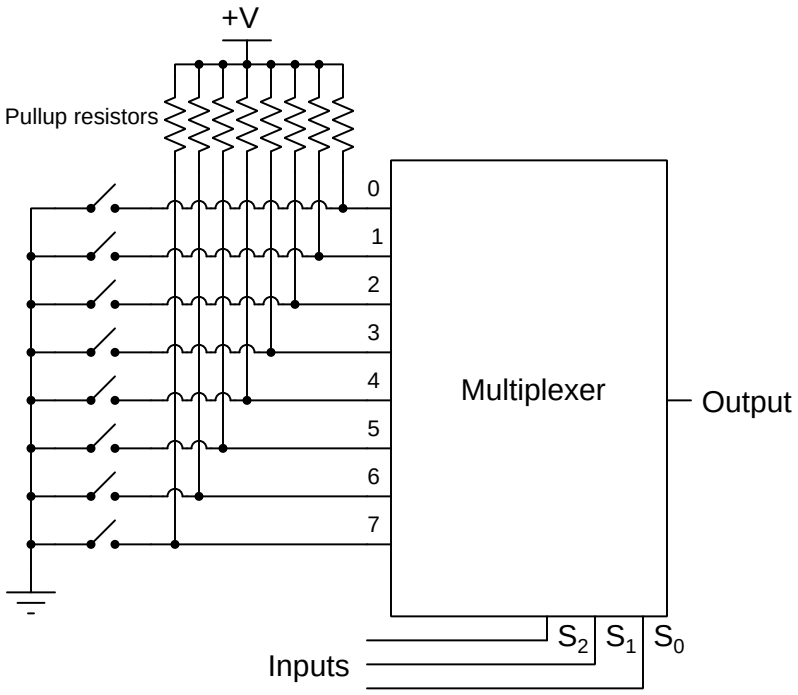
Source code

Thought experiments as a problem-solving strategy

4.1.3 Multiplexer-based logic functions

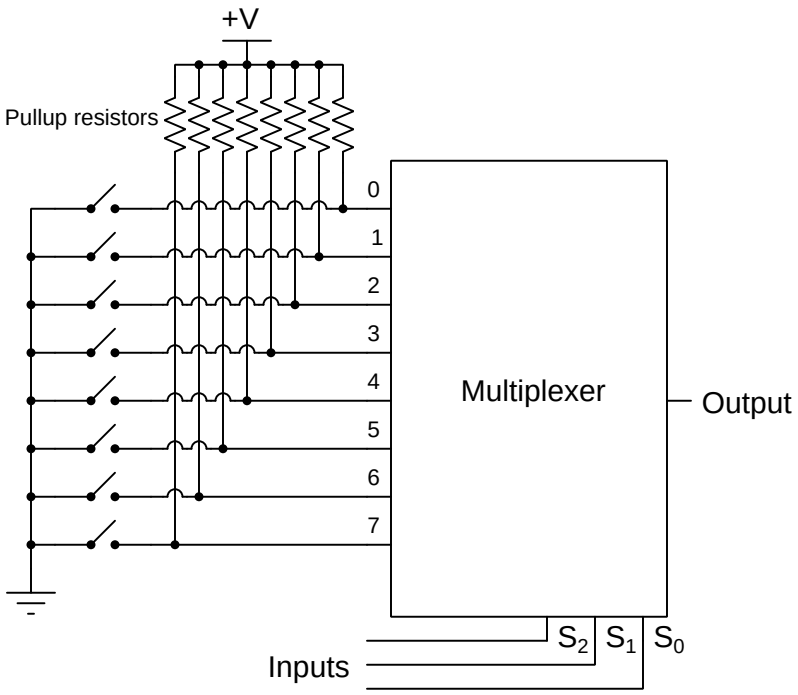
Example #1 – three-input NAND function

Determine the necessary toggle switch states to make this multiplexer implement a three-input NAND function:



