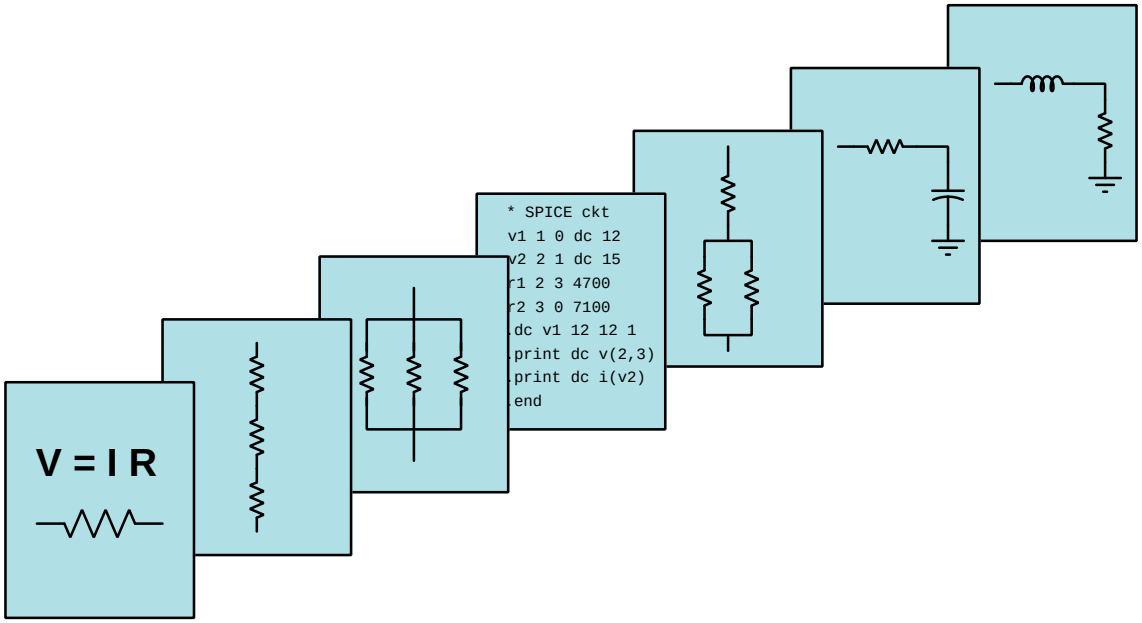
Modular Electronics Learning (ModEL) project



Digital Computing Circuits

Last update = 20 August 2024

Contents

Chapter 1

Introduction

1.1 Recommendations for students

The processing of digital words lies at the heart of digital computing. This processing takes two major forms: *arithmetic* and *bitwise logical*. Arithmetic operations are exactly what you would expect, including addition, subtraction, multiplication, division, etc. Bitwise logical operations are standard "gate" functions applied to all of the respective bits of two equal-length words. Both sets of operations are essential to digital computing.

Both arithmetic and bitwise logical operations are the foundation of general-purpose as well as application-specific computers. The Arithmetic-Logic Unit (ALU) section of a microprocessor is devoted to these tasks, implementing the operations in hardware form, those functions called upon by instructions (software) stored in the computer's program memory. Although you may never need to analyze or troubleshoot an ALU, it is nevertheless important to understand what these operations represent in order to comprehend digital computers.

Important concepts for the understanding of this topic include binary **counting** and number **range**, word **masking**, **operands** versus **opcodes**, status **flag** bits, **"don't care"** states, **inverting** (complementing) boolean states, **signed** integer numbers, **multiplexing**, **functional composition**, elementary **logic functions** (AND, OR, XOR, NOT), and basic digital **programming**.

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 binary addition overflow? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- How might an experiment be designed and conducted to demonstrate the principle of bitwise rotation using a shift register? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What are some of the useful functions offered by an ALU?
- What conditions generate a "carry" state in binary arithmetic?

- How do "bitwise" operations relate to basic logic functions (e.g. AND, OR), and how do they differ from simple logic gates?
- How does "masking" work using bitwise operations?
- How does shifting compare with rotating the bits of a binary word?
- How is it possible for a digital computer to perform mathematical functions its ALU is not directly capable of?
- Why is it important that a computer program be able to execute instructions in a particular order?
- What is a *primitive function* for an ALU?
- What are some practical examples of *function composition*?
- What are some practical uses for status flag bits in an ALU?

1.2 Challenging concepts related to digital computing circuits

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

- **Bitwise logical operations** – applying AND, OR, and XOR logical operations to respective bits of binary words is not a straight-forward concept, but is made more understandable by drawing out the binary word(s) in question and if necessary drawing small logic gates taking inputs from those bits to generate the respective bits of the output word. Another helpful perspective for understanding the purpose of bitwise operations is to view AND functions as *forcing* a 0 output if any input is 0, and OR functions as *forcing* a 1 output of any input is a 1.

The *Derivations and Technical References* chapter contains a section providing an internal design for a four-bit ALU, showing how this useful device may be made from familiar logic gates and other digital circuit elements. The same chapter also contains a section revealing the internal construction of simple binary addition circuits.

1.3 Recommendations for instructors

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

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

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

  Assessment – Students show how each of the ALU's output words were obtained from the input word(s) by the author in the Tutorial chapter's examples.

- **Outcome** – Apply the concept of bitwise logical functions to binary words

  Assessment – Provide pairs of equal-length digital words and challenge students to perform bitwise-AND, bitwise-OR, and bitwise-XOR operations to each of them; e.g. pose problems in the form of the "Bitwise logical operations" Quantitative Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate ALU datasheets and properly interpret some of the information contained in those documents including word width, operations provided, etc.

Chapter 2

Case Tutorial

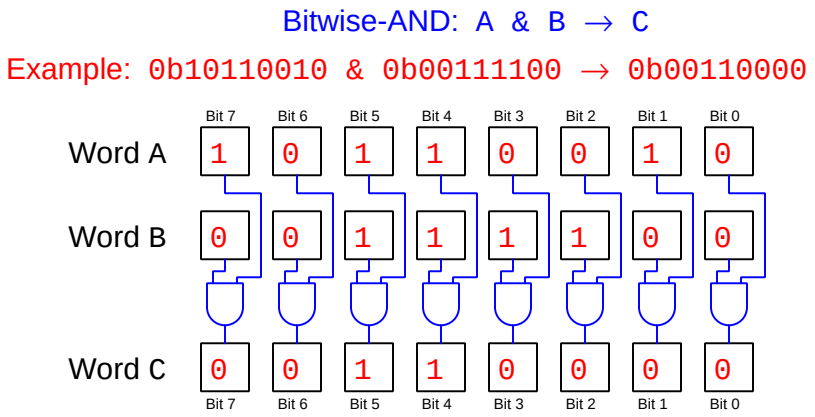
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as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

