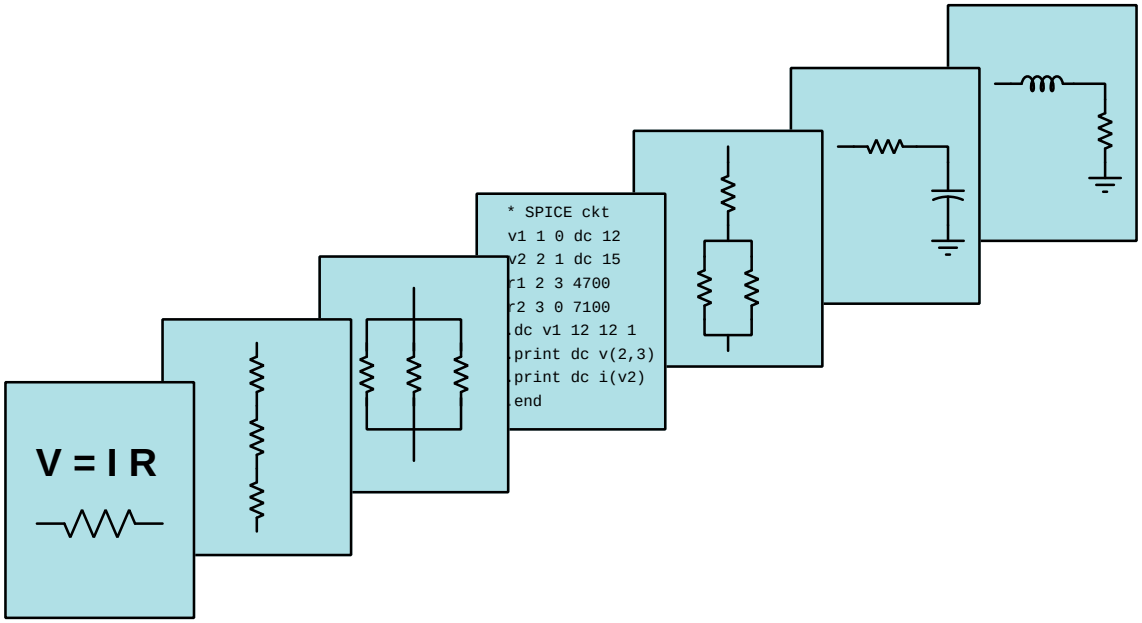
Modular Electronics Learning (ModEL) project



Latching Logic

© 2019-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 29 April 2026

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

Contents

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

2 Case Tutorial **7**
- 2.1 Example: timing diagrams for combinational gate circuits . . . 8
- 2.2 Example: timing diagrams for latches and flip-flops . . . 12
  - 2.2.1 Example: enabled SR latch timing diagram . . . 12
  - 2.2.2 Example: enabled D latch timing diagram . . . 13
  - 2.2.3 Example: SR flip-flop timing diagram . . . 14
  - 2.2.4 Example: JK flip-flop timing diagram . . . 15
  - 2.2.5 Example: D flip-flop timing diagram . . . 16
  - 2.2.6 Example: cascaded flip-flops timing diagram . . . 17
  - 2.2.7 Example: cascaded latch/flip-flops timing diagram . . . 18
- 2.3 Switch contact bounce . . . 19
- 2.4 Quadrature pulse signal generation . . . 21

3 Tutorial **25**
- 3.1 Logic functions . . . 25
- 3.2 Latch circuits . . . 28
- 3.3 Set-Reset latches . . . 30
- 3.4 SR enabled latches and flip-flops . . . 33
- 3.5 D-type latches and flip-flops . . . 36
- 3.6 JK flip-flops . . . 39
- 3.7 Master-slave flip-flops . . . 40
- 3.8 Preset and clear inputs . . . 41

4 Derivations and Technical References **43**
- 4.1 Logic families . . . 44
- 4.2 Digital pulse criteria . . . 47

5 Programming References **51**
5.1 Programming in C++ . . . 52
5.2 Programming in Python . . . 56
5.3 Modeling an SR latch using C++ . . . 61
5.4 Modeling an SR flip-flop using C++ . . . 64
5.5 Modeling a JK flip-flop using C++ . . . 67

6 Questions **71**
6.1 Conceptual reasoning . . . 75
6.1.1 Reading outline and reflections . . . 76
6.1.2 Foundational concepts . . . 77
6.1.3 CMOS gate equivalent of motor control circuit . . . 80
6.1.4 UV lamp control circuit . . . 82
6.1.5 Wrong way siren circuit . . . 84
6.1.6 Simple one-shot circuit . . . 86
6.1.7 Frequency division . . . 87
6.2 Quantitative reasoning . . . 88
6.2.1 Miscellaneous physical constants . . . 89
6.2.2 Introduction to spreadsheets . . . 90
6.2.3 Timing diagram for an SR latch . . . 93
6.2.4 Timing diagram for an enabled SR latch . . . 94
6.2.5 Timing diagram for a D latch . . . 95
6.2.6 Timing diagram for a D flip-flop . . . 96
6.2.7 Cascaded D latches . . . 97
6.3 Diagnostic reasoning . . . 98
6.3.1 Identifying fault in a NOR-based SR latch circuit . . . 99
6.3.2 Sound-controlled lamp . . . 100

A Problem-Solving Strategies **101**

B Instructional philosophy **103**
B.1 First principles of learning . . . 104
B.2 Proven strategies for instructors . . . 105
B.3 Proven strategies for students . . . 107
B.4 Design of these learning modules . . . 108

C Tools used **111**

D Creative Commons License **115**

E References **123**

F Version history **125**

Index **127**

Chapter 1

Introduction

1.1 Recommendations for students

A digital *latch* is a logic function with a memory – it is able to "remember" its last state. Like a mechanical toggle switch that can be "flipped" on (and remain on) then "flipped" off (and remain off), a latch circuit exhibits a similar toggling behavior. Latches are useful for storing digital data, for synchronizing certain events in a large digital circuit, and for discrete device control such as electric motor starters where we might want a momentary event to toggle an electric motor's state.

Important concepts related to latching digital circuits include **logic states**, **logic levels**, **high** and **low** logic states, **logic functions**, **truth tables**, **seal-in contact**, **set** versus **reset** states, **feedback**, **timing diagrams**, **active-low** and **active-high** inputs and outputs, **race condition**, **disallowed** or **invalid** state, **bistable** circuit, switch **bounce**, **enable** input, **monostable** circuit, **flip-flops** versus latches, **edge-triggering**, **propagation delay**, **set-up** and **hold** times, **registers**, **toggling**, **master-slave**, and **quadrature** signals.

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

- How might an experiment be designed and conducted to measure the minimum necessary set-up time for a flip-flop? 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 measure the minimum necessary hold time for a flip-flop? What hypothesis (i.e. prediction) might you pose for that experiment, and what result(s) would either support or disprove that hypothesis?
- What essential property must a digital circuit possess in order to be able to *latch*?
- How may a relay-based circuit be made to latch?
- How may a gate-based circuit be made to latch?
- What can a timing diagram show us that a truth table cannot?
- What is a *clock signal*, and what is it used for?

- What does it mean for a digital circuit to have an *active-high* versus an *active-low* input line?
- What is a *monostable* circuit?
- What is a *bistable* circuit?
- What causes a *race condition* in a digital circuit, and why is this a bad thing?
- What does it mean to "de-bounce" a switch signal?
- What distinguishes a flip-flop from a latch?
- What must be added to a latch to make it into a flip-flop?
- Why are adequate *set-up* and *hold* times important for digital circuits?
- How may flip-flops be used to synchronize data in a digital system?
- How is it possible to digitally discern the direction of rotation for a rotary encoder?
- What distinguishes a JK flip-flop from an SR flip-flop?
- What is a digital *register*, and what is it used for?
- What distinguishes a *synchronous* reset or clear input from an *asynchronous*, and how do latch functions with these different types of inputs behave differently from each other?

1.2 Challenging concepts related to latching logic

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

- **Seal-in contacts** – the simplest and most common way to make momentary-contact "Start" and "Stop" switches latch a load's state on and off is to wire a relay in parallel with the Start switch to "seal in" that circuit once the relay energizes. This, however, complicates analysis of the circuit by granting it *states*. State-based logic is more complex than combinational logic because the status of state-based logic depends not only on input conditions but also on past history. This means a person must analyze the circuit before and after input condition changes to determine how it will respond.

- **Set-up and Hold times** – any digital circuit using timing pulses will have set-up and hold-time requirements, which are *minimum-time specifications* that all input pulse signals must meet in order for the circuit to reliably work. Always remember that these specified times are *minimum*, which means it's perfectly okay to exceed them but not safe to time the signals for less.

- **Race conditions** – latching logic circuits exhibit state-based behavior, and one of the complexities of this is the potential for *race conditions* where two or more portions of the circuit "race" each other to achieve control over the other(s).

- **Synchronous versus Asynchronous inputs** – latch circuits often rely on a "clock" signal to synchronize actions, with some inputs having effect only when the clock pulse permits (i.e. "synchronous") and other inputs having immediate effect regardless of clock state (i.e. "asynchronous"). Clear and preset inputs may fall into either of these categories depending on the internal design of the flip-flop.

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 timing diagrams were obtained by the author in the Tutorial chapter's examples.

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

- **Outcome** – Apply foundational circuit concepts to the analysis of latching logic circuits

  Assessment – Determine how a latching logic circuit performs its intended practical function; e.g. pose problems in the form of the "Wrong way siren circuit" Conceptual Reasoning question.

  Assessment – Sketch timing diagrams showing logical states given input waveforms; e.g. pose problems in the form of the "Timing diagram for an SR latch" and "Timing diagram for an enabled SR latch" and "Timing diagram for a D latch" and "Timing diagram for a D flip-flop" and "Cascaded D latches" Quantitative Reasoning questions.

- **Outcome** – Diagnose a faulted latching logic circuit

  Assessment – Predict the effect(s) of a single component failing either open or shorted in a latching circuit; e.g. pose problems in the form of the "Sound-controlled lamp" Diagnostic Reasoning question.

  Assessment – Determine the probability of various component faults in a latch circuit given symptoms and measured values; e.g. pose problems in the form of the "Identifying fault in a NOR-based SR latch circuit" Diagnostic Reasoning question.

- **Outcome** – Independent research

  Assessment – Locate flip-flop datasheets and properly interpret some of the information contained in those documents including proper logic levels, maximum clock frequency, set-up and hold time requirements, etc.

Chapter 2

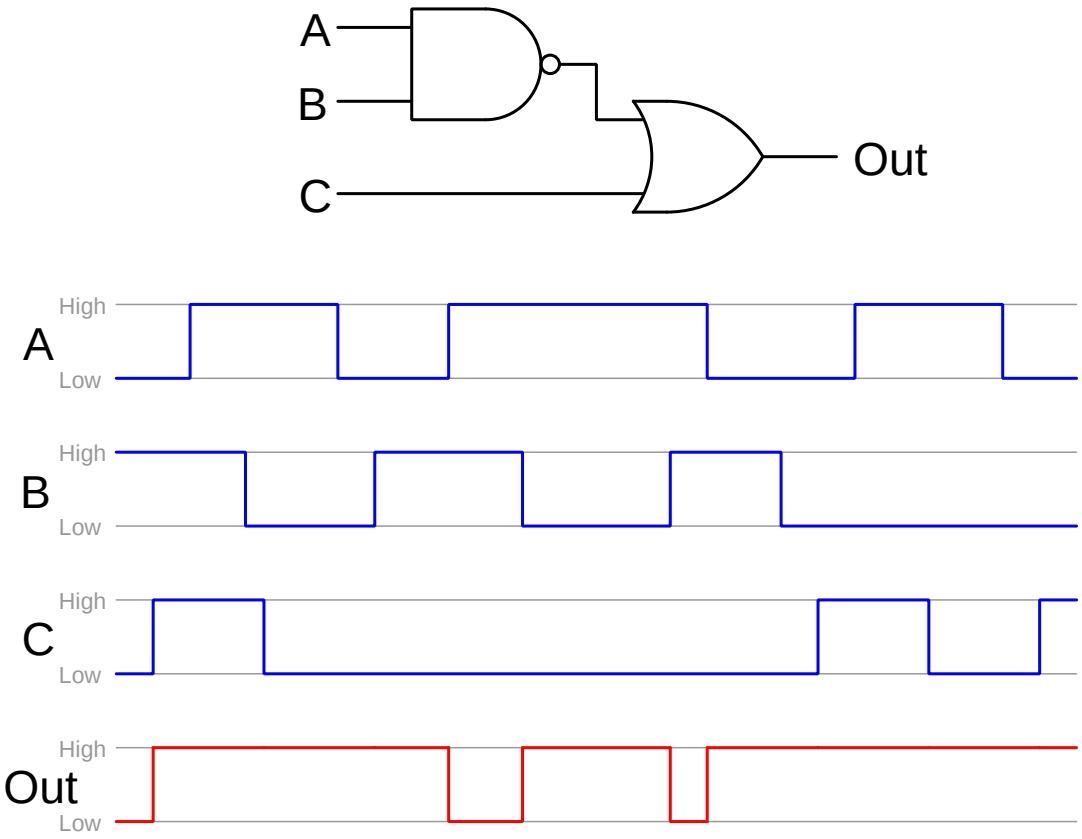
Case Tutorial

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.

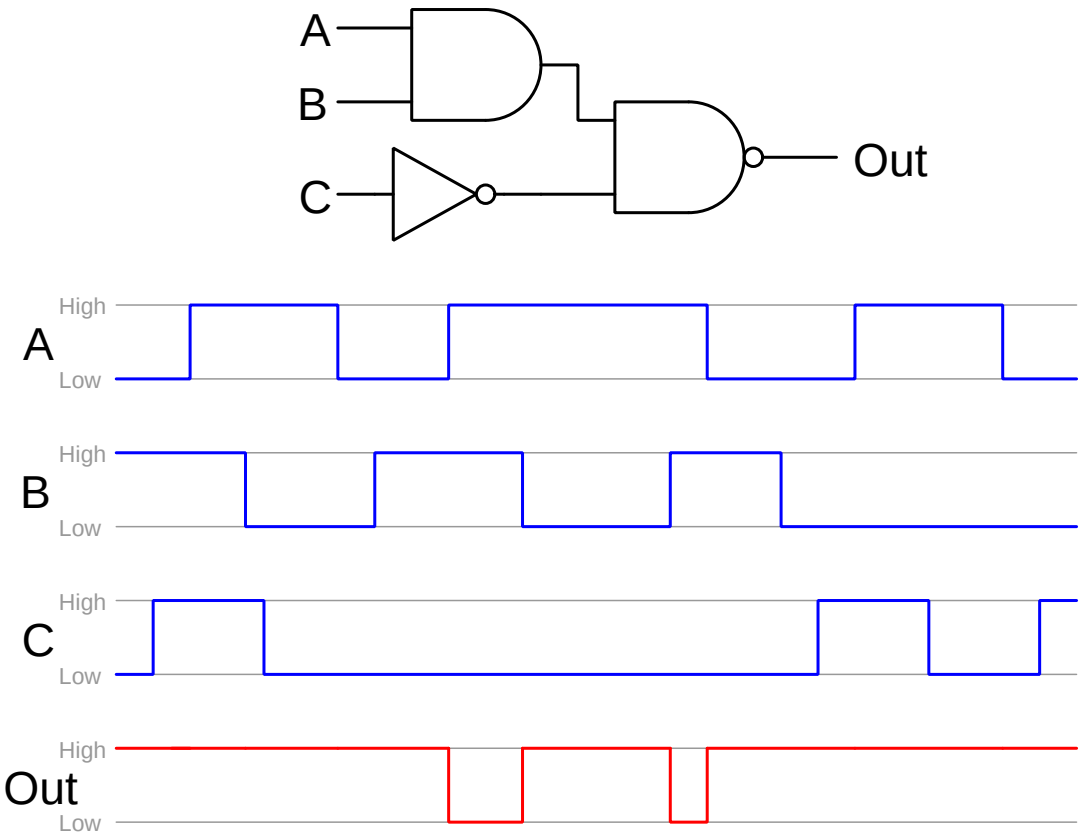
These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