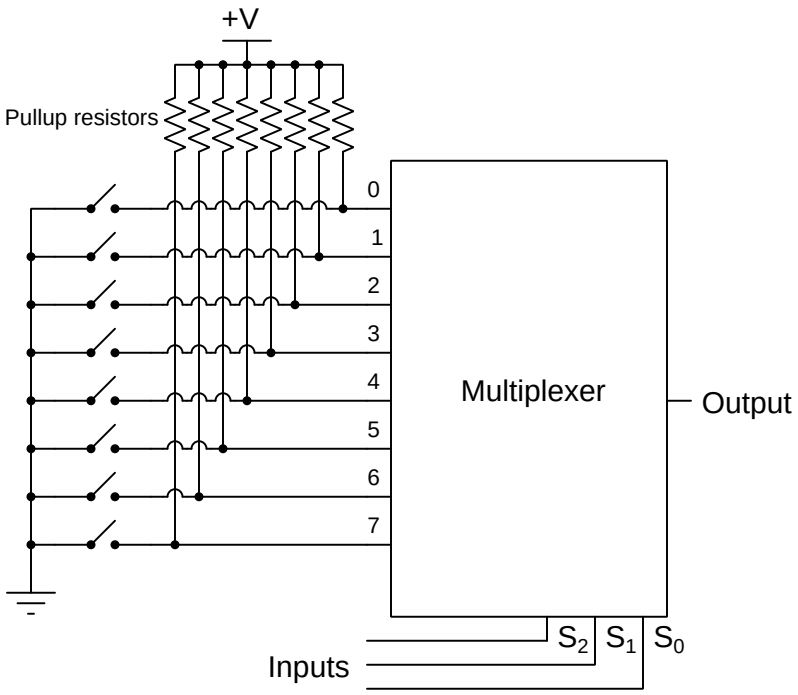
Example #2 – three-input NOR function

Determine the necessary toggle switch states to make this multiplexer implement a three-input NOR function:



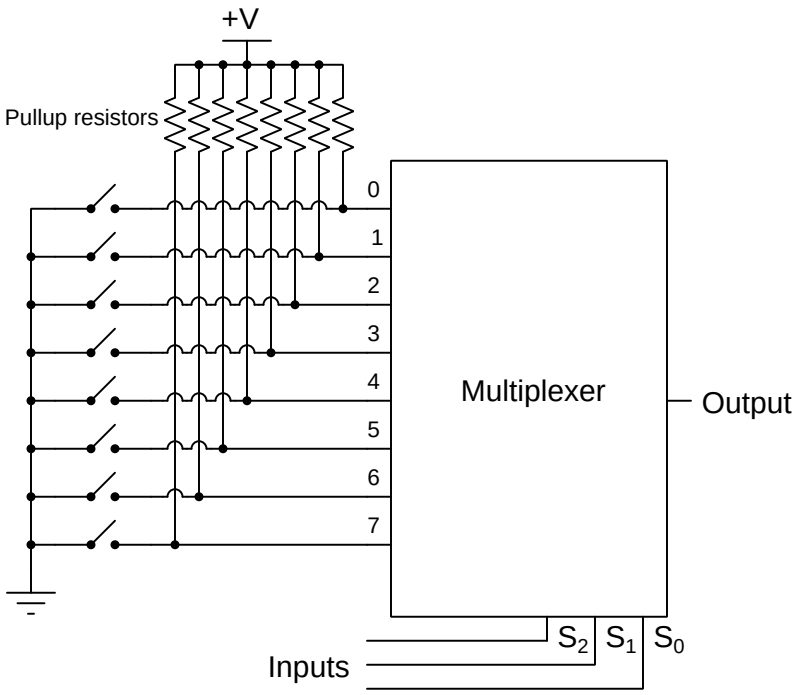
Example #3 – two-input OR function

Determine the necessary toggle switch states to make this multiplexer implement a two-input OR function where lines S_1 and S_0 are the two inputs and line S_2 is in an unknown state:



Example #4 – two-input AND function

Determine the necessary toggle switch states to make this multiplexer implement a two-input AND function where lines S_2 and S_1 are the two inputs and line S_0 is in an unknown state:

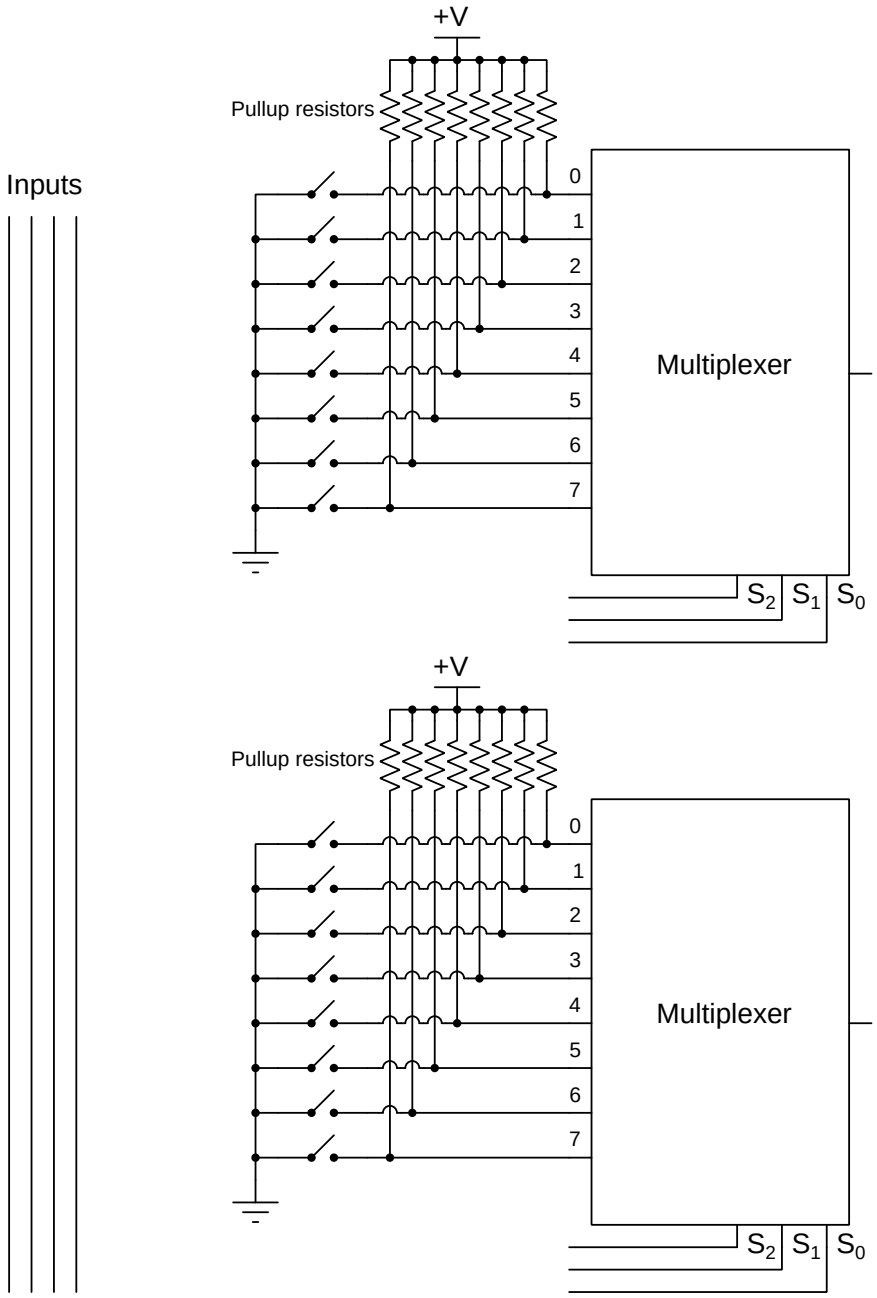


Challenges

- How might multiple 8-line multiplexers be combined to implement a four-input logic function?

4.1.4 Two multiplexers creating a logic function

Show how to connect these two 8-line multiplexers to form a programmable 4-input combinational logic function:



Challenges

- How is the creation of larger memory banks from multiple memory ICs similar to this challenge?