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

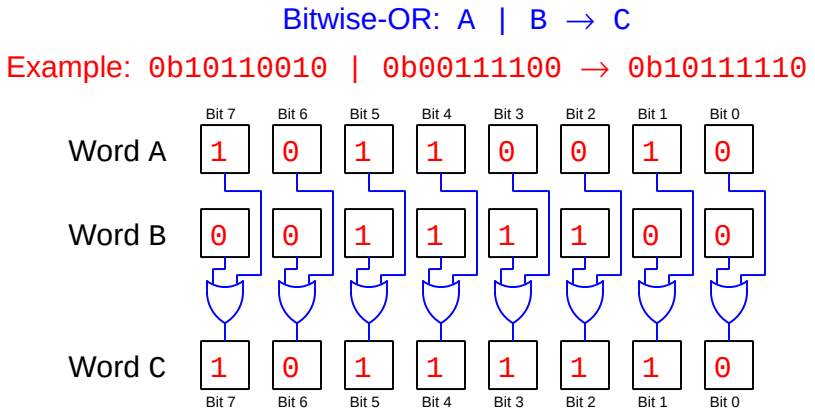
2.1 Example: bitwise logical operations

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

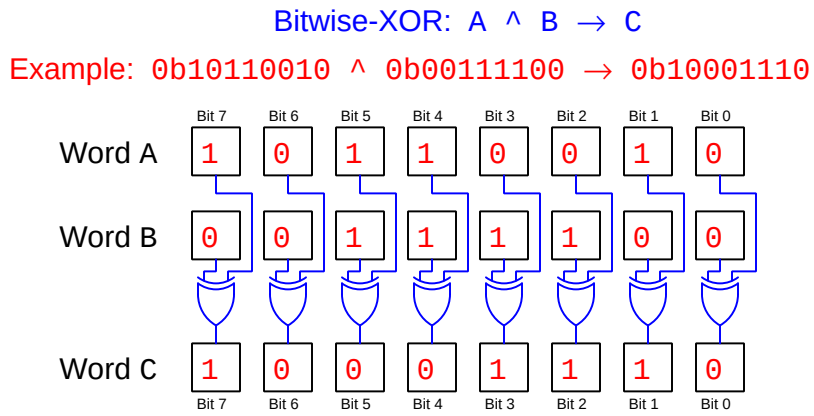
2.1.1 Bitwise-AND



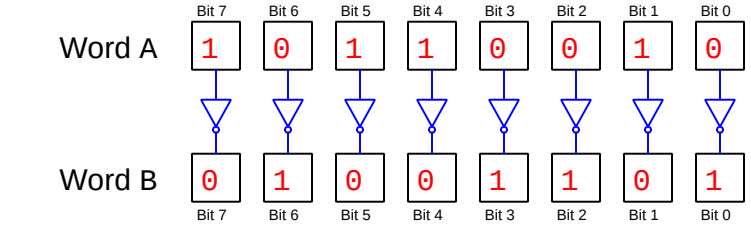
2.1.2 Bitwise-OR



2.1.3 Bitwise-XOR



2.1.4 Bitwise-complement



Chapter 3

Tutorial

Digital systems utilize *discrete states* to represent numerical values and other useful data. In electronic digital systems these discrete states typically take the form of "high" and "low" voltage levels defined on a scale appropriate to the DC power supply voltage powering the digital circuitry. The smallest division of digital data is called a *bit*, and each bit only has these two possible states (high or low, 1 or 0).

Multiple bits may be grouped together as "words[1]" to represent more complex data. For example, a *byte* (eight bits) could be used to represent an unsigned integer number with a range of 0 (00000000) to 255 (11111111), a signed integer number with a range of −128 (10000000) to 127 (01111111), or it could even be used to represent alphanumerical characters using the ASCII[2] code standard. The larger the word, the more things it can represent, which is why the trend in digital computer design is toward larger and larger word size.

It is not enough for a digital system to merely be able to receive, store, and display information represented by digital words – to be versatile the system must also be able to *manipulate* that data. Examples of productive data manipulation include *comparing* words to check for equality, performing *arithmetic operations* on one or more words, *shifting* or *rotating* bits within a word, *inverting* bits, and *masking*[3] bits.

Combinational logic circuits can perform all these functions, and when combinational logic is built to perform multiple types of data manipulation the resulting circuit is often referred to as an *arithmetic logic unit*, or *ALU*. Such circuits[4] are vital to the operation of digital computers, and will be used as monolithic networks in this Tutorial to illustrate some of these data-manipulation processes.

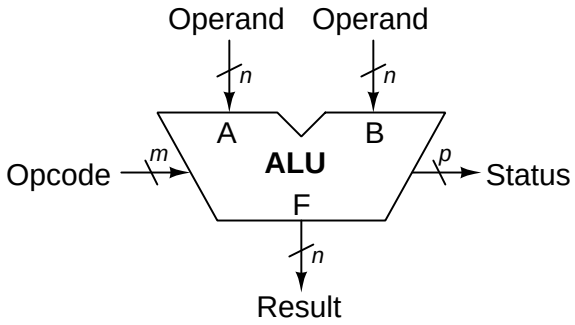
[1] A digital *word* is simply a collection of bits representing a greater set of combinations than possible with just one bit. The number of bits comprising a word is the *width* of that word. Words that are eight bits wide are called *bytes*. Whimsical names exist for other standardized word widths (e.g. 4 bits = *nybble*, 16 bits = *playte*, 32 bits = *dynner*) but are rarely used.

[2] American Standard Code for Information Interchange

[3] "Masking" refers to the selective copying of bits within a word, for example extracting just the four lower-order bits from a byte.

[4] It should be noted that ALUs are more commonly found within other integrated circuits (ICs) called *microprocessors* than found in isolation. In fact, at the time of this writing (2019) one major manufacturer of digital logic only lists *two* dedicated arithmetic ICs still in production: the 74181 ALU and the 74283 binary adder.

Arithmetic logic units are important enough to warrant their own logic symbols, akin to their own special "logic gate" symbol. The following diagram shows a generic ALU designed to operate on data words n bits wide[5]:



The two input busses at the top of the symbol (A and B) receive the digital words called *operands* to be processed by the ALU. The source of these operands is unimportant for the sake of this discussion, and could be as simple as a set of manual switches in a test circuit. In a digital computer these busses receive data directed (i.e. multiplexed) from a variety of sources including peripheral inputs and memory systems as directed by the microprocessor's program. The bus on the left-hand side receives a digital word called the *opcode* instructing the ALU which function to implement[6]. The results of the ALU's data manipulation are output at two different busses: the F (result) bus and the *status* bus, the "result" being the manipulated word and the "status" carrying additional useful information (e.g. data lines that activate when the two operands are equal to each other, when they are unequal to each other, when a carry bit has been generated from addition, etc.).

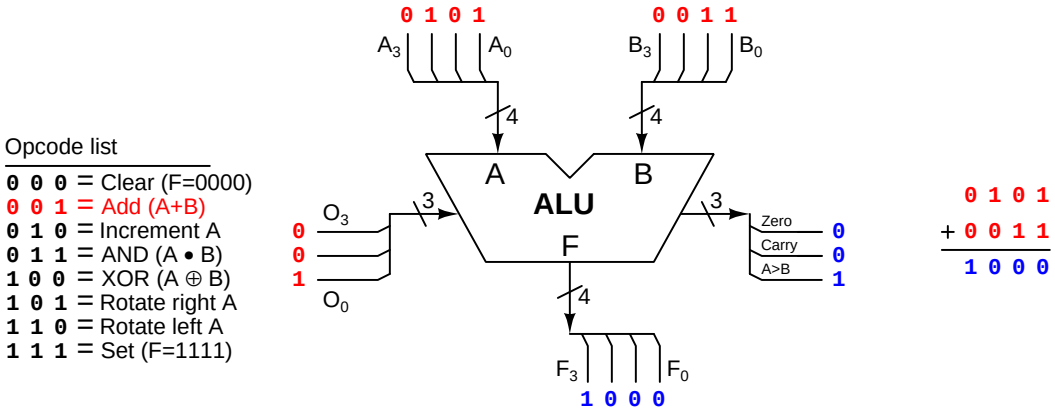
For the remainder of this Tutorial I will use a 4-bit ALU with all terminals shown to illustrate several common arithmetic and bitwise logical operations.