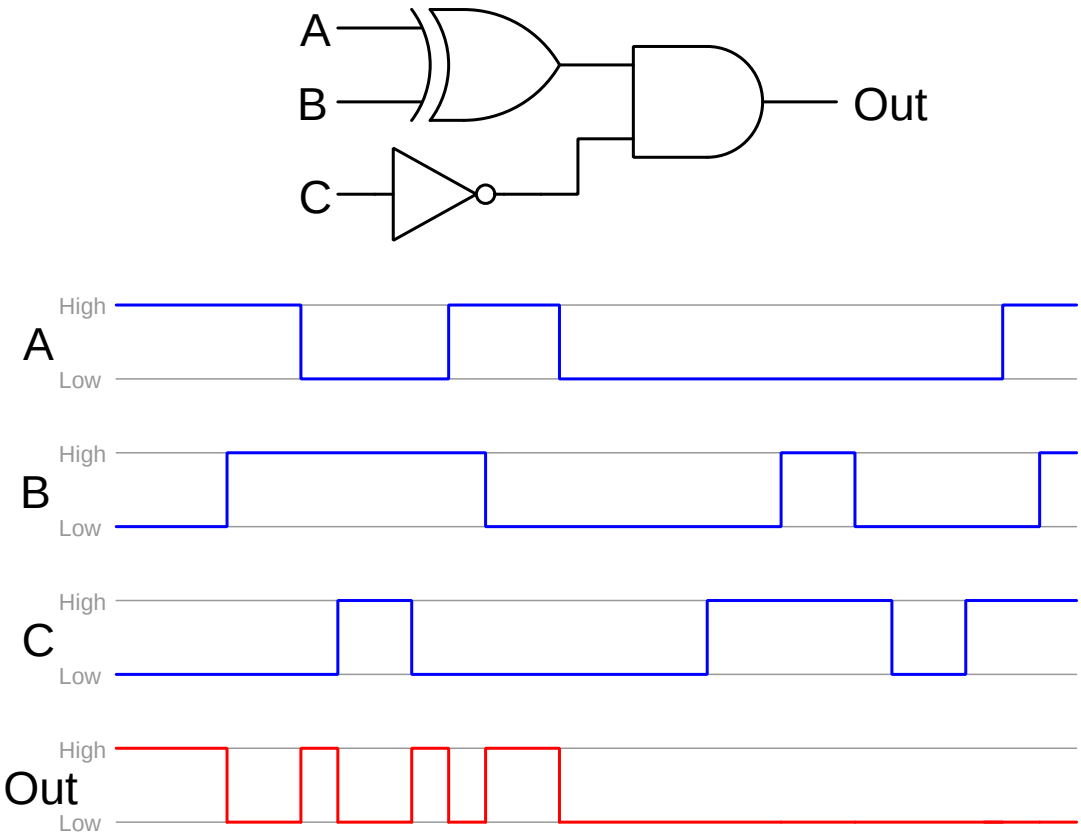
2.1 Example: timing diagrams for combinational gate circuits

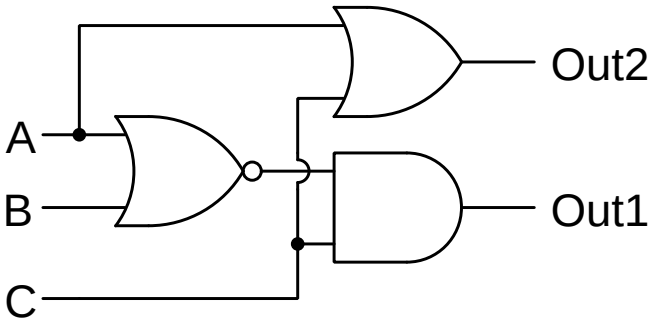
A *timing diagram* shows the "high" and "low" states of logic signals as waveforms in the time domain (i.e. with time being the horizontal axis). These diagrams are essential for analyzing logic circuits with *latching* (memory) capability, but they may also be used to document the behavior of non-latching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

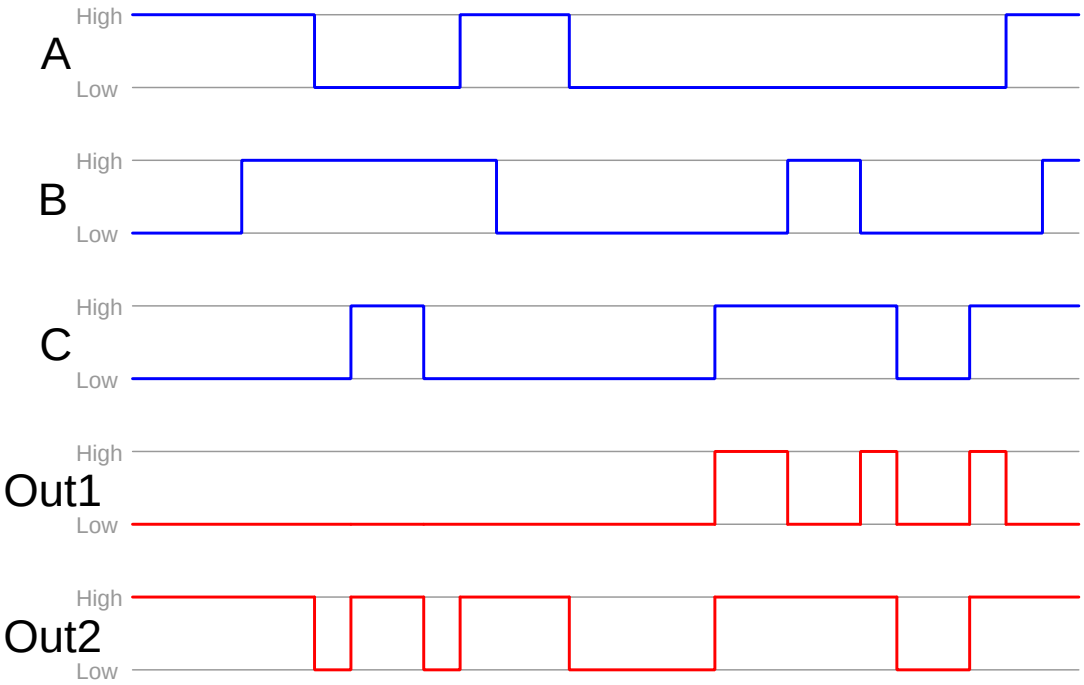


Note: a good way to approach the analysis of this and other circuits where conditions change over time is to make multiple copies of the schematic diagram – one for each different moment in time shown on the timing diagram where there is a unique set of input conditions – and then annotate each of those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


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

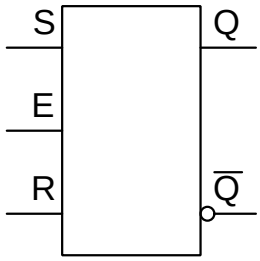

A
B
C
Out
High
A
Low
High
B
Low
High
C
Low
High
Out
Low

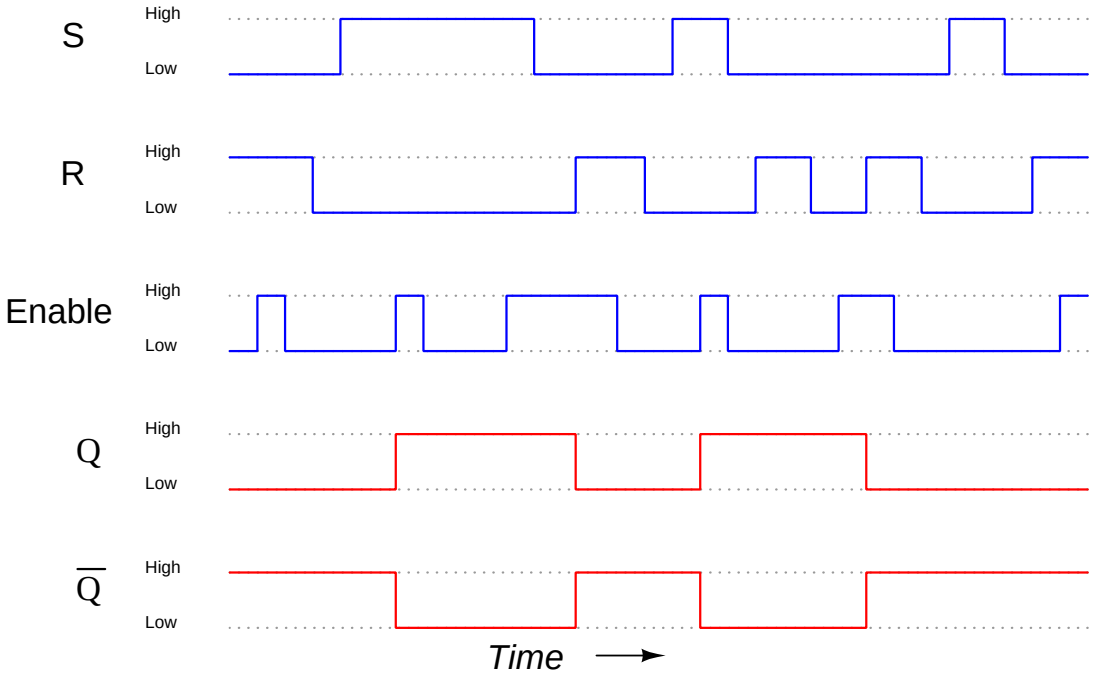

A
B
C
Out2
Out1


High
A
Low
High
B
Low
High
C
Low
High
Out1
Low
High
Out2
Low

2.2 Example: timing diagrams for latches and flip-flops

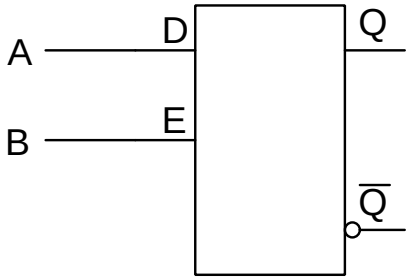
2.2.1 Example: enabled SR latch timing diagram

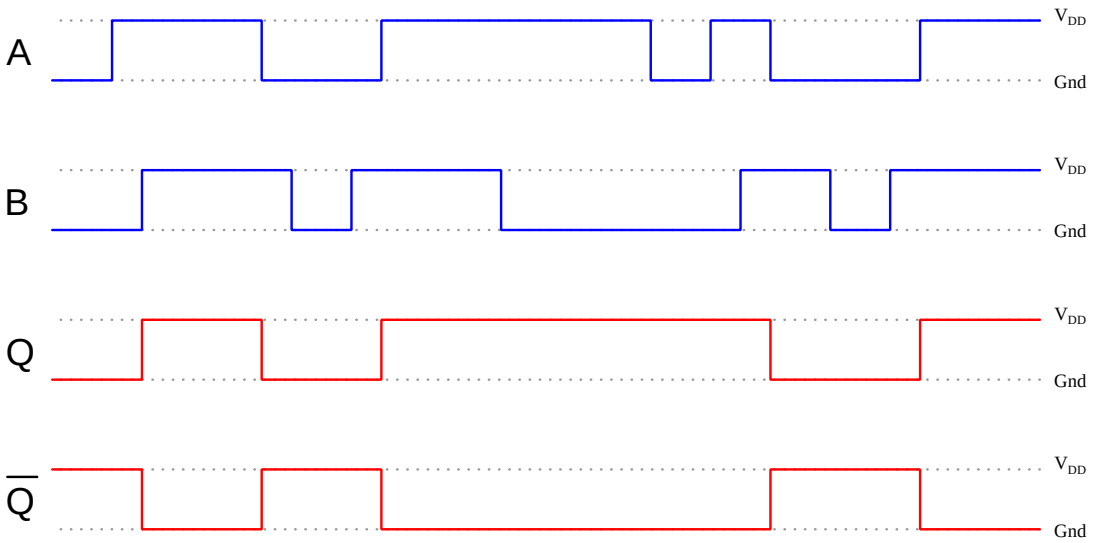




The red-colored output signals assume Q began in a low state and $\overline{Q}$ in a high state.

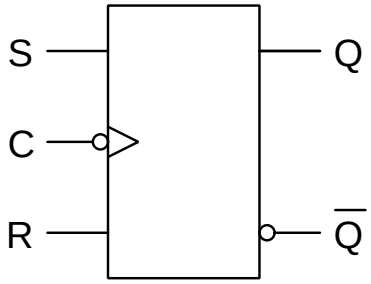
2.2.2 Example: enabled D latch timing diagram

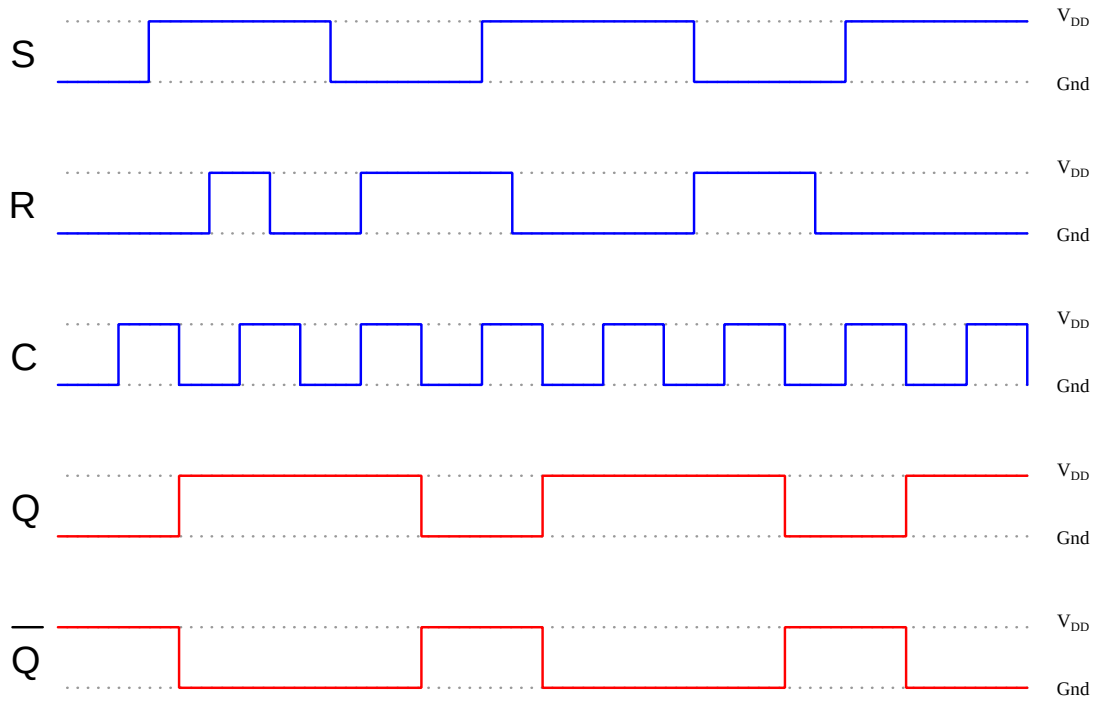




The red-colored output signals assume Q began in a low state and $\overline{Q}$ in a high state.

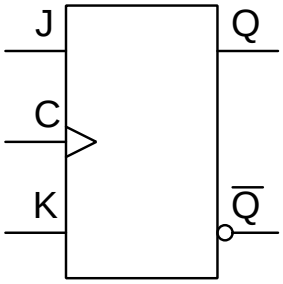
2.2.3 Example: SR flip-flop timing diagram

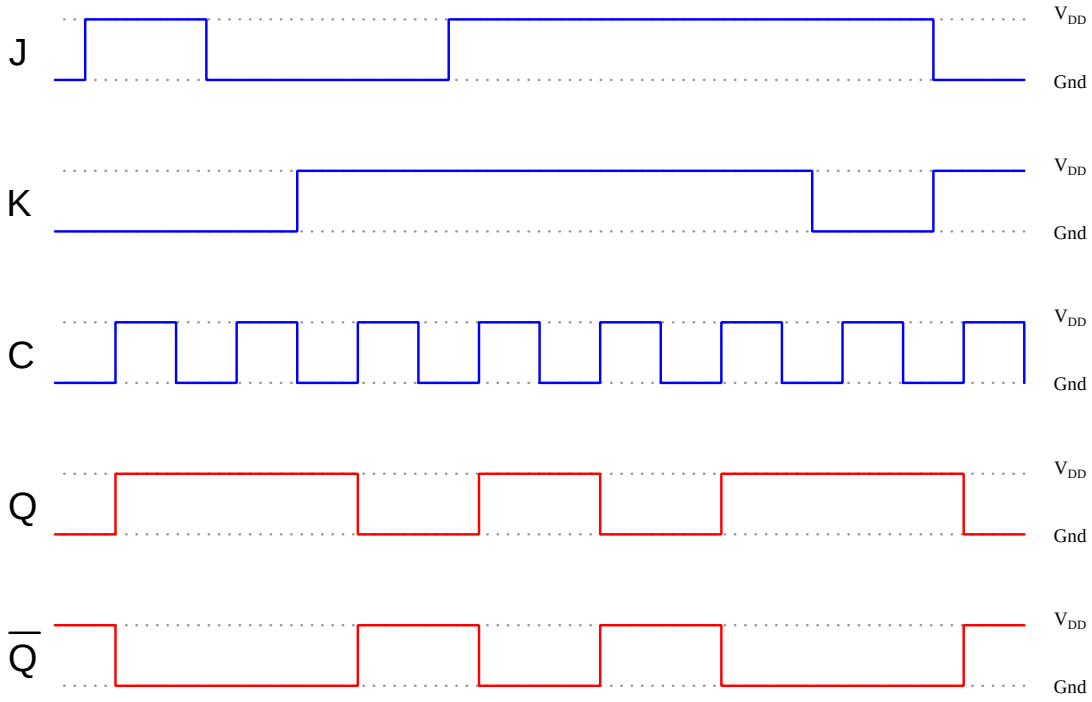




The red-colored output signals assume Q began in a low state and $\overline{Q}$ in a high state.