4.1.5 Explaining the meaning of code

Shown below is a source-code listing of a computer program written in the C language simulating a look-up table suitable for implementing any arbitrary truth table. Explain the purpose of each line of code:

Code listing:

```
#include <stdio.h>

int main (void)
{
  int lookup[8] = {0,0,0,0,1,0,1,0};
  int A, B, C, word;

  while(1)
  {
    word = 0;

    printf("Enter logic state for input A: ");
    scanf("%i", &A);

    printf("Enter logic state for input B: ");
    scanf("%i", &B);

    printf("Enter logic state for input C: ");
    scanf("%i", &C);

    if (A)
      word = word + 4;

    if (B)
      word = word + 2;

    if (C)
      word = word + 1;

    printf("Output state = %i\n\n", lookup[word]);
  }

  return 0;
}
```

Also, edit this program to make the look-up table implement a three-input OR function.

Challenges

- Which of the three input bits (`A`, `B`, `C`) are the least significant and most significant?
- How could this program be altered to implement a four-input look-up table?

4.2 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.

4.2.1 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.

4.2.2 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.*

4.2.3 First quantitative problem

Challenges

- ???.
- ???.
- ???.

4.2.4 Second quantitative problem

Challenges

- ???.
- ???.
- ???.

4.2.5 ??? simulation program

Write a text-based computer program (e.g. C, C++, Python) to calculate ???

Challenges

- ???.
- ???.
- ???.

4.3 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.

4.3.1 First diagnostic scenario

Challenges

- ???.
- ???.
- ???.

4.3.2 Second diagnostic scenario

Challenges

- ???.
- ???.
- ???.

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

"A Brief Introduction to VHDL and Verilog Hardware Description Languages (HDLs)", RealDigital.com, accessed online 4 January 2024.

Ashenden, Peter J., *The VHDL Cookbook*, First Edition, University of Adelaide department of computer science, South Australia, 1990.

Ashenden, Peter J., *VHDL Tutorial*, Elsevier Science, 2004.

Birkner, John M. and Chua, Hua-Thye, *US Patent 4,124,899*, "Programmable Array Logic Circuit", application 23 May 1977, patent granted 7 November 1978.

Brown, Stephen and Rose, Jonathan, "Architecture of FPGAs and CPLDs: A Tutorial", University of Toronto.

Cyliax, Ingo, "The FPGA Tour", Circuit Cellar magazine, November 1999.

Freeman, Ross H., *US Patent 4,870,302*, "Configurable Electrical Circuit Having Configurable Logic Elements and Configurable Interconnects", application 19 February 1988, patent granted 26 September 1989.

Han, Jin-Woo; Moon, Dong-Il; and Meyyappan, M., "One Time Programmable Antifuse Memory Based on Bulk Junctionless Transistor", Universities Space Research Association, Columbia, Maryland, 19 June 2018.

IEEE Standard for Verilog Hardware Description Language, IEEE Std 1364-2005, IEEE Computer Society, sponsored by the Design Automation Standards Committee, Institute of Electrical and Electronic Engineers Incorporated, New York, NY, 7 April 2006.

IEEE Standard for VHDL Language Reference Manual, Design Automation Standards Committee, Institute of Electrical and Electronic Engineers Incorporated, New York, NY, 5 September 2019.

Smith, Douglas J., "VHDL & Verilog Compared & Contrasted – Plus Modeled Example Written in

VHDL, Verilog and C", VeriBest Incorporated, Huntsville, AL.

Turner, John E. and Rutledge, David L., *US Patent 4,761,768*, "Programmable Logic Device", application 4 March 1985, patent granted 2 August 1988.

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.

30 January 2024 – corrected some typographical errors, courtesy of Joe Archer and Daniel Wing. Also added an introductory section to the Tutorial contrasting programmable logic ICs against other programmable technologies such as microprocessors.

27 December 2023 through 4 January 2024 – added more content to the Tutorial chapter regarding hardware description languages.

23-26 November 2023 – added more content to the Tutorial and Introduction chapters.

25 October 2023 – document first created.

Index