[5]The slash-mark through each line in this diagram show the lines to represent *data busses* consisting of multiple terminals. For example, a 4-bit ALU would have four electrical terminals for the A data input, four terminals for the B data input, and four terminals for the F function result output. Of course, any ALU packaged as its own integrated circuit would also have DC power supply terminals as well.

[6]Similar to how a multiplexer or demultiplexer circuit is commanded to select one particular data channel by a multi-bit *selection* word. For this reason, the opcode bus is often called "select" although I will refer to it as "opcode" because this more closely relates the function of an ALU within the microprocessor in which it is typically found.

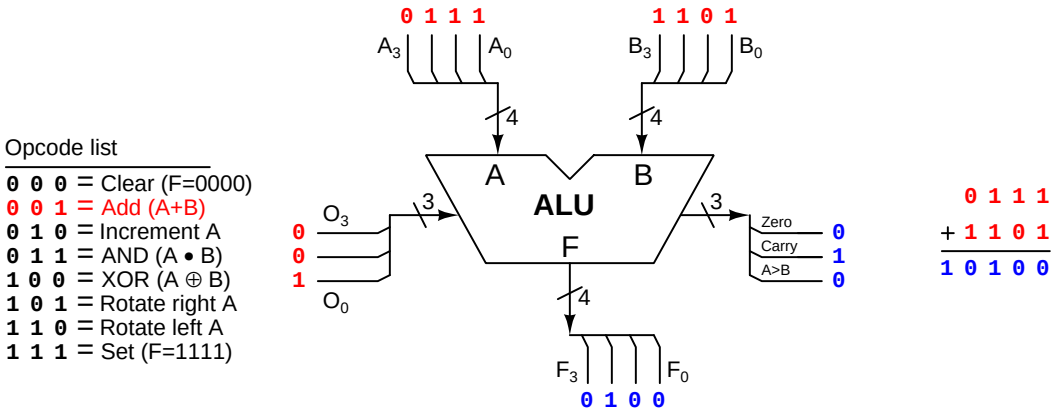
3.1 Arithmetic: addition

Consider the following example of a simple ALU instructed to add the two integer operands 0101 (five) and 0011 (three) resulting in a sum of 1000 (eight). For ease of understanding, all bit states input to the ALU are shown in red while all outputs from the ALU are shown in blue:



The actual process of addition highlighted on the right-hand side of the diagram is as easy as it looks: simply apply the same technique you learned for adding multi-digit decimal numbers, and you will get the same result shown here. Note the $A > B$ status flag bit that is set simply by virtue of the fact operand A is larger than operand B.

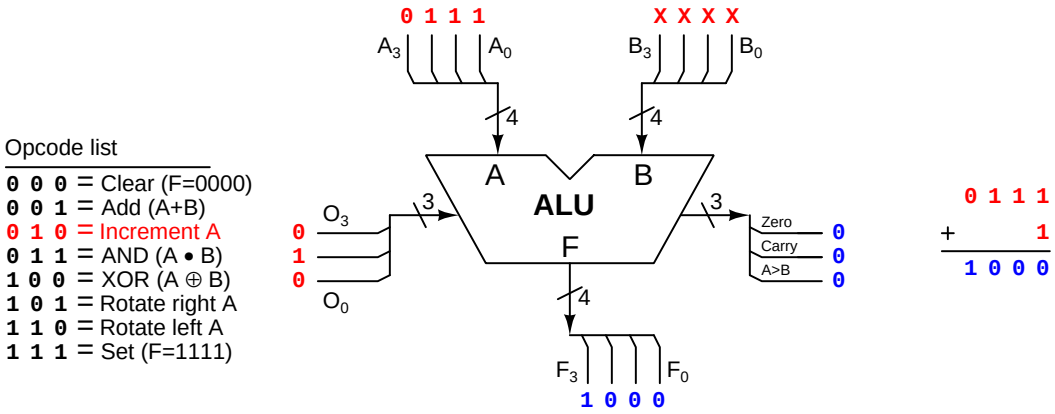
If we purposely feed operand values into the ALU that will yield a sum larger than four bits, we will see the carry status flag activate:



Here, the sum of A (seven) and B (thirteen) is Carry + F (twenty).

3.2 Arithmetic: increment

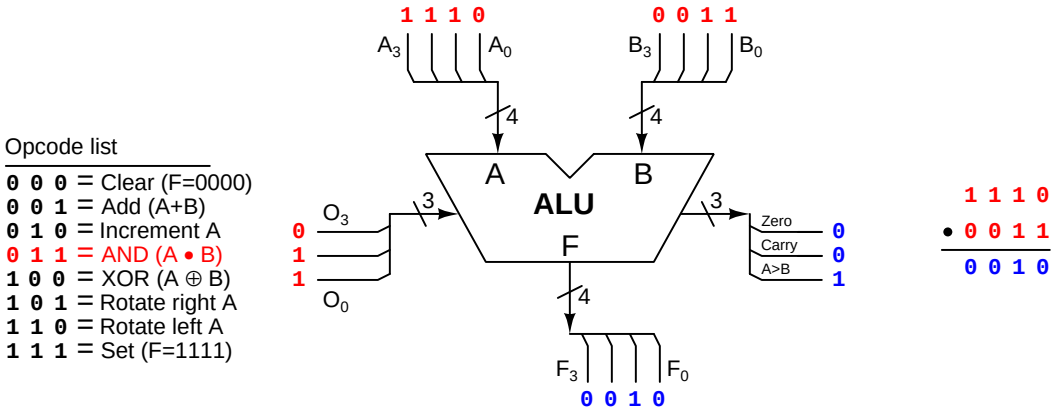
The *increment* function is extremely simple, as it simply adds one to the value of operand A:



Note how for this operation the status of operand B is irrelevant, which is why those input line statuses are shown with `X` ("don't care") characters.

3.3 Bitwise: AND

Our first bitwise logical operation, AND, simply applies the logical AND function to each pair of respective bits[7] in the two operands, the resulting bits comprising the ALU's output. In this case, we will AND 1110 with 0011:



Recall from the truth table of an AND function that any 0 input to an AND function guarantees a 0 output. Thus, by ANDing one digital word with another we prioritize all 0 bit states, so that the result contains every 0 bit contained in both words. The only way a 1 bit can exist in the result is if both words contained 1 bits in that same position.

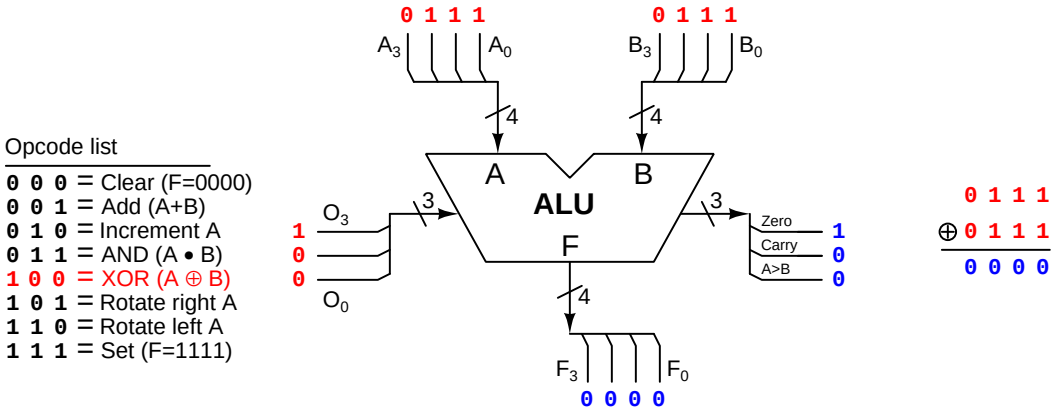
A very practical use for the bitwise AND operation is called *masking*. This is where we desire to extract only certain bits from a digital word while "masking off[8]" the others. In this case, we could say that the two 0 bits within the B operand 0011 "masks off" the two most-significant bits of operand A. Alternatively, we could also say that the single 0 bit within the A operand 1110 "masks off" the least-significant bit (LSB) of B.