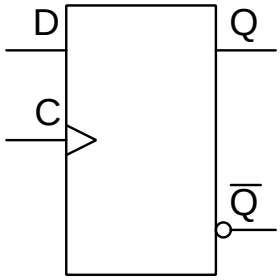
2.2.4 Example: JK flip-flop timing diagram

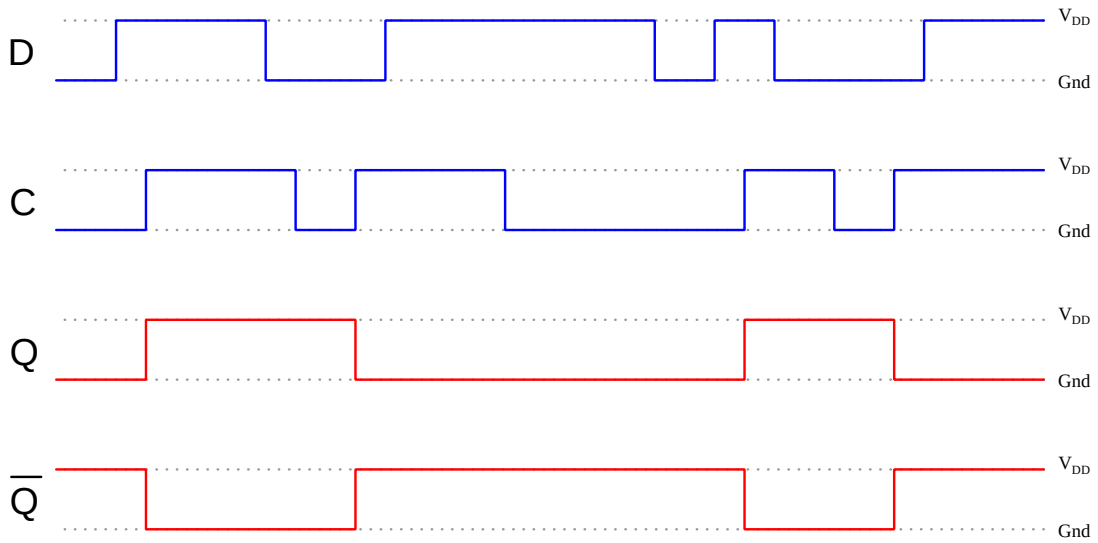




The red-colored output signals assume Q began in a low state and $\overline{Q}$ in a high state.

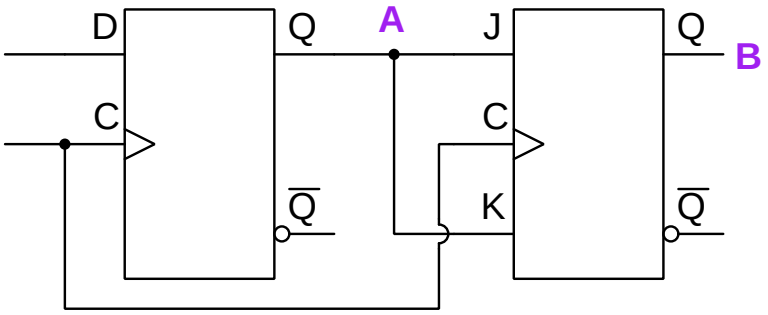
2.2.5 Example: D flip-flop timing diagram

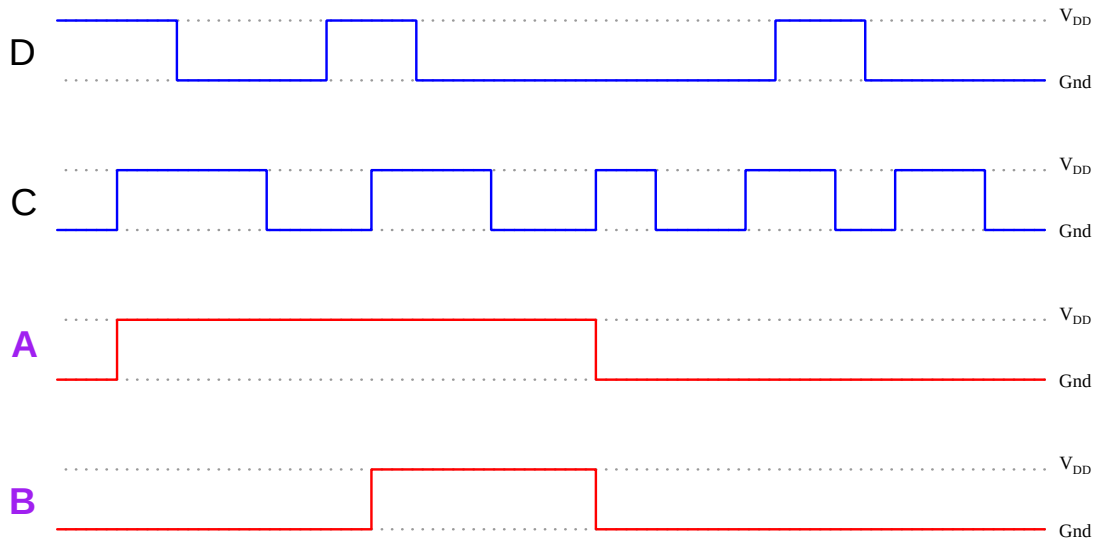




The red-colored output signals assume Q began in a low state and $\overline{Q}$ in a high state.

2.2.6 Example: cascaded flip-flops timing diagram

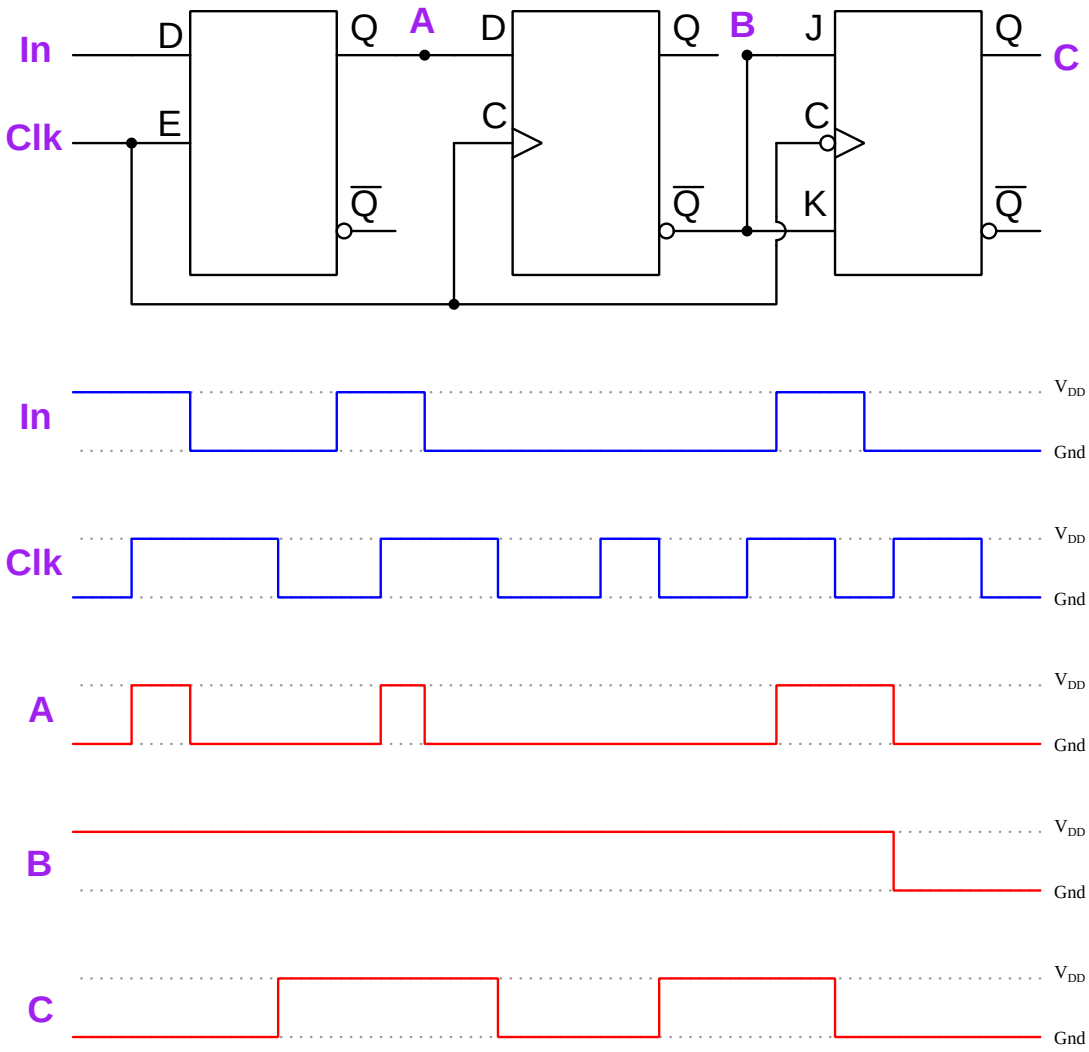




The red-colored output signals assume both Q outputs began in a low state and both $\overline{Q}$ outputs in a high state.

2.2.7 Example: cascaded latch/flip-flops timing diagram

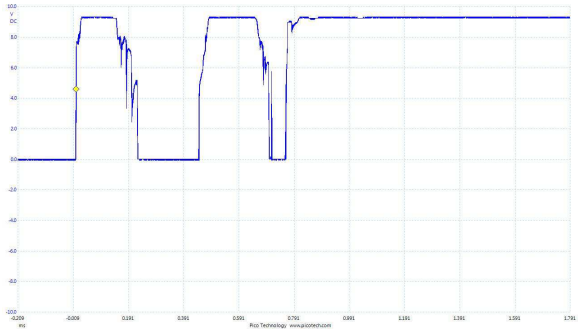
Note that the first element is a D-type *latch* while the second is a D-type *flip-flop*. Also, note that the JK flip-flop is *negative* edge-triggered rather than positive:

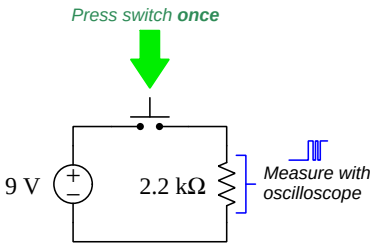


The red-colored output signals assume all Q outputs began in a low state and all $\overline{Q}$ outputs in a high state.

2.3 Switch contact bounce

When mechanical switch contacts open and close, they tend to do so in a "noisy" fashion, making intermittent contact when opening and when closing. We may capture these intermittent contact events using an *oscilloscope* to graphically plot voltage over time:





This particular oscillograph was captured during the time when the pushbutton switch was pressed a single time. What we see the oscilloscope detect during this time is actually *three* distinct closures of the switch's contacts, with jagged rising and falling edges. The reason we see three closures of the switch over a span of less than 1 millisecond is because the metal contact surfaces are literally *bouncing* of of one another as the pushbutton force acts to press them together. This phenonenon is not unlike dropping a ball on a hard floor surface, the ball bouncing several times before coming to rest on that floor.

Switch bounce is a common problem when any mechanical switch contact generates a signal for the input of a digital counter circuit, the purpose being for that counter to increment or decrement *once* for each switch actuation. "Bouncing" switch contacts will "fool" the counter into counting multiple times per switch actuation instead of just once.

Various techniques exist to mitigate switch bounce:

- **Use mercury-wetted switch contacts** – these are special switch contacts housed in a hermetically-sealed glass tube with a small amount of liquid mercury present. This liquid mercury adheres to the metal switch contact faces, providing a mercury "bridge" maintaining continuity between the contact faces when they are separated by very small distances, thereby maintaining contact during the "bouncing" period.

- **Use an electronic switch** – eliminating mechanical switch contacts altogether is a direct way of solving this problem. "Switches" using magnetic-field sensors and transistors as the switching elements may be used to sense the operation of a pushbutton actuator, and in the case of limit switches used to detect machine motion there exist both inductive-style and capacitive-style proximity switches that will do the same task in a bounce-less manner.

- **Connect a capacitor in the circuit** – inserting a small capacitor into the circuit to stabilize the voltage signal is another way to "de-bounce" mechanical switch contacts, preventing those

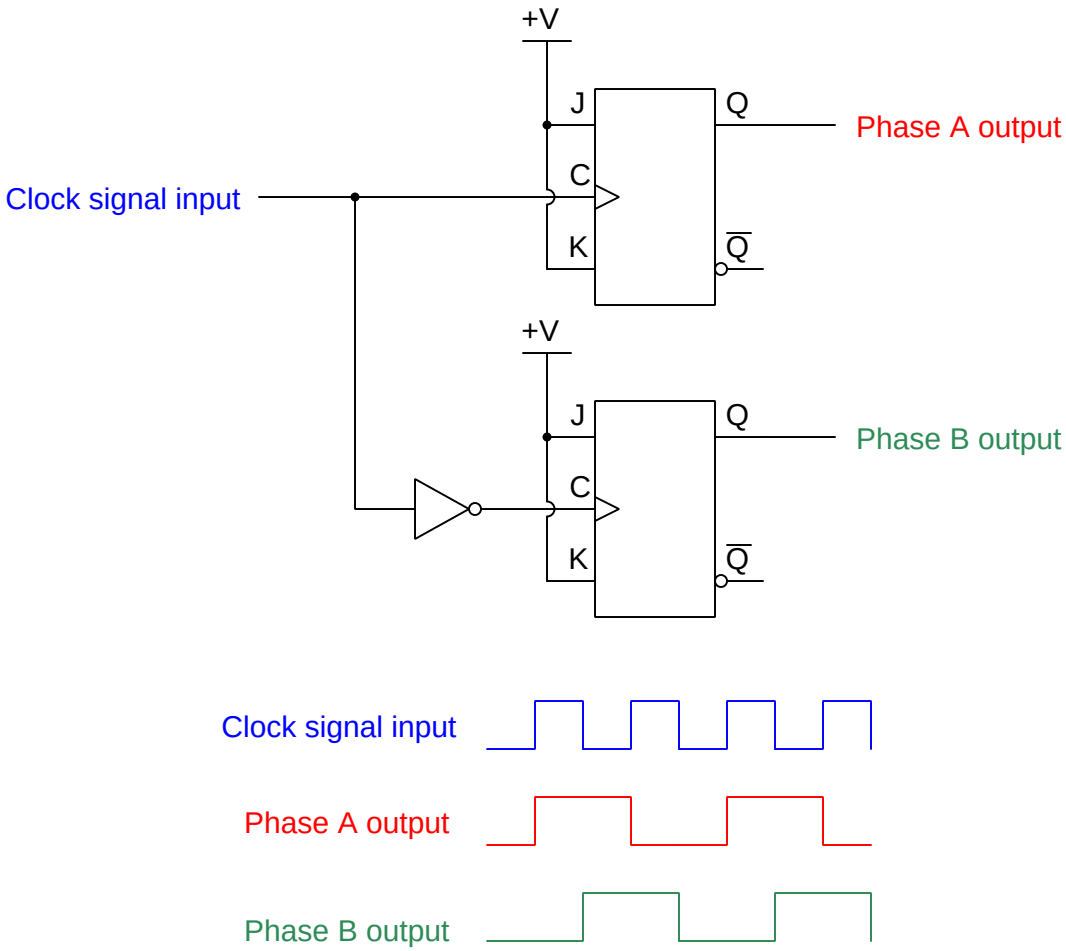
contacts from generating transient voltage pulses of too-short duration. This technique works especially well in digital electronic circuits if the capacitor-stabilized voltage signal is sampled by the input of a *Schmitt-trigger* style of logic gate, this type of gate circuit designed to tolerate voltage levels between valid "high" and valid "low" states.

- **Use a shift register** – a digital *shift register* circuit sampling the switch's signal in its serial input terminal, driven by a clock pulse signal of suitable frequency, will populate that register's bits with successive states of the switch. An AND logic function reading all parallel bits from the register then provides a "de-bounced" signal that will be "high" (1) only if *all* previous states of the switch were also "high" (i.e. only if the switch contacts have remained closed for a certain duration of time established by the clock pulse frequency and the number of parallel bits offered by the shift register).

- **Use software sampling** – if the switch's signal goes to the input of a microcontroller or other similarly programmable digital device, that device may be programmed to perform the same repeated sampling and testing of the switch signal described in the shift register technique.

2.4 Quadrature pulse signal generation

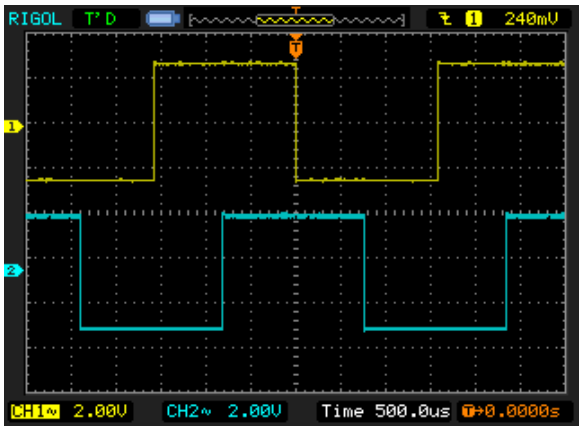
A simple flip-flop based circuit for creating a set of *quadrature* pulse signals is shown below. The word "quadrature" simply means "phase-shifted by one-quarter of a cycle" (i.e. phase-shifted by 90^o). Here we make use of two JK flip-flops to convert a single clock pulse signal into a quadrature signal pair having half the frequency of the original clock:



The key to this circuit's operation is that the upper JK flip-flop works directly from the clock pulse signal while the lower JK flip-flop sees an inverted version of that same clock signal. This means the upper flip-flop toggles with every rising edge of the clock while the lower flip-flop toggles with every falling edge of the clock. It is this distinction in clock pulse edges that makes the two flip-flops toggle at different times. So long as the clock pulse has a 50% duty cycle, the phase shift between the two flip-flop Q outputs will be 90 degrees[1].

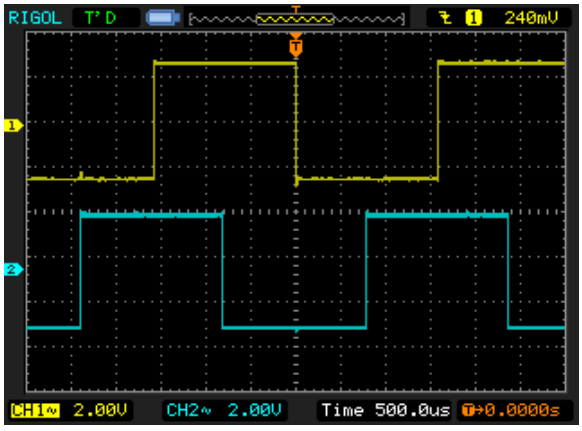
[1] In truth, the phase shift will not be *exactly* 90 degrees even with a perfectly symmetrical clock pulse input due to

An oscilloscope screen-capture shows the results of this circuit at a clock frequency of approximately 630 Hz, with the phase A (channel 1) and phase B (channel 2) outputs measured by the oscilloscope at 315 Hz each.



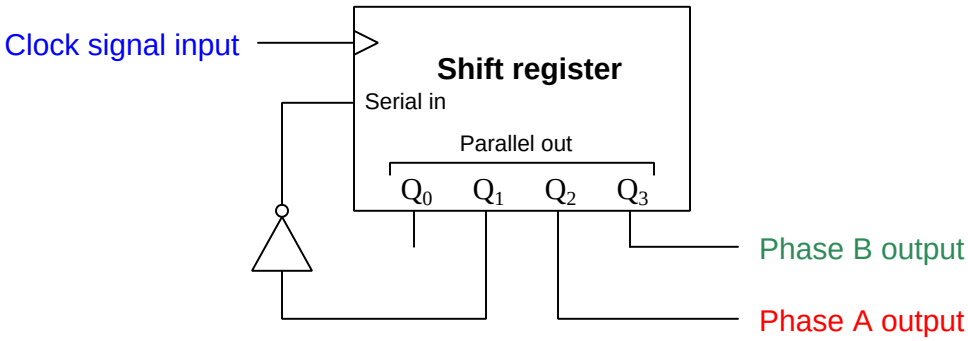
The leading versus lagging phase relationship between phases A and B is more difficult to predict, as this depends not only on the power-up states of both flip-flops (which are random) but also whether the first clock pulse edge after power-up happens to be rising or falling. If both flip-flops happen to start up in their reset states (i.e. Q low) *and* the first clock pulse edge is a rising one, then phase A will lead phase B by 90 degrees. However, if both flip-flops happen to start up in their reset states but the first clock pulse edge is a falling one, phase B will be the first to toggle and it will lead phase A by 90 degrees. Similarly, if we imagine a start-up condition where the upper flip-flop happens to begin in its set state (i.e. Q high) and the lower flip-flop begins in its reset state, and the first clock pulse edge is a rising one, B will once again lead A by 90 degrees.

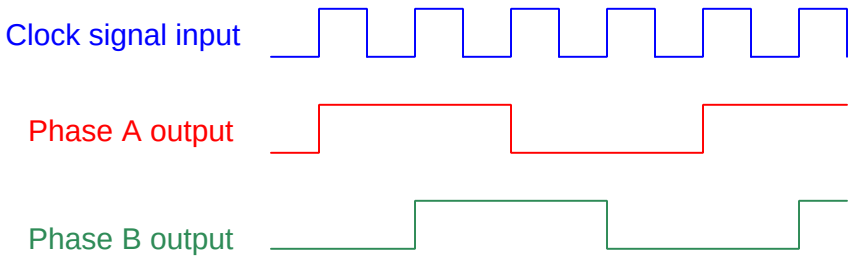
Note the next oscilloscope screen-capture taken on the exact same circuit after powering it up and down several times until the phase relationship happened to be reverse of what it was before:



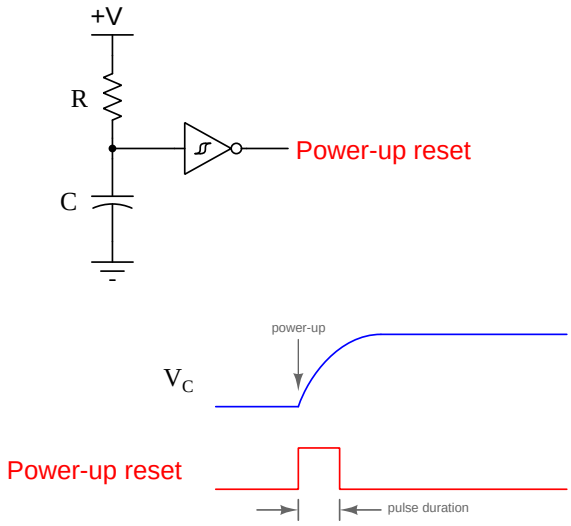
the propagation delay time of the inverter, which makes the lower flip-flop respond *slightly after* the clock pulse falls.

A different technique for generating quadrature pulse signal pairs uses a *shift register* rather than flip-flops. So long as all parallel data output lines on the shift register begin in the low state, the output signals will be one-quarter of the clock's frequency and the phase relationship will always be A leading B:

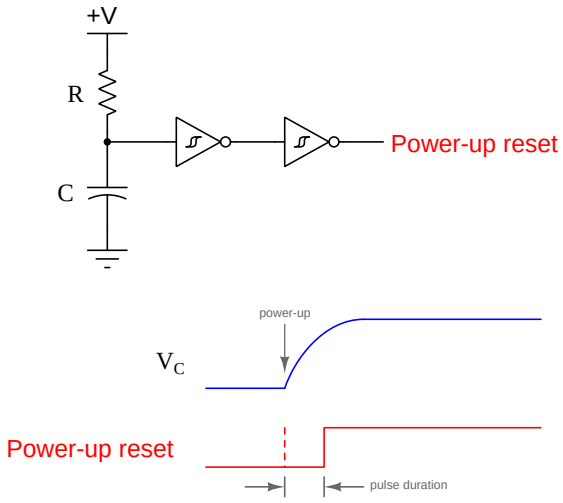




A simple way to ensure the shift register begins with all outputs cleared is to build an RC time-delay network to generate a "reset" pulse upon power-up:



This circuit works for any digital latch, flip-flop, or register having an active-high reset input. If our logic IC has an active-low reset, we may generate the correct reset pulse by simply adding another inverter[2] to the circuit:



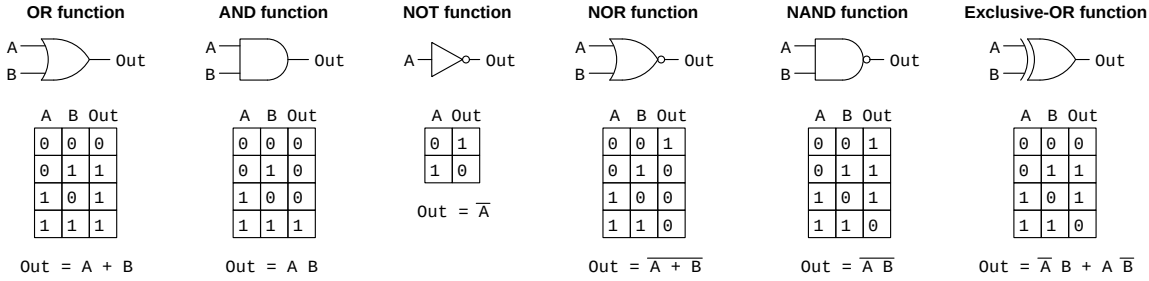
[2]This second inverter need not be a Schmitt-trigger style like the first, but since inverters are almost always found as multiple units within a single package we might as well use one of the others we already have instead of adding another IC just to have access to a normal (non-Schmitt-trigger) inverter gate!

Chapter 3

Tutorial

3.1 Logic functions

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



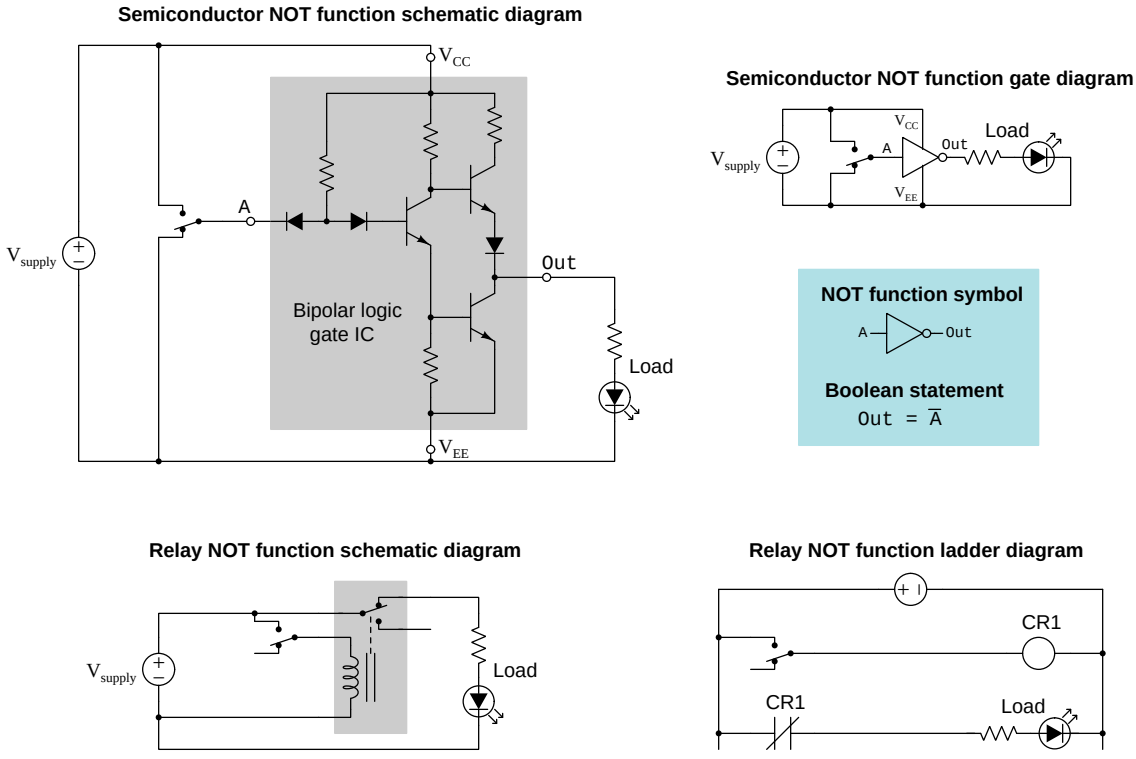
Although the use of arithmetic (e.g. $A + B$ for the OR function, AB for the AND function) may seem strange, it makes sense when you consider the limited values each discrete variable has. If each variable may only be a 0 or a 1, it makes sense, for example, that an AND function whose output is 1 only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

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

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

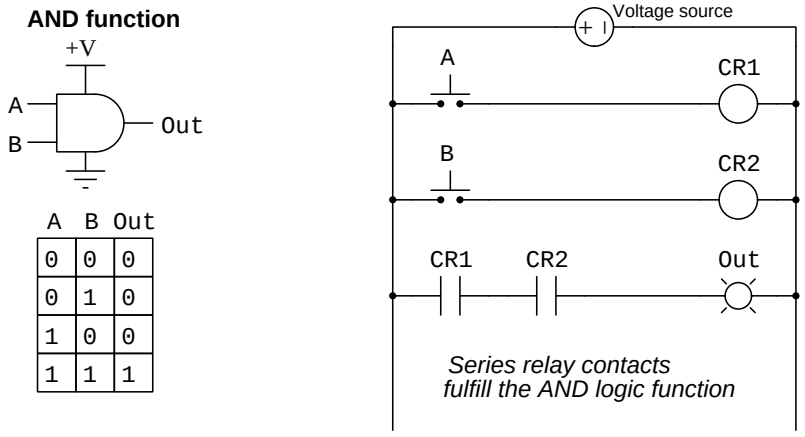
Electrical logic circuits use discrete voltage signals to represent 0 and 1 logical states. Typically, a "high" voltage value (at or near the positive power supply rail voltage with respect to ground) represents 1 and a "low" voltage value (at or near ground potential) represents 0. Logical functions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

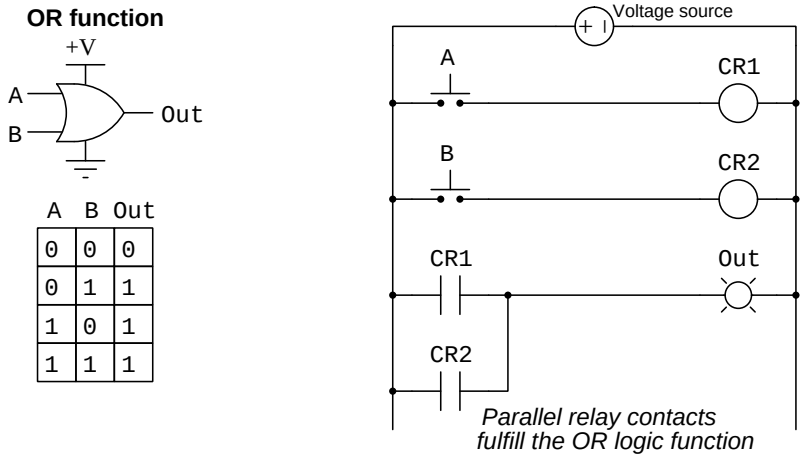
The NOT function, for example, may be constructed using bipolar junction transistors and packaged in an integrated circuit (IC), or alternatively it could manifest as an interconnection of electromechanical relays. Four diagrams below show how the NOT function may be implemented using either solid-state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



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

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



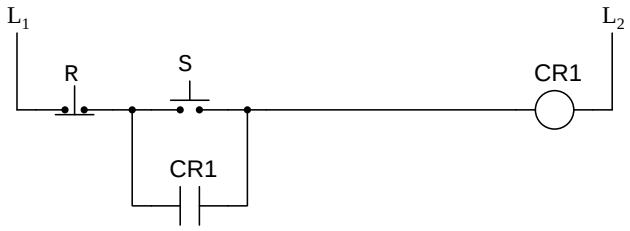


It is important to closely study the conventions of each diagram style, where we find similar or even identical symbols used to represent different things. A small circle, for example, refers to a terminal on an integrated circuit (IC) package, whereas on a gate diagram an identical circle represents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

3.2 Latch circuits

One category of discrete logic circuits with its own unique characteristics is that of *latches*. A "latch" is a logic function designed to retain its last output state for certain input conditions. Like a toggle switch that may be "flipped" into one of two states by a temporary motion and then remain in its last state, a latch circuit does the same at the command of electrical input signals.

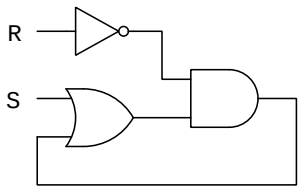
A very simple latch circuit may be constructed using an electromechanical relay, where one of the contacts in the relay is used to route electrical energy to its own coil, thereby *allowing the relay to control itself*. This is typically referred to as a *seal-in* circuit, with an example shown in the following ladder logic diagram:



Note how relay CR1 has a normally-open contact connected in parallel with the "S" ("Set") pushbutton switch. If S is pressed while R is released, the coil will energize, causing its normally-open contact CR1 to close, thereby "sealing in" the energized state of the relay coil when the S switch is released. In other words, momentarily pressing the "S" pushbutton causes the relay to energize and remain energized, which we define as *setting* the latch circuit.

De-energizing relay CR1 requires that the "R" ("Reset") pushbutton be pressed. This normally-closed pushbutton switch will open when pressed, interrupting power to relay coil CR1 and forcing it to de-energize. As relay CR1's coil de-energizes, its normally-open contact returns to its open state, so that when the R pushbutton is released the coil will remain de-energized. In other words, momentarily pressing the "R" pushbutton causes the relay to de-energize and remain that way, which we define as *resetting* the latch circuit.