[7] For example, F_0 is equal to A_0 AND B_0; F_1 is equal to A_1 AND B_1, etc.

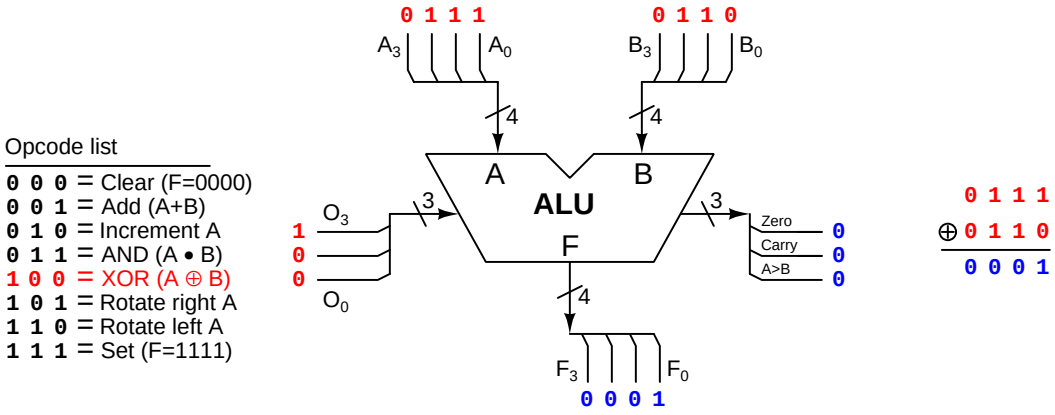
[8] This is not unlike using masking tape to cover something you would rather not get painted, allowing you to freely apply paint while not obscuring what has been masked off. When we "mask" bits we are declaring which bit-positions in the word matter to use and which do not.

3.4 Bitwise: XOR

Another very useful bitwise logical operation is Exclusive-OR, otherwise known as XOR. Recall from the truth table of an XOR function that the output will be `1` only if the inputs are *different* states. This makes the XOR function a natural fit for comparing two digital words for equality. For example, consider what happens if we make operands A and B both equal to `0111`:

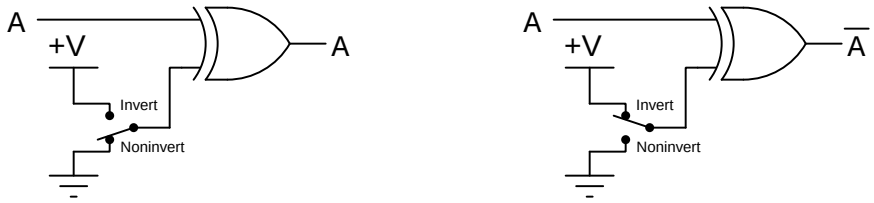


The operands' equality is indicated by the ALU's *Zero* status flag terminal outputting a `1` state following the XOR operation. Contrast this with a scenario where the two operands are unequal:

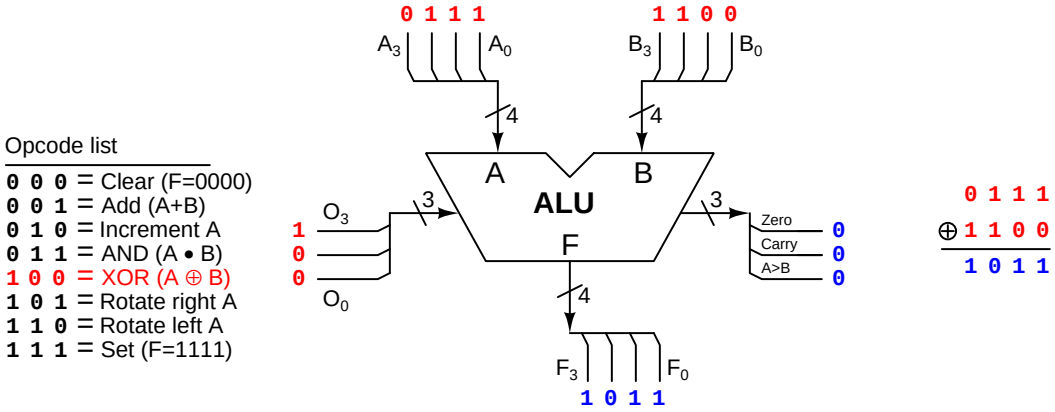


Following the XOR operation the Zero status flag is at a state of `0`, signifying unequal operands.

Bitwise XOR operations are also useful for selectively toggling (i.e. inverting) bits of a digital word, by using the XOR logical function as a *controlled inverter*:



If for some reason we wished to invert the middle two most significant bits of a 4-bit digital word, all we would have to do is XOR that word with `1100` as shown here:



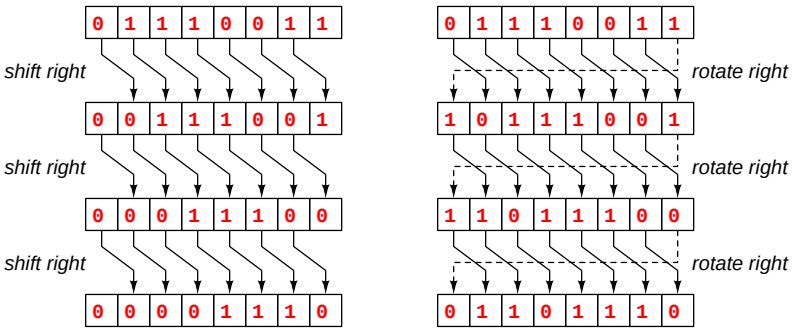
Here, we have successfully inverted the left-two bits of operand A, turning `0111` into `1011`.

A practical use of this inversion technique is to XOR a word with another operand containing all `1` bits (e.g. `1111` for a 4-bit ALU), the result of this being called the *one's complement* of the original word.

3.5 Bitwise: shift/rotate

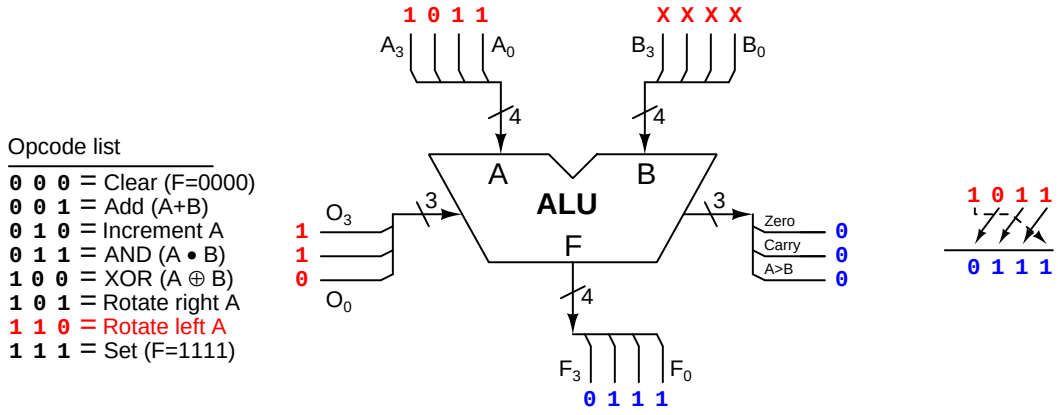
Two very common bitwise logical operations are *shift* and *rotate* (the latter also known as *circular shift*). In either case, the bit-states within a digital word are moved one place to the right or to the left, the difference between "shift" and "rotate" being what happens to the end-most bits after the shift has taken place. In the case of a plain shift operation, one end bit "falls off" and disappears while at the other end of the digital word a `0` state takes the place of the bit that got shifted the other direction. In the case of a rotate operation, the bit that "falls off" the word at one end gets recycled onto the other end of the word.