The equivalent logic gate circuit is just as simple, and the presence of signal feedback[2] is easy to identify as shown in the following diagram:



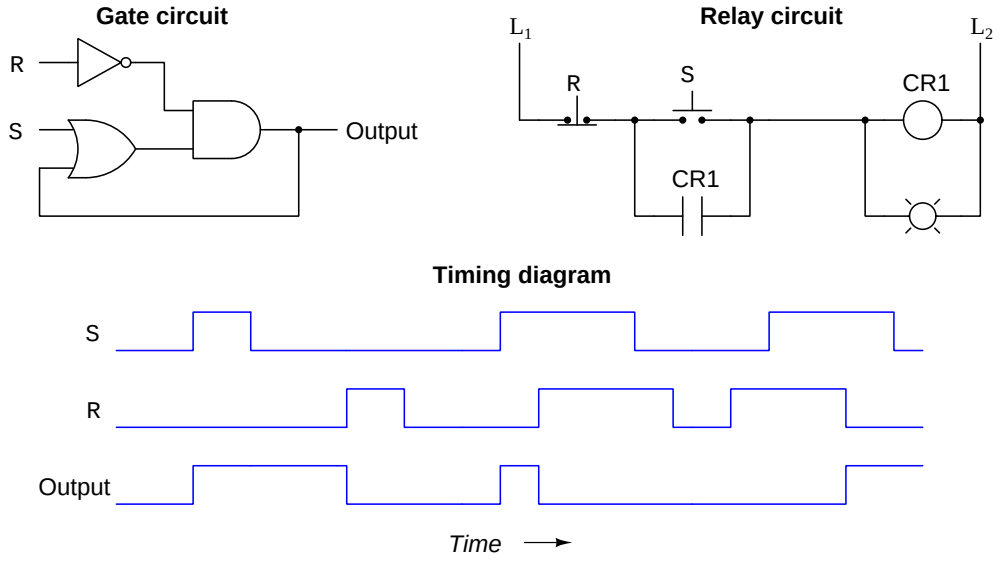
[2]Technically, this is an example of *positive* feedback, which is defined as the tendency to reinforce itself.

Representing the logical function of a latch by means of a truth table is a more complicated task than for combinational circuits where the output state is solely defined by input conditions. Consider the following truth table for the set-reset latch circuits previously shown:

S	**R**	**Output**
0	0	*latch*
0	1	0
1	0	1
1	1	0

As previously demonstrated, the latch will become "set" when the S input is activated and the R input is at rest. We also know the latch will definitely "reset" when the R input is active and the S input rests. If we attempt to simultaneously set and reset this latch (either the relay or the gate circuit version), the reset state takes priority and the circuit resets. What is not so clear is what happens when both S and R inputs are inactive (i.e. at rest). We cannot say for sure what state the latch's output will assume, because that depends on its *previous* state. All we can conclude is that the latch will remain in its last state; i.e. that it will be *latched*.

An alternative means of describing a logic function is to use a *timing diagram* instead of a truth table. This format is similar to an oscillograph, showing discrete logic signals as waveforms alternating between "high" and "low" states as time progresses from left to right. Timing diagrams permit the demonstration of conditions evolving in a particular order, which is a good match for latch circuits which depend on *sequence*.



The timing diagram clearly shows how each latch circuit responds to individual S and R activations, as well as when the S and R activations overlap in time.

3.3 Set-Reset latches

Relay-based latching circuits are usually about as simple as previously shown. Semiconductor gate-based latching circuits, however, are often more complicated for the purpose of providing multiple outputs and more advanced functionalities useful for high-speed digital systems. The two fundamental forms of gate-based latch circuit are shown in the following diagrams, one based on NOR gates and the other on NAND gates:

Set-Reset (SR) latch

S	R	Q	$\overline{Q}$
0	0	*Latch*	
0	1	0	1
1	0	1	0
1	1	0	0

S	R	Q	$\overline{Q}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	*Latch*	

Recall that a NOR gate's output will be forced to a 0 state by a 1 state at *any* input terminal. Conversely, a NAND gate will be forced to a 1 state by a 0 state present at *any* input. In other words, NOR gates are forced by "high" input states while NAND gates are forced by "low" input states. The overriding input logic state for each type of gate defines what is considered "active" for the latch inputs. This means a NOR-based SR latch requires its inputs to go "high" whenever we need to set or reset it, and a NAND-based SR latch requires "low" input signals to set or reset. The technical terminology for this is to say that NOR-based SR latches have *active-high* inputs while NAND-based SR latches have *active-low* inputs.

Note how both circuits have two outputs, Q and $\overline{Q}$. These two outputs are supposed to be complements of each other, but as you can see it is possible to force both outputs to the same state if both S and R inputs activate simultaneously. This is commonly referred to as the *disallowed* or *invalid* state for an SR latch.

Latch circuits are so useful in digital electronics that they are manufactured as complete units, packaged as integrated circuits (ICs) just like logic gates. The following illustrations show common symbols for integrated-circuit SR latches. Note how the active-low nature of the Set and Reset inputs on a NAND-based SR latch is indicated by inverting "bubbles" on the input lines, and complemented $\overline{S}$ and $\overline{R}$ variables:

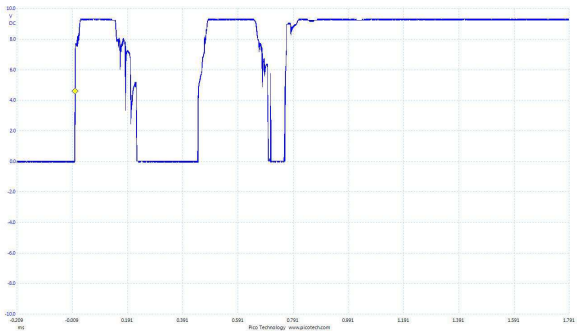
Latches are, or course, designed to fall into one of two states – either *set* or *reset* – when their inputs are inactive. This is why they are classified as *bistable* circuits. An interesting problem arises when a latch circuit transitions from any "disallowed/invalid" state (i.e. where both Q and $\overline{Q}$ are equal) into its "latch" state. For example, if both S and R inputs are active and then they both de-activate simultaneously. Another example is power-up, since both latch outputs are "low" when the power is off, but once power is established those outputs should assume opposite states. In either case, the latch should return to either the set or reset state, but which one?

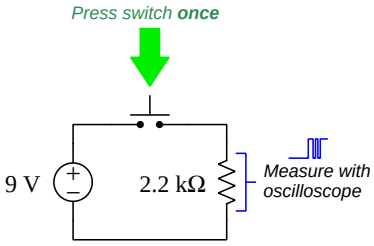
The answer to this question is far from certain. If both NOR or NAND gates are *precisely* matched in terms of switching speed, what will happen is both gates will reverse their output states, which will then cause the two gates to reverse their output states again, and the cycle will repeat until one or both of the inputs are externally activated again. In practice, endless cycling is impossible because two semiconductor logic gates will never switch at *exactly* the same speed. Whichever gate is inherently faster will "win" the race and cause the latch to return to that gate's winning state. However, just because sustained oscillations are practically impossible does not mean that a latch circuit cannot "chatter" for one or two cycles following simultaneous de-activation of its inputs. In fact, *any* bistable circuit externally driven into a mutually incompatible (i.e. "disallowed/invalid") condition will find itself "racing" toward a new stable condition when that driving condition ceases. Such *race conditions* cause erratic behavior and are typically difficult to diagnose[3].

The best solution to race conditions is *avoidance*: designing the system so that disallowed or invalid states never occur, and a race never commences. For integrated circuit SR latches, a good strategy is to design the input circuitry so that the S and R inputs can never be simultaneously activated. However, this still doesn't prevent instability following power-up. A more practical solution is to intentionally place one of the latch's racing elements at a disadvantage, so that it is forced to switch slower than the other. This biases the race so that the outcome is always stable and predictable no matter the initial condition causing the "disallowed/invalid" state. Another solution is to redesign the latch circuit itself so that no disallowed/invalid condition exists at all, such as the OR-AND-NOT latch design discussed earlier: whether returning from a condition of simultaneous S and R activation or just powering up, this latch circuit design always returns to the reset state and cannot cycle.

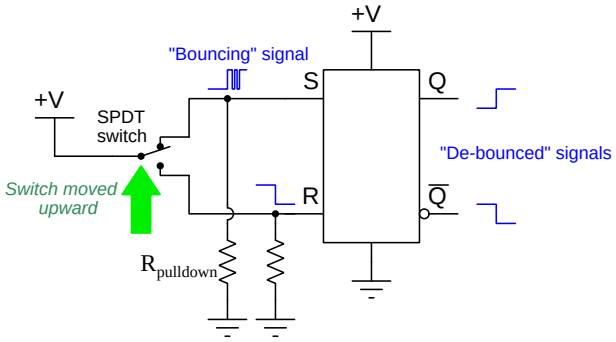
[3] If the bistable circuit in question uses electromechanical relays rather than semiconductor gates, identification of a race condition is aided by the fact that the "chattering" is often audible. However, the usefulness of this audible indicator is limited, as it typically doesn't indicate the root cause of the race.

A popular and practical application for semiconductor SR latches is to *de-bounce* discrete signals from mechanical switch contacts. Switch "bounce" is the phenomenon where a mechanical switch contact experiences intermittent contact while opening or closing. The term "bounce" relates very well to the action of metallic contact faces during the process of closing, where the elasticity of the moving switch pole and the mass of the contact face result in the moving metal contact literally rebounding off of the stationary contact rather than gently coming to rest. An example of switch contact bounce is shown in the following oscilloscope trace capture (left), with the experimental circuit shown on the right:





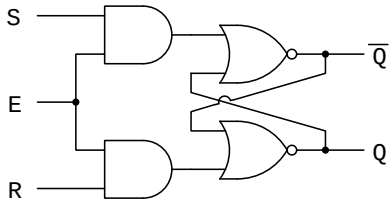
As you can see on the oscillograph of one particular test, the switch contacts actually made contact *three times* and broke contact *twice* before settling to the desired "on" state, all in less than one millisecond. Such "bouncing" will create problems if the discrete signal is sent to the input of a high-speed counting circuit or something similar. What we need in such applications is a "clean" off-to-on signal transition as the imperfect pushbutton switch is pressed. If the application in question permits the use of a SPDT (Single Pole Double Throw) pushbutton switch instead, we may connect one to the inputs of an SR latch and achieve the desired "de-bouncing" of the contact status:



3.4 SR enabled latches and flip-flops

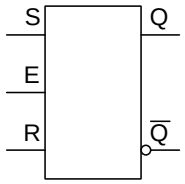
A design variation on the basic SR latch is to add a third input called an *enable*. The concept of an "enable" input is simple to explain: when the enable input is active, the circuit responds normally to the other inputs; when the enable is inactive, the circuit *ignores* all other inputs. An example of an enabled SR latch based on NOR gates is shown in the following illustration, along with its truth table and standard symbol:

Enabled SR latch internal schematic



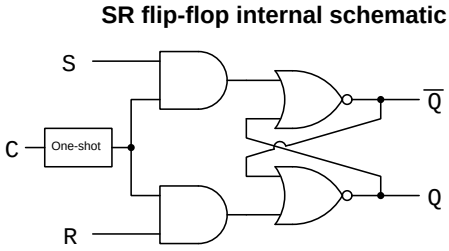
E	S	R	Q	$\overline{Q}$
0	0	0	*Latch*	
0	0	1	*Latch*	
0	1	0	*Latch*	
0	1	1	*Latch*	
1	0	0	*Latch*	
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

Enabled SR latch symbol

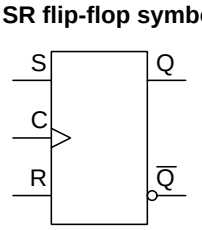


By passing the S and R input signals through AND gates prior to the cross-connected NOR gates, and also connecting the new E (enable) input to both AND gates, the enable forces the latch into its latching state whenever inactive.

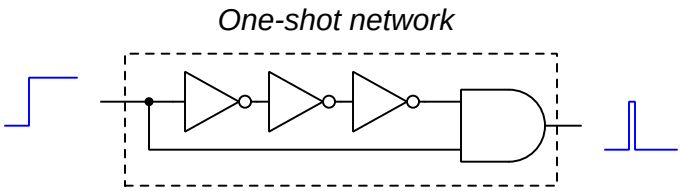
A variation on this theme adds one more component called a *one-shot*[4], also known as a *monostable multivibrator*[5]. A "one-shot" has one discrete input and one discrete output, its output *momentarily* activating whenever the input transitions from the inactive to active state. Adding a one-shot to the enable (E) input of the SR latch makes the latch responsive to its S and R inputs only during that brief moment in time when that signal *transitions* from inactive to active. In honor of this, the enabling input is now called the *clock* (C) because the latch responds only on the rising edge of that signal's pulse, and the circuit is now called a *flip-flop* rather than a latch. The following illustration shows this modification:



C	S	R	Q	$\overline{Q}$
X	0	0	*Latch*	
X	0	1	*Latch*	
X	1	0	*Latch*	
X	1	1	*Latch*	
⎍	0	0	*Latch*	
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	0	0



One method of creating a one-shot for this type of application is to use a set of logic gates such as this:



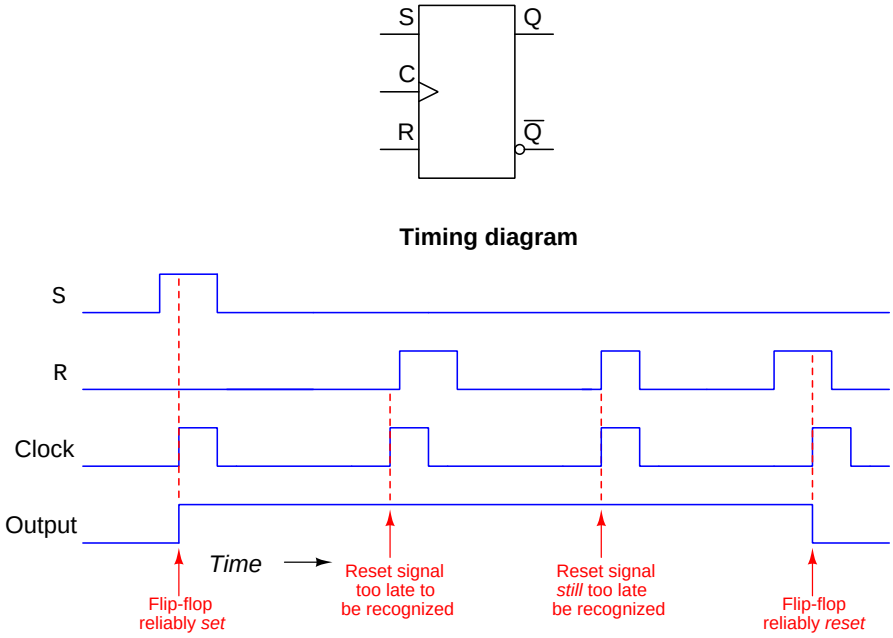
The AND gate, of course, outputs a high signal only with both of its inputs are high. This happens for a very brief moment of time as the input transitions from low to high, as the input signal arrives instantaneously at the lower AND gate input while its inverted counterpart arrives at the upper AND gate input only after the total propagation delay of the three inverter gates. If a longer pulse width is required, the circuit may be modified to have more inverter gates (always an odd number of them) in order for the AND gate's upper input to experience greater propagation delay.

[4] The term "one-shot" is apt, as this device outputs a single pulse ("shot") every time its input activates.

[5] A "multivibrator" is any digital circuit employing feedback, of which latches and flip-flops are two types.

An interesting consideration for flip-flops is that their input terminal (e.g. S, R) states must be established slightly in *advance* of the clock signal's active edge in order to be reliably "seen" by the flip-flop. The amount of advance time required for reliable operation is called the *set-up time*, and its necessity is a product of *propagation delay* in the logic gates comprising the flip-flop, especially the circuitry within the one-shot unit.

While it should be obvious that any change-of-state on an input line will be missed if that change occurs *after* the clock pulse, it may come as a surprise to learn such a signal change could also be missed if it occurs *simultaneous* with the other input state changes. The following timing diagram for an SR flip-flop demonstrates these facts, beginning with the flip-flop in the reset state:



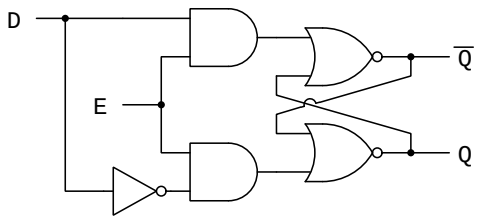
On the first clock pulse the flip-flop reliably toggles to its *set* state because the active-high input signal on the S input line was already established well in advance of the clock pulse's rising edge. The second clock pulse arrives to the flip-flop at a time just *before* the R input becomes active, and so (obviously) no reset occurs at that time. However, on the third clock pulse when that pulse arrives at the flip-flop *simultaneous* with the R's activation, still we see that no reset occurs. What is missing from this third clock pulse scenario is adequate *set-up time* between the R input activation and the clock pulse's edge. The fourth clock pulse finds success, as the activated R input state occurs far enough in advance of the clock pulse to be recognized.

Some flip-flops have an additional criterion for reliable operation called *hold time*, referring to a certain minimum amount of time that the input lines must hold in stable states *after* the clock pulse's edge occurs. For example, a flip-flop may not respond to its *last* input state(s) if those states switch just as the clock pulse's edge arrives.

3.5 D-type latches and flip-flops

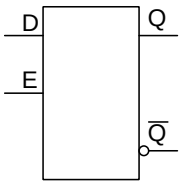
Another variation on this theme is consolidation of Set and Reset inputs into a single input labeled "D" for *data*. Unlike its SR cousin, the D-type multivibrator only comes in enabled (often called *transparent*) or flip-flop versions. The following schematics show D latch and flip-flop circuits based on NOR gates:

D latch internal schematic

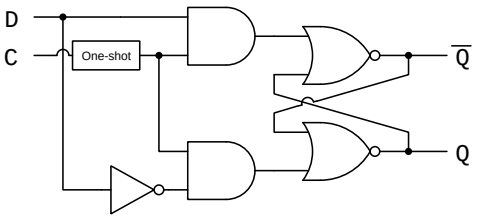


E	D	Q	$\overline{Q}$
0	0	*Latch*	
0	1	*Latch*	
1	0	0	1
1	1	1	0

D latch symbol

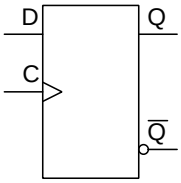


D flip-flop internal schematic



C	D	Q	$\overline{Q}$
×	0	*Latch*	
×	1	*Latch*	
⌐	0	0	1
⌐	1	1	0

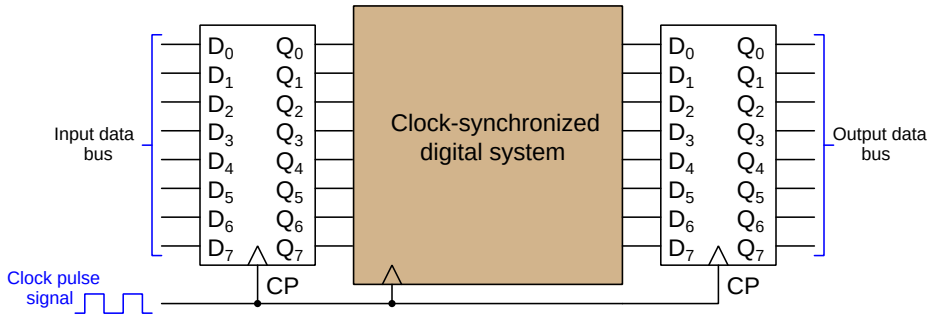
D flip-flop symbol



These circuits are simpler than SR latches and SR flip-flops: the Q output simply mimics the D input when enabled. For the D latch, this enabled time lasts as long as the E input remains active; for the D flip-flop, this enabled time is only during the rising[6] edge of the clock pulse.

[6] Like all flip-flops, D-type flip-flops are also available in *negative* edge-triggered versions.

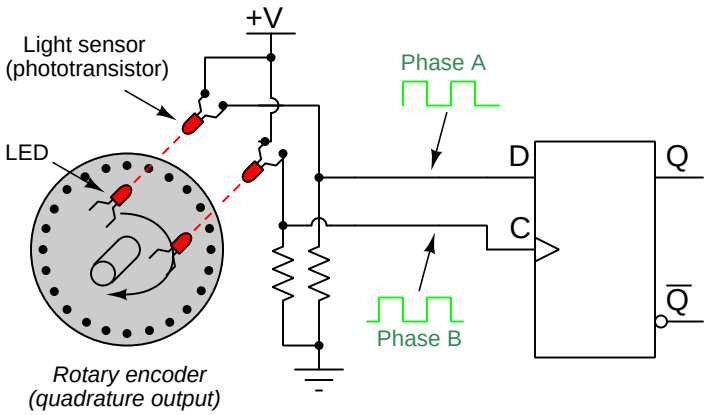
D-type latches and D-type flip-flops find extensive use in digital systems as "gateways" for data throughput. A D latch "transparently" passes data from D to Q when enabled, and blocks the passage of data at all other times. A D flip-flop serves a similar purpose by "synchronizing" incoming data with the clock pulse sequencing all other functions in the digital system, as shown in the following diagram where two *octal*[7] D flip-flops control the flow of data into and out of a digital system:

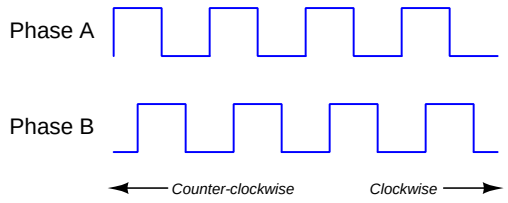


When multiple latches or flip-flops are ganged together like this to manage the communication of multi-bit digital *words*, the circuits are often referred to as *registers*. The group of conductors conveying the multi-bit digital signals is commonly referred to as a *bus*. Collections of logic gates and other digital circuit elements which have their input and/or output signals gated through registers at the behest of a clock signal are often referred to as *registered logic*.

[7] This is an example of a *medium-scale* integrated circuit where eight flip-flops are built on the same silicon substrate. Note the absence of $\overline{Q}$ outputs, which would be superfluous in this application.

Another useful application of a D-type flip-flop is to detect direction of motion for a pair of *quadrature* signals. The term "quadrature[8]" simply means two signals 90^o out of phase with each other. An example of a quadrature digital signal is a pair of square-wave pulses generated by a *rotary encoder* used to sense mechanical rotation, the signals emanating from a pair of light sources and offset photodetectors sensing light passed through a slotted disk: as the shaft turns the disk, the photodetectors receive pulses of light, causing them to periodically conduct electricity. Their physical offset means the two pulse waveforms will always be out of phase with each other, as shown in the following timing diagram:





Direction of encoder rotation may be discerned from these quadrature pulse signals by sensing which one leads and which one lags. As the timing diagram reveals, A leads B when the shaft turns clockwise, but B leads A when the shaft turns counter-clockwise. If phase A connects to the D input and phase B connects to the clock input, the flip-flop will "set" when the shaft turns clockwise[9] and "reset" when the shaft turns counter-clockwise[10].

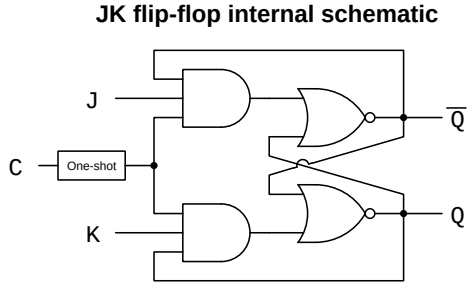
[8] Literally, one-*fourth* ("quad") of a cycle out of phase.

[9] In the clockwise direction, phase A leads phase B. This means A will already be "high" when B rises from "low" to "high", sending a "high" signal to the Q output with every clock pulse.

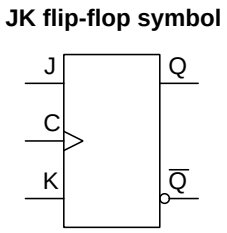
[10] In the counter-clockwise direction, phase A lags phase B. This means A will still be "low" at the time of B's rising edge, sending a "low" signal to the Q output with every clock pulse.

3.6 JK flip-flops

Our final latching circuit, which only exists in flip-flop form, and which gives full meaning to the phrase "flip-flop[11]", is the *JK flip-flop*. An example of a JK flip-flop using NOR gates is shown in the following illustration, along with its truth table and standard symbol:

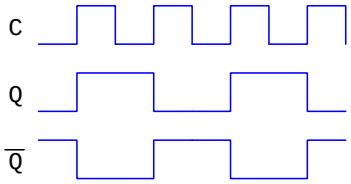


C	J	K	Q	$\overline{Q}$
X	0	0	*Latch*	
X	0	1	*Latch*	
X	1	0	*Latch*	
X	1	1	*Latch*	
⎍	0	0	*Latch*	
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	*Toggle*	



JK flip-flops provide all the valid functions of an SR flip-flop, plus one more[12]: the ability to *toggle* between its two valid states at every clock pulse when both J and K inputs are active. In this "toggle" mode, the outputs of a JK flip-flop oscillate at one-half the frequency[13] of the clock pulse, as shown in the following timing diagram:

Toggling behavior of a JK flip-flop with positive edge-triggering



Note how the Q and $\overline{Q}$ outputs only change state during the *rising* edge of the clock pulse signal. This is called *positive edge-triggering*, because all action takes place at the edge of the clock signal as it transitions in a positive (upward) direction. Negative edge-triggered flip-flops also exist, the only difference in symbology being a "bubble" at the clock input terminal. Internally, the difference is wrought by an inverter (NOT gate) added before the one-shot.

[11] In the "toggle mode" a JK flip-flop's outputs literally "flip" and "flop" between its two valid states at each clock pulse.

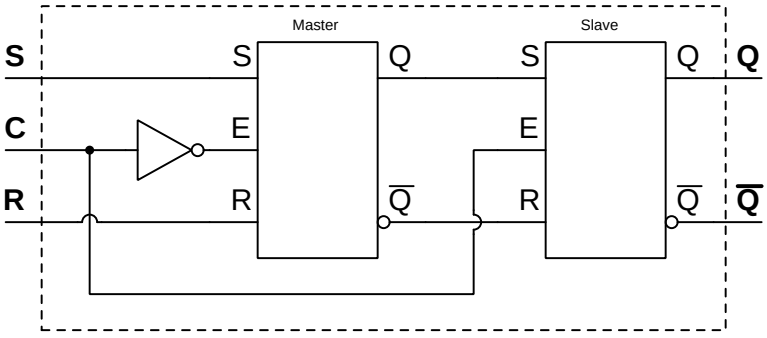
[12] An SR flip-flop, by contrast, is fairly useless when both S and R inputs are active. At every clock pulse the SR flip-flop gets forced into its "disallowed/invalid" state which means it becomes unstable after the clock pulse passes.

[13] This frequency-dividing behavior is more useful than one might first think. A popular application is to cascade multiple JK flip-flops so that the output of one controls the toggling of the next, so that the output of each stage is half the frequency of the one before. If you track these toggling outputs over time, you find that they constitute the individual bits of a *binary number* counting in sequence with each clock pulse! This is how binary counting circuits are built.

3.7 Master-slave flip-flops

Earlier we saw how an enabled latch could be converted into a flip-flop by the addition of a one-shot circuit, producing a very brief pulse at either the rising or falling edge of the clock signal. An alternative to this time-delayed solution is to form a flip-flop circuit from *two* enabled latch circuits, those latches enabled during opposite states of the clock signal. This technique is known as *master-slave*, and is shown here using two SR enabled latches:

Master-slave SR flip-flop

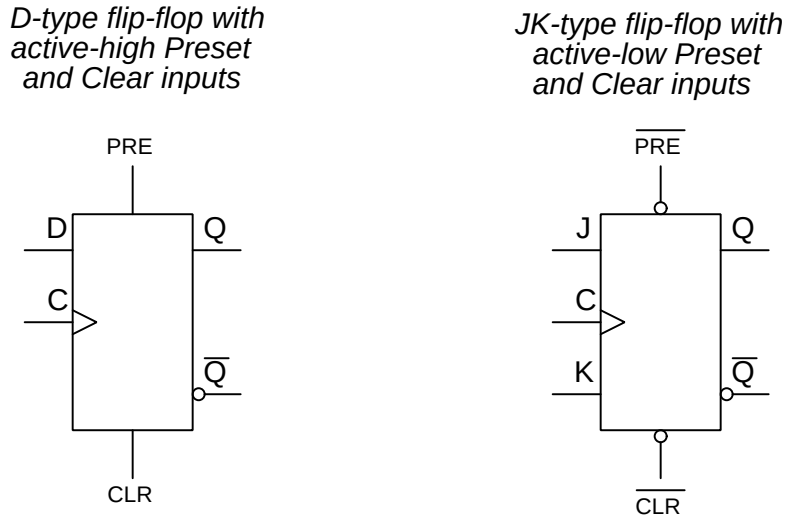


Rather than using a time-delay network (i.e. one-shot) to produce a very brief "enable" pulse signal for the enabled latch to act upon, the circuit shown above relys on two oppositely-enabled latches to ensure that the Set and Reset data only ever act upon the final Q and $\overline{Q}$ outputs when the clock signal transitions from low to high. When the clock signal is low, the master SR latch is enabled but the slave isn't, allowing Set and Reset states to affect only the *internal* Q and $\overline{Q}$ states. Those internal Q and $\overline{Q}$ states, however, will pass along to the final Q and $\overline{Q}$ output lines when the clock switches from a low state to a high state. At no time and during no clock signal state will the external input states pass transparently to the external output lines. The "hand-off" from the master latch to the slave latch only occurs at the clock pulse's rising edge.

A disadvantage of this design compared to the one-shot-enabled concept is that with master-slave the master latch is sensitive to all changes in input states while the clock pulse keeps it enabled. This means the presense of noise on the inputs may cause the master latch to switch its output states, and if that noise occurs close enough to the active edge of the clock pulse, those erroneous master states may affect the slave latch's output states when it becomes enabled. By contrast, a flip-flop circuit triggered by a one-shot will completely ignore all input state-changes occurring prior to the one-shot's enabling pulse and therefore be less susceptible to unstable input states prior to the clock pulse. Another way to characterize this difference is to say that the minimum *set-up time* requirement may be longer for a master-slave flip-flop than for a one-shot-triggered flip-flop.

3.8 Preset and clear inputs

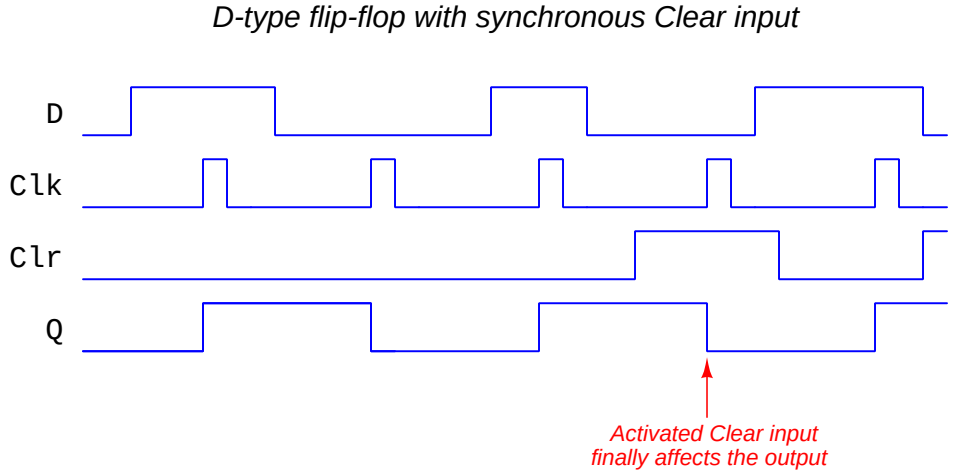
Many flip-flop integrated circuits provide one or more additional inputs designed to force the output lines to particular states, overriding the other input(s). These additional inputs are called *Preset* and *Clear*, the purpose of the Preset input being to force the flip-flop to a "set" state ($Q = 1$ and $\overline{Q} = 0$) and the purpose of the Clear input being to force the flip-flop to a "reset" state ($Q = 0$ and $\overline{Q} = 1$). Below we see schematic diagram symbols of a D-type flip-flop that happens to have active-high Preset and Clear inputs, as well as a JK flip-flop that happens to have active-low Preset and Clear inputs:



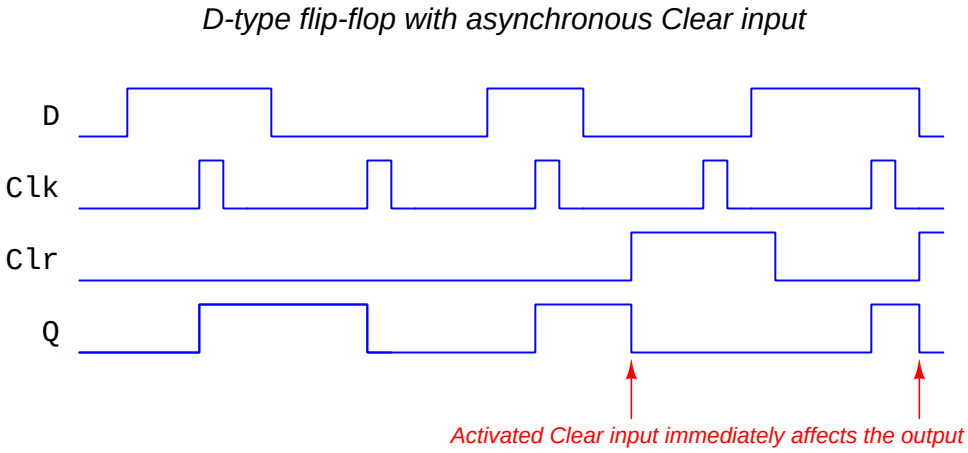
In the case of the D flip-flop, an active Preset or Clear input will override the state of the D input. In the case of the JK flip-flop, an active Preset or Clear input will override both J and K inputs. Preset and Clear inputs are particularly useful when we build complex latching logic circuits with multiple latches and/or flip-flops connected together, where we may use these "extra" inputs to simultaneously set or reset all of them at once. This is useful, for example, when constructing digital counter circuits and we need a way to clear the counter to a "zero" condition which requires placing all flip-flops in a reset condition with a single control signal.