The following examples contrast a series of *shift-right* operations versus a series of *rotate-right* operations starting with the same digital word (`01110011`):



With enough shift operations, the original word's bits become entirely replaced by `0`'s. However, rotate operations merely circulate the bit-states and never replace them with `0`'s.

An example of our hypothetical ALU performing a *rotate left* operation is shown in the following illustration:

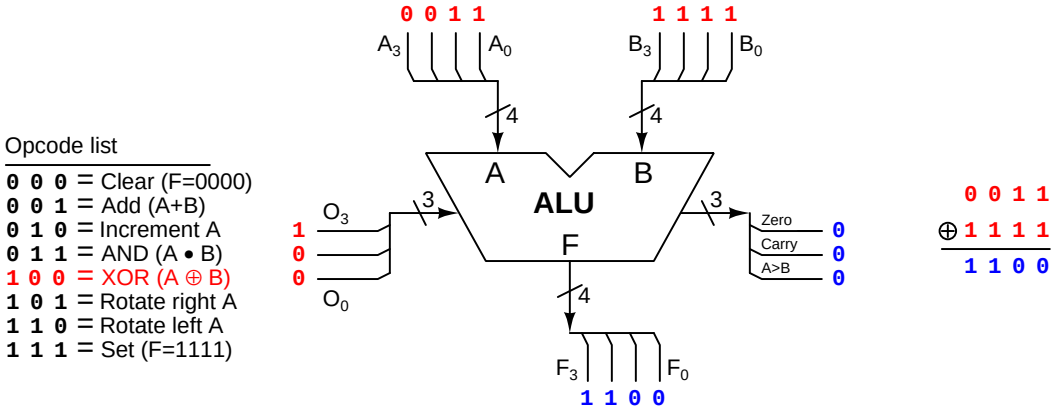


3.6 Arithmetic: subtraction

Our hypothetical ALU conspicuously lacks a subtraction opcode, so how can we subtract two binary numbers using this ALU? They key is a technique known as *two's complement addition*, and it relies on the properties of *signed integer* values where we regard the most-significant bit (MSB) as having a negative place-weight. Also, we will require a means to store and recall results from the ALU and recycle those results back into the ALU as operands. The electrical details of storing and recalling results will not be shown here, but know that in a microprocessor circuit there will be multiplexers and registers and sequencing logic designed just for this purpose.

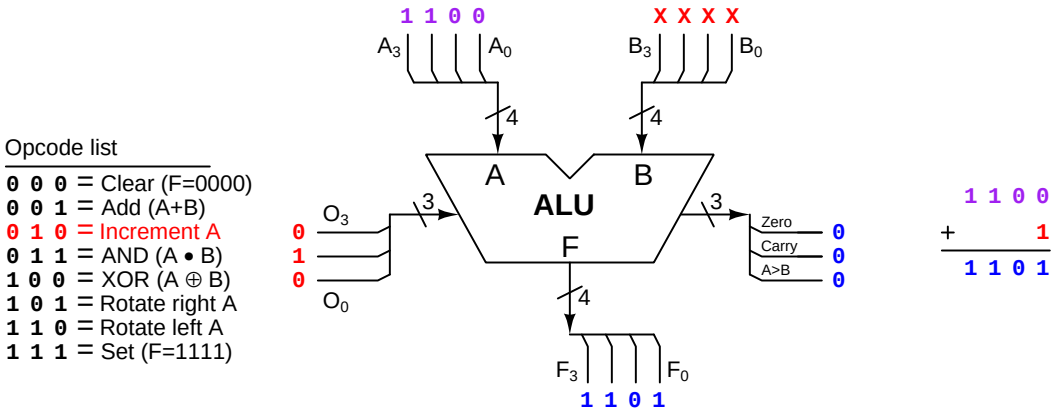
Suppose we wish to have the ALU subtract three from five to yield a difference of two ($5-3=2$). Since we only have an ADD opcode but no SUBTRACT opcode, an alternative to direct subtraction is to add *negative three* to five. Now all we need is a way to use the opcodes available to us to convert an input value of (positive) three to negative three. The "two's complement" technique begins by first taking the positive value and inverting all bit states, in this case `0011` (three) becoming `1100`. We call this result the *one's complement* of the number. Next, we increment that value to yield the two's complement: `1101`. Note that for a four-bit signed integer (with MSB place-weight being -8) the binary result `1101` is indeed equal to negative three ($-8+4+1=-3$). All we need to do now is add negative three (`1101`) to five (`0101`) and disregard any carry bit generated, and the result will be two.

A simple way to invert all bits of operand A is to use the XOR opcode and set B to `1111`, exploiting the XOR function's ability to act as a controlled inverter:



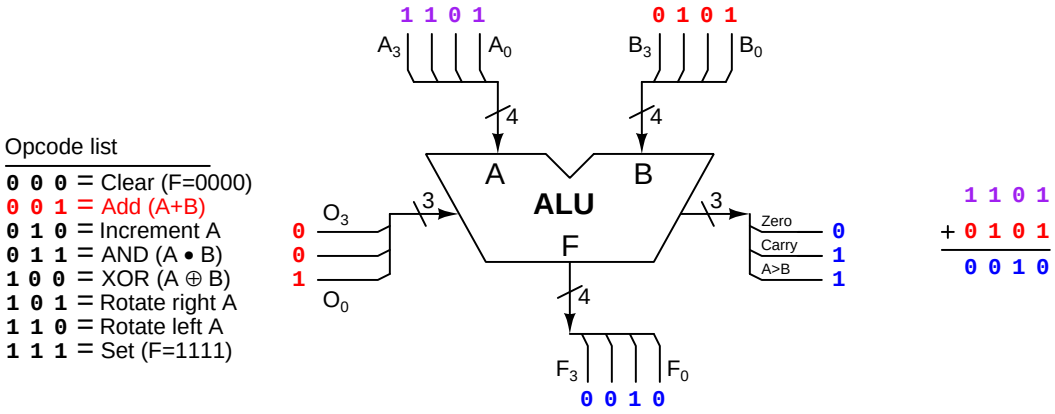
The result, `1100`, must now be stored in a register for later re-use.

In the next step of this multi-step subtraction process, we recall our result from the register and send it back to the ALU as operand A (colored violet to represent its origin as having come from a previous ALU operation). Invoking the increment operation, we convert this one's complement value into a proper two's complement value which in the signed integer framework will be the negative value of our original number:



Once again, the system must store this result in a register for later re-use.

In the next step we recall that stored value of negative three and send it back to the ALU as one of the operands, setting the other operand to a value of five. Now we invoke the ADD opcode and finally get the difference of two as our answer to $5 - 3$:



We must ignore the carry status flag since we are using signed 4-bit integers, which means a fifth bit has no meaning. The result is simply `0010` which of course is a positive two.