In addition to active-high versus active-low varieties for Preset and Clear inputs, another important distinction is *synchronous* versus *asynchronous* Preset and Clear inputs. The term "synchronous" refers to events happening at the same time, and so a *synchronous* Preset or Clear input will not have any effect until the clock pulse arrives – i.e. its effect is always synchronized with the clock signal. In contrast, an *asynchronous* Preset or Clear input effects the Q and $\overline{Q}$ outputs *immediately* without waiting for a clock pulse. In other words, asynchronous Preset or Clear inputs function like the S and R inputs on a plain (non-enabled) SR latch, affecting the output states immediately when activated. One cannot tell whether Preset and/or Clear inputs are synchronous or asynchronous from the schematic symbols – only the datasheet for that particular integrated circuit will show you!

To better understand the behavior of synchronous versus asynchronous inputs, let's examine two timing diagrams for D-type flip-flops equipped with Clear inputs, one of them synchronous and the other asynchronous, both with positive-edge-triggered clock inputs. First, the synchronous-Clear D-type flip-flop:



Next, the asynchronous-Clear flip flop responding to the exact same sequence of input states:



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 Logic families

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

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

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

The early years of digital logic circuit manufacturing saw emergence of families such as Resistor-Transistor Logic (RTL), Diode-Transistor Logic (DTL), and Emitter-Coupled Logic (ECL), the first two now considered obsolete. Later emerged the Transistor-Transistor Logic (TTL) family based on NPN and PNP bipolar transistor circuitry, this family identified by part numbers beginning with either 54 or 74, the 54-series ICs having military-grade specifications (e.g. operating temperature limits) and the 74-series ICs having commercial-grade specifications. After that came the Complementary Metal-Oxide Semiconductor (CMOS) family based on N-channel and P-channel MOSFETs rather than bipolar transistors, this family identified by part numbers beginning with 4 or 14[1].

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

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

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

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

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

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

- **4000 classic CMOS family** – uses complementary (N- and P-channel together) MOSFET technology; operates on a wide range of DC power supply voltages, typically 3 Volts minimum to 18 Volts maximum, often standardized at 5 Volts, 10 Volts, or 15 Volts; notably slower in switching speed than classic TTL but operates at a *far* lower power dissipation[2]

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

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

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

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

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

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

- **5400/7400 LVC sub-family** – this "low-voltage CMOS" sub-family operates with a considerably lower DC power supply range than previous 5400/7400 digital logic sub-families, typically 2 Volts minimum and 3.6 Volts maximum, often standardized at 3.3 Volts

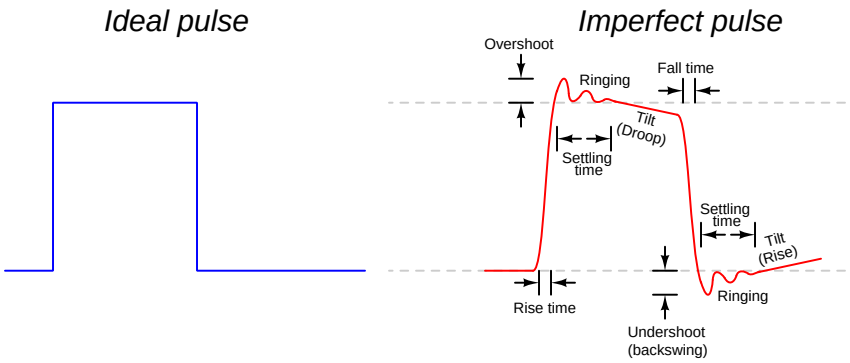
- **5400/7400 LVT sub-family** – this "low-voltage BiCMOS" sub-family uses a combination of bipolar transistors and MOSFETs internally and operates on a DC power supply voltage range of 2.7 Volts minimum and 3.6 Volts maximum, often standardized at 3.3 Volts

- **5400/7400 AVC sub-family** – this "advanced low-voltage CMOS" sub-family extends the operating DC power supply voltage to even lower levels with 1.4 Volts being minimum, often standardized at 3.3 Volts, 2.5 Volts, or 1.8 Volts

- **5400/7400 AUC sub-family** – this "advanced ultra-low voltage CMOS" sub family pushes the DC power supply voltage envelope down even further, with 0.8 Volts minimum and 3.6 Volts maximum, often standardized at 2.5 Volts, 1.8 Volts, and 1.2 Volts

4.2 Digital pulse criteria

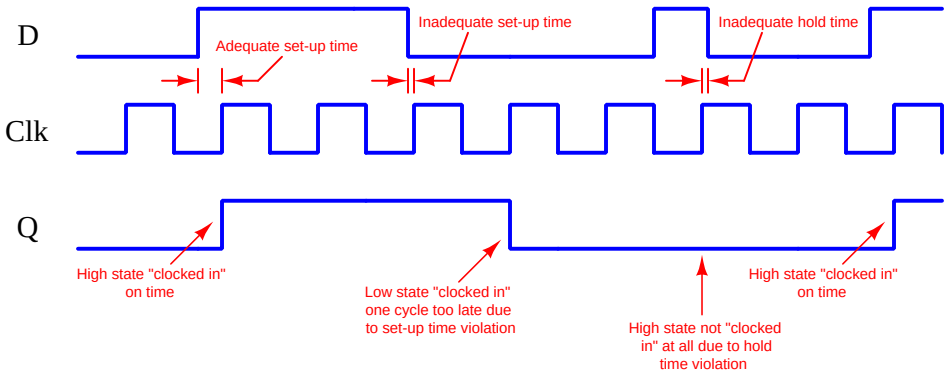
Ideal "pulse" signals have infinitely-steep rise and fall times, level "high" and "low" states well within the specified voltage ranges, and no other imperfections or artifacts. Real pulses always deviate from ideal, often in multiple ways:



Rise and fall times are strongly influenced by parasitic capacitance existing on the signal line with reference to ground, as well as the current-sourcing and current-sinking capability of the logic gate or other device generating the pulse signal. The capacitive "Ohm's Law" formula $I = C\frac{dV}{dt}$ predicts how much current will be necessary to create a linear rate-of-rise or rate-of-fall of voltage for a given capacitance. Note that to achieve infinitely steep rise or fall times an *infinite* amount of current would be necessary to charge/discharge whatever capacitance happens to exist on that signal line!

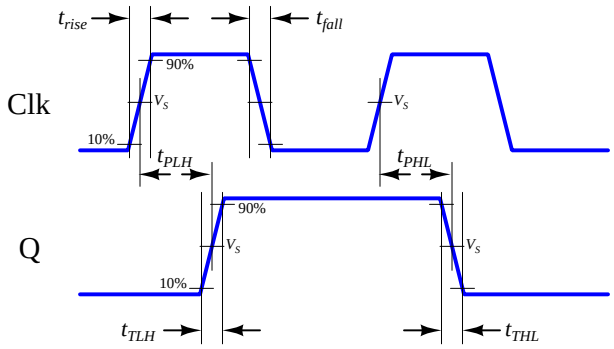
Ringing is caused by *resonance* occurring between parasitic capacitance and parasitic inductance, those two phenomena naturally exchanging energy back and forth with each other to produce the oscillatory waves we call "ringing". Overshoot and undershoot are also the result stored energy within these parasitic L and C circuit properties, and since parasitic capacitance and parasitic inductance can never be fully eliminated from any real circuit it means the effects of over/undershoot and ringing is likewise unavoidable. If you don't see these effects in your pulses, you just aren't viewing them at a fast enough time scale!

Clock-synchronized digital logic circuits such as counters, shift registers, and microprocessors require their input signals to be at stable states immediately before and immediately after the clock pulse arrives. For example, the following timing diagram shows input and output states for a D-type flip-flop circuit (positive-edge triggered), showing the effects of some signal timing violations:



Datasheets for digital circuits often provide timing diagrams showing criteria related to pulse signal timing and logic states. These diagrams don't typically show ideal square-edged pulses, but rather *trapezoidal* pulse profiles intended to exaggerate realistic features such as rise and fall times, propagation delays, and minimum set-up/hold times. Such diagrams usually confuse students who are accustomed to seeing square-edged pulses in their textbook timing diagrams. This technical reference will show some typical timing diagrams and explain what they represent.

For example, consider this timing diagram for a positive-edge-triggered JK flip-flop having both its J and K inputs tied high so as to maintain the circuit in its "toggle" mode. As such we would expect its output (Q) to change state with every rising edge of the clock pulse:



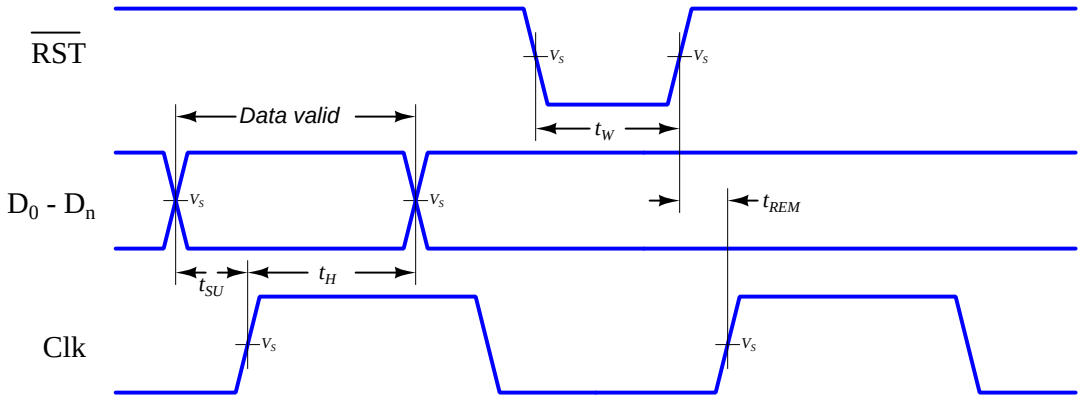
Each of the labels found in this diagram is defined as follows:

- t_{rise} = Rise time of input signal, typically measured from 10% of signal amplitude to 90% of signal amplitude
- t_{fall} = Fall time of input signal, typically measured from 90% of signal amplitude to 10% of signal amplitude
- t_{TLH} = Low-to-High transition time of output signal, typically measured from 10% of signal amplitude to 90% of signal amplitude (the same concept as rise time, but applied to the output signal instead of the input signal)
- t_{THL} = High-to-Low transition time of output signal, typically measured from 90% of signal amplitude to 10% of signal amplitude (the same concept as fall time, but applied to the output signal instead of the input signal)
- t_{PLH} = Propagation delay time of output signal when switching from low to high
- t_{PHL} = Propagation delay time of output signal when switching from high to low
- V_S = Switching threshold voltage, typically defined as 50% of signal amplitude

This timing diagram shows how a digital logic circuit reacts to a single input signal, in this case the clock pulse. Although this example happens to be for a JK flip-flop in toggle mode, the same type of timing diagram with its exaggerated rise/fall times and propagation delays could be applied to any digital logic gate whose output state depended solely on the state of a single input.

For synchronous digital logic circuits where input signals must coordinate with the clock pulse signal in order to be properly accepted by the circuit, we typically find timing diagrams comparing these input states to each other, often without showing the output(s) at all. Instead of showing us how the digital logic circuit will react to an input signal, this sort of timing diagram shows what the digital logic circuit *expects* of its multiple input signals.

The example is shown here for a positive-edge-triggered D register[3] having multiple data lines (D_0 through D_n), one asynchronous[4] reset line ($\overline{RST}$), and one clock input. The arbitrary logic levels of the multiple data lines are shown as a pair of complementary-state pulse waveforms, the only relevant features being the *timing* of the data and not the particular voltage levels of the data signals:



Labels shown in this diagram refer to *minimum* time durations the logic circuit requires for reliable operation:

- t_{SU} = Minimum set-up time before the arrival of the next clock pulse
- t_H = Minimum hold time following the last clock pulse
- t_W = Minimum width (duration) of the asynchronous reset pulse
- t_{REM} = Minimum removal time before the arrival of the next clock pulse

Violations of any of these minimum times may result in unexpected behavior from the logic circuit, and is an all-too-common cause of spurious errors in high-speed digital circuit designs. The assessment of digital pulse signals with regard to reliable circuit operation is generally known as *digital signal integrity*.

[3] In this case, a "D register" is synonymous with multiple D-type flip-flops sharing a common clock input, passing data through from each D input to each corresponding Q output synchronously with each clock pulse.

[4] To review, a *synchronous* input depends on a clock pulse while an *asynchronous* input is able to affect the circuit independent of the clock pulse.

Chapter 5

Programming References

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

5.1 Programming in C++

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

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

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

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

  cout << endl;

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

  return 0;
}
```

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

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

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

- `#include <iostream>` and `using namespace std;` are set-up instructions to the compiler giving it some context in which to interpret your code. The code specific to your task is located between the brace symbols ({ and }, often referred to as "curly-braces").
- `int main (void)` labels the "Main" function for the computer: the instructions within this function (lying between the { and } symbols) it will be commanded to execute. Every complete C++ program contains a `main` function at minimum, and often additional functions as well, but the `main` function is where execution always begins. The `int` declares this function will return an *integer* number value when complete, which helps to explain the purpose of the `return 0;` statement at the end of the `main` function: providing a numerical value of zero at the program's completion as promised by `int`. This returned value is rather incidental to our purpose here, but it is fairly standard practice in C++ programming.
- Grouping symbols such as (parentheses) and {braces} abound in C, C++, and other languages (e.g. Java). Parentheses typically group data to be processed by a function, called *arguments* to that function. Braces surround lines of executable code belonging to a particular function.
- The `float` declaration reserves places in the computer's memory for two *floating-point* variables, in this case the variables' names being `x` and `y`. In most text-based programming languages, variables may be named by single letters or by combinations of letters (e.g. `xyz` would be a single variable).
- The next two lines assign numerical values to the two variables. Note how each line terminates with a semicolon character (`;`) and how this pattern holds true for most of the lines in this program. In C++ semicolons are analogous to periods at the ends of English sentences. This demarcation of each line's end is necessary because C++ ignores whitespace on the page and doesn't "know" otherwise where one line ends and another begins.
- All the other instructions take the form of a `cout` command which prints characters to the "standard output" stream of the computer, which in this case will be text displayed on the console. The double-less-than symbols (`<<`) show data being sent *toward* the `cout` command. Note how verbatim text is enclosed in quotation marks, while variables such as `x` or mathematical expressions such as `x - y` are not enclosed in quotations because we want the computer to display the numerical values represented, not the literal text.
- Standard arithmetic operations (add, subtract, multiply, divide) are represented as `+`, `-`, `*`, and `/`, respectively.
- The `endl` found at the end of every `cout` statement marks the end of a line of text printed to the computer's console display. If not for these `endl` inclusions, the displayed text would resemble a run-on sentence rather than a paragraph. Note the `cout << endl;` line, which does nothing but create a blank line on the screen, for no reason other than esthetics.

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

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

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

```
./myprogram.exe
```

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

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

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

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

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

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

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

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

5.2 Programming in Python

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
x = 200
y = -560.5

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

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

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

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

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

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

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

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

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

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

5.3 Modeling an SR latch using C++

The following program, written in the C++ language, simulates the operation of an SR latch. It is a "looping" program designed to repeatedly prompt the user for S and R input states, displaying the Q and $\overline{Q}$ output states in response:

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

int main (void)
{
  bool Set, Reset, Out;

  Out = 0;

  while(1)
  {
    cout << "--------" << endl;
    cout << "Set = ";
    cin >> Set;
    cout << "Reset = ";
    cin >> Reset;

    cout << endl;

    if (Set == 1)
      Out = 1;

    if (Reset == 1)
      Out = 0;

    cout << "Q = " << Out << endl;
    cout << "Q' = " << !Out << endl;
  }

  return 0;
}
```

Let's analyze how this program words "SET" and "RESET" in addition to assigning values to Q:

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

int main (void)
{
  bool Set, Reset, Out;

  Out = 0;

  while(1)
  {
    cout << "--------" << endl;
    cout << "Set = ";
    cin >> Set;
    cout << "Reset = ";
    cin >> Reset;

    cout << endl;

    if (Set == 1)
    {
      Out = 1;
      cout << "SET" << endl;
    }

    if (Reset == 1)
    {
      Out = 0;
      cout << "RESET" << endl;
    }

    cout << "Q = " << Out << endl;
    cout << "Q' = " << !Out << endl;
  }

  return 0;
}
```

When run, we get the following result:

```
--------
Set = 0
Reset = 0

Q = 0
Q' = 1
--------
Set = 1
Reset = 0

SET
Q = 1
Q' = 0
--------
Set = 0
Reset = 0

Q = 1
Q' = 0
--------
Set = 0
Reset = 1

RESET
Q = 0
Q' = 1
--------
Set = 0
Reset = 0

Q = 0
Q' = 1
```

Note how the words `SET` and `RESET` print only when the respective inputs are activated, and not when the system is latching (i.e. both inputs 0).

5.4 Modeling an SR flip-flop using C++

The following code simulates an SR *flip-flop* rather than an SR *latch*:

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

int main (void)
{
  bool Set, Reset, Clock, LastClock, Out;

  Out = 0;
  LastClock = 0;

  while(1)
  {
    cout << "--------" << endl;
    cout << "Set = ";
    cin >> Set;
    cout << "Reset = ";
    cin >> Reset;
    cout << "Clock = ";
    cin >> Clock;

    cout << endl;

    if (Clock == 1 && LastClock == 0)
    {
      if (Set == 1)
        Out = 1;

      if (Reset == 1)
        Out = 0;
    }

    cout << "Q = " << Out << endl;
    cout << "Q' = " << !Out << endl;
    LastClock = Clock;
  }

  return 0;
}
```

The difference, of course, between a flip-flop and a latch is that the flip-flop responds to inputs only when the clock signal *transitions* from one state to another. In this case, we have a program for a flip-flop acting on the *rising* edge of the `Clock` signal. C/C++ `if` conditional statements, however, are only able to check for states of equality and inequality, not dynamic changes. Somehow we must write the program so that the "outer" `if` statement is satisfied only when the `Clock` *switches* from 0 to 1.