We may summarize this sequence of steps in tabular format:

Step	**Operand A**	**Operand B**	**Opcode**	**Result**	*Comment*
1	0011	1111	XOR (100)	(store)	One's complement of 3
2	(recall)	– – – –	Increment A (010)	(store)	Two's complement of 3
3	(recall)	0101	Add (001)	(answer)	$-3 + 5$

Such sequences of operations form the basis of *assembly language programming* for microprocessors: a set of steps, each one with an opcode and at least one operand, executed in a particular order to achieve the desired results. The "step" value for a microprocessor program is held in a binary counter called the *program counter* or *PC*, with the microprocessor able to direct data to and from multiple storage registers, to and from other location in memory, and also able to "jump" out of sequence to different steps.

While this exercise in sequential ALU operations may seem tedious, it is an example of a very fundamental and important concept in digital computing called *function composition*[9], whereby we synthesize higher-level functions by combining multiple *primitive functions* offered by the ALU and other microprocessor circuitry. So long as the ALU's "instruction set" of opcodes contains operations of sufficient versatility, it is theoretically possible to have it perform *any* arithmetic or logical operation via composition.

[9] Function composition is easy to illustrate with simple arithmetic functions, for example multiplication. A multiplication problem such as 5×3 is equivalent to repeated addition: 5 added to itself three times $(5 + 5 + 5)$. In a similar manner, digital microprocessors combine simple mathematical and logical functions to achieve more complex *composite* functions.

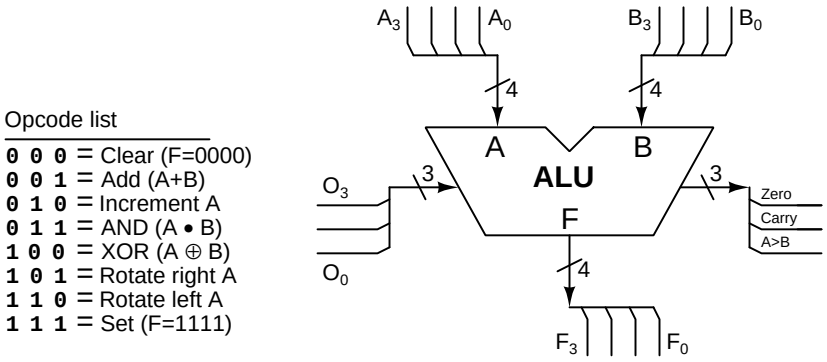
Chapter 4

Derivations and Technical References

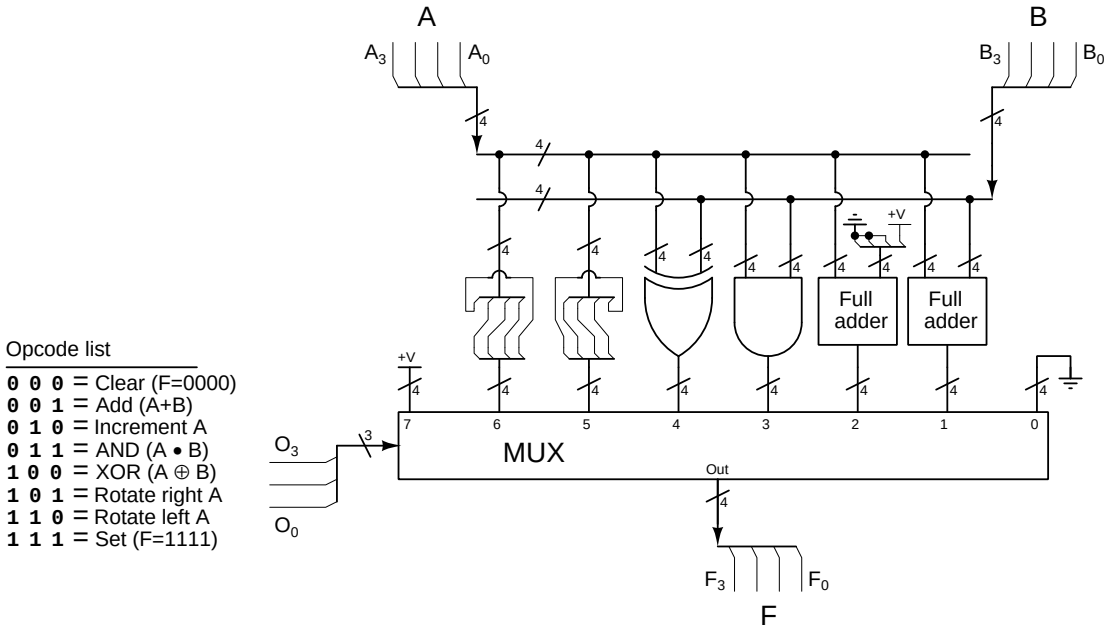
This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

4.1 Simple four-bit ALU

An ALU may be thought of, at least conceptually, as a set of logical functions combined with a multiplexer to select which of those functions becomes the ALU's output. Take for instance the hypothetical ALU used in the Tutorial:



This ALU offers eight unique operations, each one specified by a three-bit opcode. Each operation acts upon either one or two four-bit operands, A by itself or A combined somehow with B. We may represent the operand and result circuitry of this ALU in terms of logic gates and multiplexers:

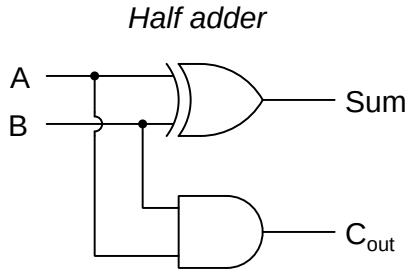


Real ALUs are not designed quite like this, because it would require more transistors than absolutely necessary. However, the concept of having individual functions (e.g. full adder, AND,

XOR) act on the operand(s) and then having a multiplexer select which of those function's output gets sent to the result (F) terminals makes the functionality of an ALU easier to grasp. From this conceptual diagram we can see that the "opcode" is nothing more than the selection word commanding a multiplexer to choose one of its word inputs over the others.

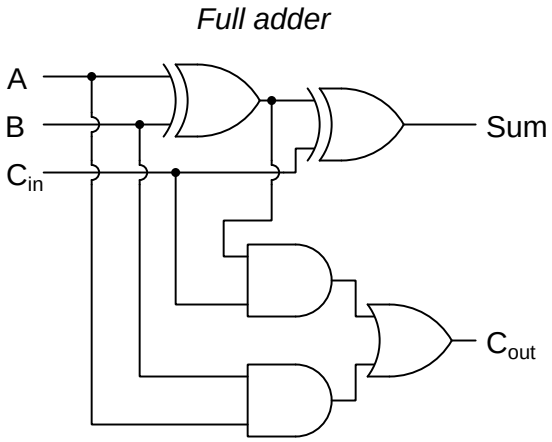
4.2 Binary adder circuits

A simple gate circuit designed to add two binary bits appears in the following diagram:

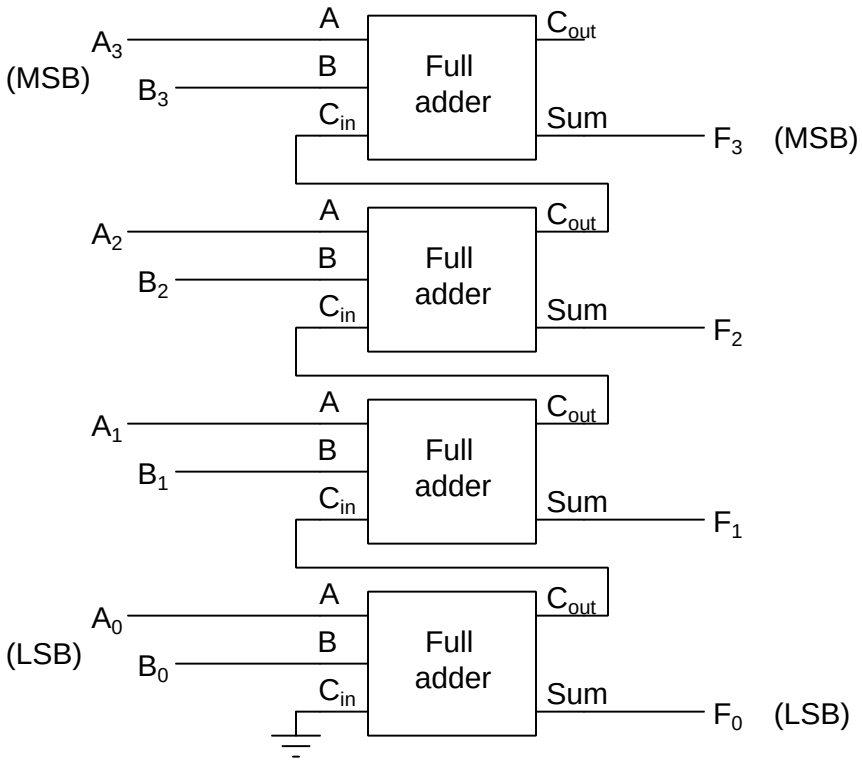


This circuit is known as a *half adder* because it is incomplete: it is able to generate a carry bit, but it cannot *receive* a carry bit from another adder circuit which is important if we wish to gang multiple adders together so as to be able to add more than just two bits together.

A *full adder* circuit is an expansion on the concept of a half adder:



Creating an adder circuit capable of summing two binary *words* is as simple as cascading a set of full adders together. Here we see a 4-bit adder made of four cascaded full adders:



A problem with this design, however, is the same *ripple* effect we saw when we formed a binary counter using cascaded JK flip-flops: the output word (F) does not necessarily update all of its bits simultaneously. Any of the full adders receiving a carry in (C_{in}) bit from a lower-order adder must wait for that lower-order adder to settle into its new C_{out} state. The result is that the output word's LSB settles first, followed by the next higher-order bit (F_1) and so on through to the MSB. In applications where the adder's output drives a display and nothing else, this will not be a concern because the ripple effect occurs too fast for human eyes to see.

We can prevent false sums from emerging from the adder by passing the output word through a D-type latch or D-type flip-flop register with a clock speed slow enough to allow all the output bits to settle before "presenting" them to external circuitry. However, this introduces a speed limit for our adder, as it cannot compute sums faster than it takes for *all* of its output bits to stop rippling.

A solution to this problem is the so-called *look-ahead adder* which uses combinational logic to "predict" carry states without having to wait for cascaded adders to sequentially compute each one. Even this solution is imperfect, although it is faster than the ripple adder design.

Chapter 5

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?

5.1 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.

5.1.1 Reading outline and reflections

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

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

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

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

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

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

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

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

5.1.2 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.

Discrete signal

Word

Arithmetic Logic Unit (ALU)

Operand

Opcode

AND function

Masking

XOR function

One's complement

Two's complement

Bit-shifting

Bit-rotating

Assembly language

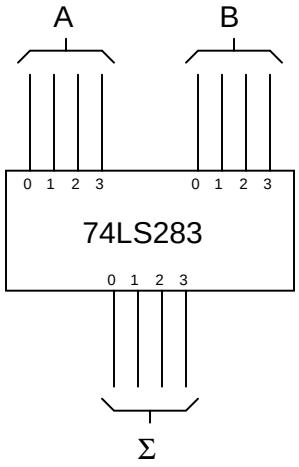
Microprocessor

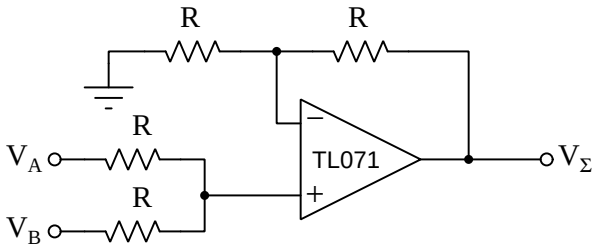
Program counter

Functional composition

5.1.3 Digital versus analog addition

Compare the following two circuits, the first one being a digital adder and the second one being an analog summer:





These two circuits perform the same mathematical function, yet the manners in which they perform this function are quite different. Compare and contrast the digital adder and the analog summer circuits shown here, citing any advantages or disadvantages of each.

Challenges

- We are all familiar with digital computers and digital calculators. What would a fully *analog* computer look and function like, in comparison?

5.1.4 Half versus Full adders

Explain in your own words how a *half adder* differs in form and function from a *full adder*.

Challenges

- If we ignore the state of a half adder's *carry* output line, what arithmetic or logical operation does it perform?

5.1.5 74181 ALU

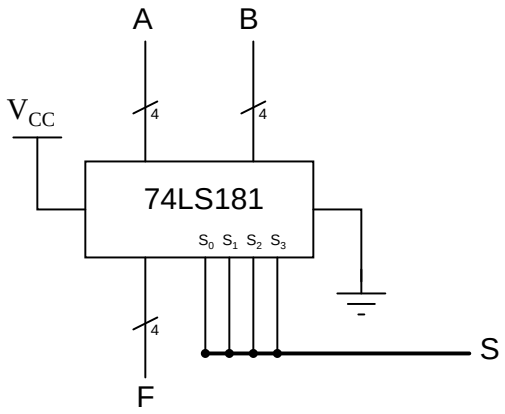
Consult the datasheet for a model 74181 *arithmetic logic unit*, and determine how its various modes of operation (addition, subtraction, comparison) are selected.

Challenges

- An interesting feature of the 74AS181 is that it provides "arithmetic" functions as well as "logic" functions. These two modes could also be referred to as "binary" and "boolean", respectively. Explain what distinguishes these two operating modes from one another, and why they are classified differently.

5.1.6 Digital line notation

Explain the meaning of the digital lines A, B, F, and S as drawn in the following schematic diagram:



Challenges

- Why represent digital lines in this manner?

5.1.7 Shift-right sequential program

In an arithmetic logic unit (ALU) is connected to a set of multiplexers, registers, and busses, it becomes possible to specify sequences of operations useful for performing more advanced functions. An example of this is discussed in the Tutorial, showing how subtraction may be performed using multiple operations offered by the simple ALU.

Assuming the use of this same ALU, write a sequential program for performing a *shift-right* operation.

Step	**Operand A**	**Operand B**	**Opcode**	**Result**	*Comment*
1					
2					
3					

Challenges

- If the ALU offered *shift* operations but no *rotate* operations, could a "rotate" function be made using "shift"?

5.2 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.

5.2.1 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.

5.2.2 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	

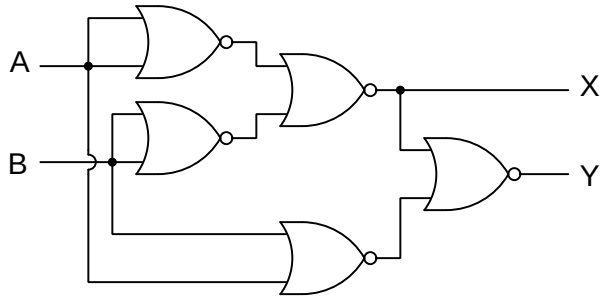
Note how the square-root term (y) is calculated in cell `C1`, and the denominator term (z) in cell `C2`. This makes the two final formulae (in cells `B1` and `B2`) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly 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.*

5.2.3 NOR-based half-adder

The following half-adder circuit uses nothing but NOR gates:



Identify which output is the Sum and which is the Carry.

Challenges

- What advantage might there be in using nothing but NOR gates to make a half-adder network, as opposed to different types of gates (e.g. an AND gate and an XOR gate)?

5.2.4 Binary addition

Add the following binary numbers:

```
  10010          1011101          10011
+ 1100         + 1000000      + 1111101

  10011001       11000011       1001100
+   100111     +   101111     + 1100101
```

Challenges

- Compare and contrast the manual processes of adding decimal numbers versus adding binary numbers.

5.2.5 One's complementation

What is the *one's complement* of a binary number? If you had to describe this principle to someone who just learned what binary numbers are, what would you say?

Determine the one's complement for the following binary numbers:

- `10001010` =
- `11010111` =
- `11110011` =
- `11111111` =
- `11111` =
- `00000000` =
- `00000` =

Challenges

- Identify the logic circuitry necessary to perform one's complementation.
- Is the one's complement `11111111` identical to the one's complement of `11111`? How about the one's complements of `00000000` and `00000`? Explain.

5.2.6 Two's complementation

Convert the following eight-bit two's complement binary numbers into decimal form:

- `01000101` =
- `01110000` =
- `11000001` =
- `10010111` =
- `01010101` =
- `10101010` =
- `01100101` =

Challenges

- Identify the most-positive and the most-negative quantities representable in eight-bit two's complement notation.

5.2.7 Signed binary addition

Two's complement notation really shows its value in binary addition, where positive and negative quantities may be handled with equal ease. Add the following byte-long (8 bit) two's complement numbers together, and then convert all binary quantities into decimal form to verify the accuracy of the addition:

```
  00110101        01110110        00111101
+ 00001100      + 00000010      + 11111011
----------      ----------      ----------

  00001010        11111110        11111110
+ 10010101      + 11011101      + 11111101
----------      ----------      ----------

  10110111        00111101        11111011
+ 01110110      + 00111011      + 11111011
----------      ----------      ----------

  10000001        01111011        01111111
+ 10010001      + 00111101      + 10000001
----------      ----------      ----------
```

Challenges

- What happens to the MSB carry?
- Some of the binary sums are actually incorrect. Explain why.
- Explain how to determine whether overflow has occurred just by examining the binary bits of the operands (i.e. *addend* and *augend*) and sum.

5.2.8 Multiplication by point-shifting

If the decimal point of a decimal number is shifted one place to the left, what happens to the value of that number? What if the point shifts one place to the right?

What happens when we do the same to the binary point of a binary number?

Challenges

- How does the 74181 ALU implement a bit-shift.

5.2.9 Bitwise logical operations

Determine the result of each of the following bitwise operations:

- `0b100101` bitwise-AND `0b111001` =
- `0x4C` bitwise-AND `3E` =
- `0b10110011` bitwise-OR `0b00010111` =
- `0xB5` bitwise-OR `0xA2` =
- `0b1110011` bitwise-XOR `0b0011001` =
- `0x22` bitwise-XOR `0x79` =

Challenges

- The Exclusive-OR (XOR) function may be thought of as a *controlled inverter*. Explain why.
- Which bitwise function may be used to selectively force certain bits of a word to high (1) states while leaving the other bits unaffected?
- Which bitwise function may be used to selectively force certain bits of a word to low (0) states while leaving the other bits unaffected?

5.3 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.

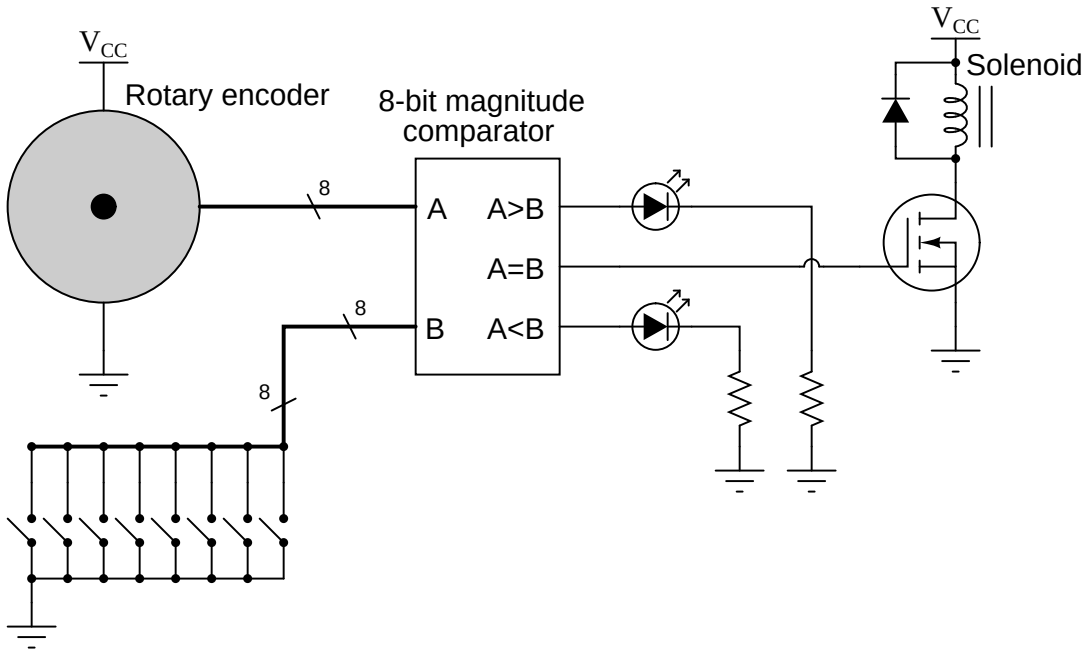
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of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

5.3.1 Rotary encoder

The purpose of this circuit is to provide indication of when the rotary encoder shaft is in a particular position (matching the setting of the 8-position switch array):



Identify specific component failures that could result in the solenoid coil not energizing in this condition.

Challenges

- Does this circuit function with binary encoding, Gray encoding, or either?

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

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.

18-20 August 2024 – added another section to the Introduction chapter specifically for challenges related to this module's topic. Also added a Case Tutorial chapter with a section on bitwise operations with graphic images borrowed from mod_c. Re-named the "Bitwise: rotate" Tutorial subsection to be "Bitwise: shift/rotate".

22 May 2024 – added a new Quantitative Reasoning question on half-adder logic circuits.

18 January 2024 – added more spacing between addition problems in the "Signed binary addition" Quantitative Reasoning question to make it appear more pleasing and less confusing on the page.

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

26 August 2021 – instructor note added to "Signed binary addition" Quantitative Reasoning question.

10 May 2021 – commented out or deleted empty chapters.

21 January 2021 – minor edits to the Tutorial.

4 September 2020 – added Quantitative Reasoning question on bitwise operations.

27 August 2020 – corrected a typographical error pointed out by Ty Weich.

23 August 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions. Also, re-named "Simplified Tutorial" to "Tutorial" and similarly edited all references to it throughout the

module.

28 January 2020 – added Foundational Concepts to the list in the Conceptual Reasoning section.

25 December 2019 – finished writing Tutorial.

24 December 2019 – began writing Tutorial.

23 December 2019 – document first created.

Index