Comparison of this code against the last (for the SR *latch*) reveals the addition of two new features: new Boolean variables added to the declaration line (`Clock` and `LastClock`), and *nested* `if` statements. By "nested" I mean to say we have `if` statements written *inside of* another `if` statement, such that the "inner" `if` conditionals are only executed when the "outer" `if` condition is met.

We can accomplish this change-of-state check by using *two* variables for the clock signal, in this case `Clock` and `LastClock`. Whatever value the user enters for the clock signal is assigned to the `Clock` variable, while `LastClock` "remembers" the value of `Clock` during the last iteration of the `while` loop.

The variable `LastClock` gets initialized to zero prior to the start of the `while` loop. Meanwhile, the user enters the value of `Clock` near the beginning of the `while` loop along with the other inputs. `LastClock` doesn't get its value updated until the very end of the `while` loop after all the conditional statements have been evaluated, where it gets assigned the value held by `Clock`. The result of this is that at the point of evaluation, `Clock` holds the most recent user entry while `LastClock` holds the user entry from the *last* `while` iteration.

The outer `if` statement permits the inner conditionals to be evaluated only if `Clock` is equal to 1 *and* `LastClock` is equal to 0, which means the clock signal has just transitioned from a (last) 0 state to a (present) 1 state.

A sample execution of this program demonstrates *setting* the flip-flop:

```
--------
Set = 0
Reset = 0
Clock = 0

Q = 0
Q' = 1
--------
Set = 1
Reset = 0
Clock = 0

Q = 0
Q' = 1
--------
Set = 1
Reset = 0
Clock = 1

Q = 1
Q' = 0
--------
Set = 0
Reset = 1
Clock = 1

Q = 1
Q' = 0
--------
Set = 0
Reset = 1
Clock = 0

Q = 1
Q' = 0
--------
Set = 0
Reset = 1
Clock = 1

Q = 0
Q' = 1
```

Note how neither the set action nor the reset action occurs until the clock transitions from 0 to 1.

5.5 Modeling a JK flip-flop using C++

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

int main (void)
{
  bool J, K, Clock, LastClock, Out;

  Out = 0;
  LastClock = 0;

  while(1)
  {
    cout << "--------" << endl;
    cout << "J = ";
    cin >> J;
    cout << "K = ";
    cin >> K;
    cout << "Clock = ";
    cin >> Clock;

    cout << endl;

    if (Clock == 1 && LastClock == 0)
    {
      if (J == 1 && K == 0)
        Out = 1;

      else if (J == 0 && K == 1)
        Out = 0;

      else if (J == 1 && K == 1)
        Out = !Out;
    }

    cout << "Q = " << Out << endl;
    cout << "Q' = " << !Out << endl;
    LastClock = Clock;
  }

  return 0;
}
```

JK flip-flops differ from SR flip-flops in one important regard: when both J and K are active, the Q and $\overline{Q}$ output lines *toggle* at the active edge of the clock pulse signal. To implement this additional feature requires we modify the inner `if` conditionals, adding one more to account for the simultaneous activation of `J` and `K`.

Each `if` conditional must check *both* `J` and `K` variables. Since these three conditions are mutually exclusive, we are able to use `else if` statements instead of `if` statements for the last two of the three. Truth be told, we could have used plain `if` statements for all three conditionals, but when the respective conditions are mutually exclusive, `else if` is slightly more efficient. This is because the computer will completely skip past any of the `else if` statements if one of the previous conditions is met. If we were to use plain `if` statements for all three conditions, the computer would take the time to evaluate each and every one of those conditions even if it was not logically necessary.

Another type of conditional statement offered by C and C++ alike, but not used in this program, is the `else` statement. This is applicable at the very end of a set of `if` and `else if` conditionals, to trigger code execution if *none of the previous conditions were met*. We have no purpose for an `else` statement in this JK flip-flop simulation, but it's worth mentioning while we are on the topic of C/C++ `if` statements.

Compiling and running this code, we are able to demonstrate its "toggle" functionality:

```
--------
J = 1
K = 1
Clock = 0

Q = 0
Q' = 1
--------
J = 1
K = 1
Clock = 1

Q = 1
Q' = 0
--------
J = 1
K = 1
Clock = 1

Q = 1
Q' = 0
--------
J = 1
K = 1
Clock = 0

Q = 1
Q' = 0
--------
J = 1
K = 1
Clock = 1

Q = 0
Q' = 1
```

Chapter 6

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?

6.1 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.

6.1.1 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.

6.1.2 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

Logic function

Truth table

Boolean algebra

OR function

AND function

NOT function

NOR function

NAND function

XOR function

Logic state

Logic gate

Relay ladder diagram

Latch

Astable

Monostable

Bistable

Flip-flop

Multivibrator

Seal-in

Set versus Reset

Timing diagram

SR latch

Active-low versus Active-high

Disallowed state

Race condition

Set-up time

Hold time

Switch contact bounce

Enabling

Clocking

D latch

Octal D latch

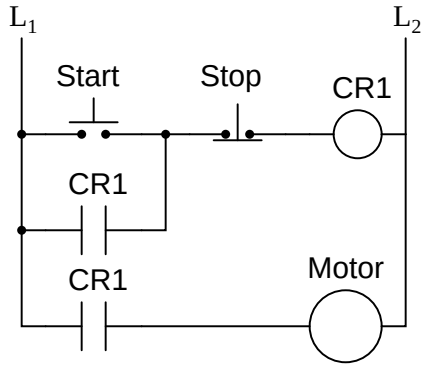
JK flip-flop

Quadrature

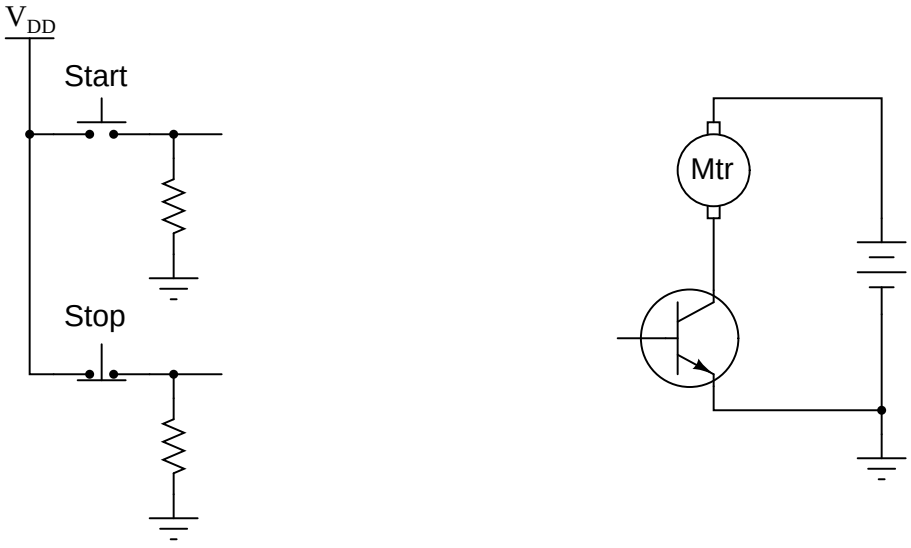
Bus

6.1.3 CMOS gate equivalent of motor control circuit

The following relay logic circuit is for starting and stopping an electric motor:



Draw the CMOS logic gate equivalent of this motor start-stop circuit, using these two pushbutton switches as inputs:



Make sure that your schematic is complete, showing how the logic gate will drive the electric motor (through the power transistor shown).

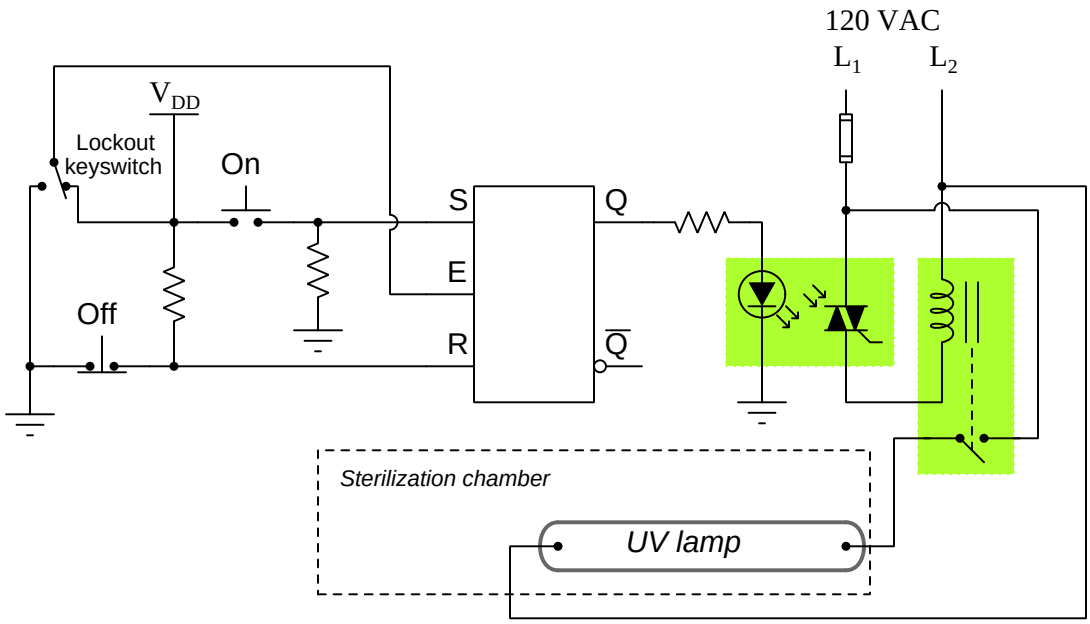
Challenges

- Why is the "Stop" switch always normally-closed in motor control circuits, whether it be relay logic or semiconductor logic? It is easy enough to invert a signal if we wish to, either by using a relay or by using a NOT gate, so shouldn't the choice of switch "normal" status be arbitrary?

- Why not operate the electric motor off the same V_{DD} power source that the gates are powered by? If we had to do such a thing, what circuit additions would you propose to minimize any potential trouble?

6.1.4 UV lamp control circuit

Here, a gated SR latch is being used to control the electric power to a powerful ultraviolet (UV) lamp, used for sterilization purposes in a life-sciences laboratory:



Explain the purpose of the "Lockout" keyswitch.

Explain how the CMOS latch is able to exert control over the high-power lamp despite the latch IC having very limited current sourcing and sinking capability.

Now, suppose the lab personnel want to add a feature to the ultraviolet sterilization chamber: an electric solenoid door lock, so that personnel can open the door to the chamber only if the following conditions are met:

- Lamp is *off*
- "Lockout" switch is sending a "low" signal to the latch's Enable input

Modify this circuit so that it energizes the door lock solenoid, allowing access to the chamber, only if the above conditions are both true.

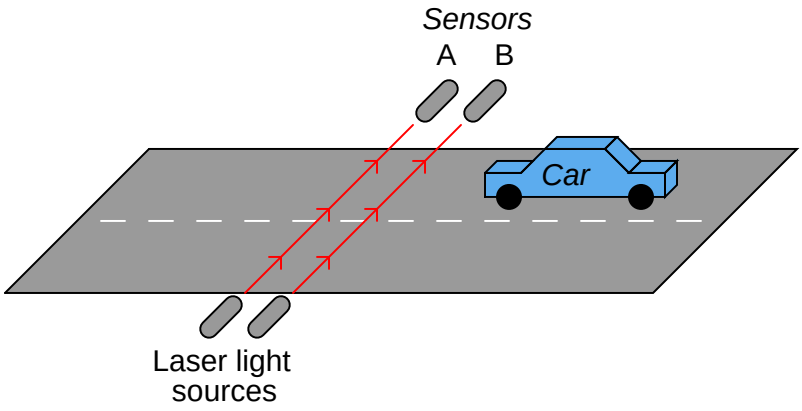
Challenges

- Suppose someone presents a design modification for the electric door lock, and their design senses the status of the latch's Q output rather than directly sensing power applied to the UV lamp. Identify how a component failure in this system might create an unsafe condition, whereby the UV lamp is still energized when the door unlocks.

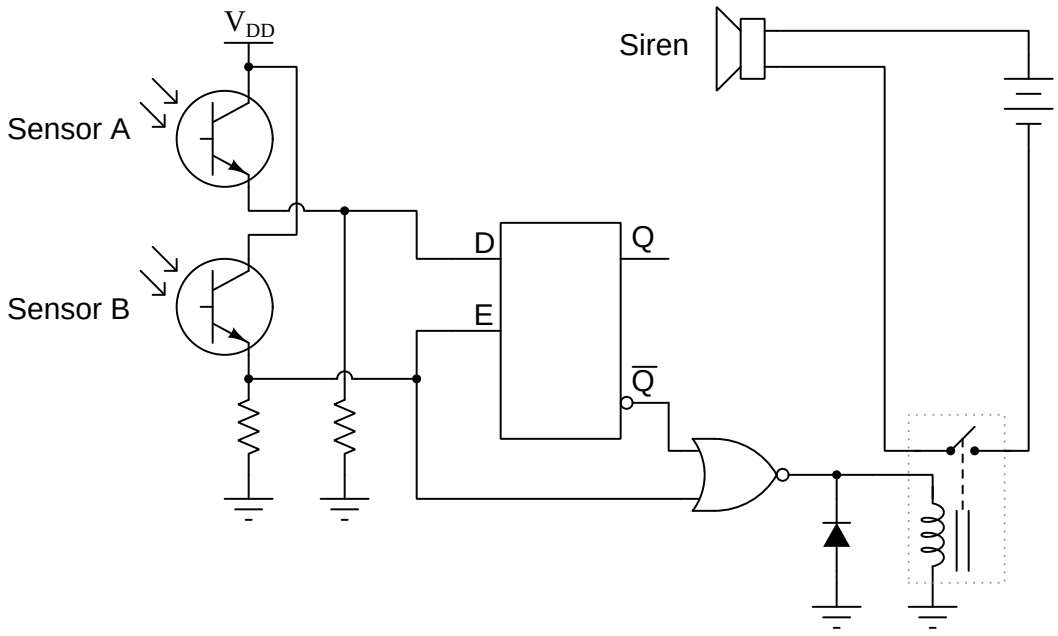
- Defend the use of *two* cascaded interposing devices in the UV lamp circuit. Why not just use one device?
- A common industrial safety practice called *Lock-Out, Tag-Out* ("LOTO") is used to secure all dangerous sources of energy to a system prior to maintenance or repair work commencing on that system. This usually requires turning circuit breakers off, pulling fuses out of sockets, etc., followed by attaching warning tags and finally by inserting a locking device to prevent re-energization. Where would you recommend this procedure being done to this system?

6.1.5 Wrong way siren circuit

This one-way street is equipped with an alarm to signal drivers going the wrong way. The sensors work by light beams being broken when an automobile passes between them. The distance between the sensors is less than the length of a normal car, which means as a car passes by, first one beam is broken, then both beams become broken, then only the last beam is broken, then neither beam is broken. The sensors are phototransistors sensitive only to the narrow spectrum of light emitted by the laser light sources, so that ambient sunlight will not "fool" them:



Both sensors connect to inputs on a D-type latch, which is then connected to some other circuitry to sound an alarm when a car goes down the road the wrong way:



The first question is this: which way is the *correct* way to drive down this street? From left to right, or from right to left (as shown in the illustration)?

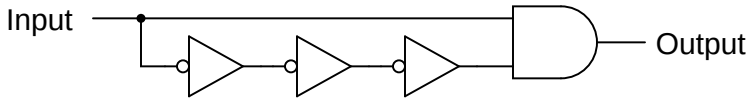
The second question is, how will the system respond if sensor A's laser light source fails? What will happen if sensor B's laser light source fails?

Challenges

- Explain why a good problem-solving strategy to use here would be to sketch a timing diagram showing all the signal states for each direction of the vehicle's travel.
- Explain why a good problem-solving strategy to use here would be to write the truth table for a NOR gate.
- A practical component included in this schematic diagram is a *commutating diode*. Identify its purpose.

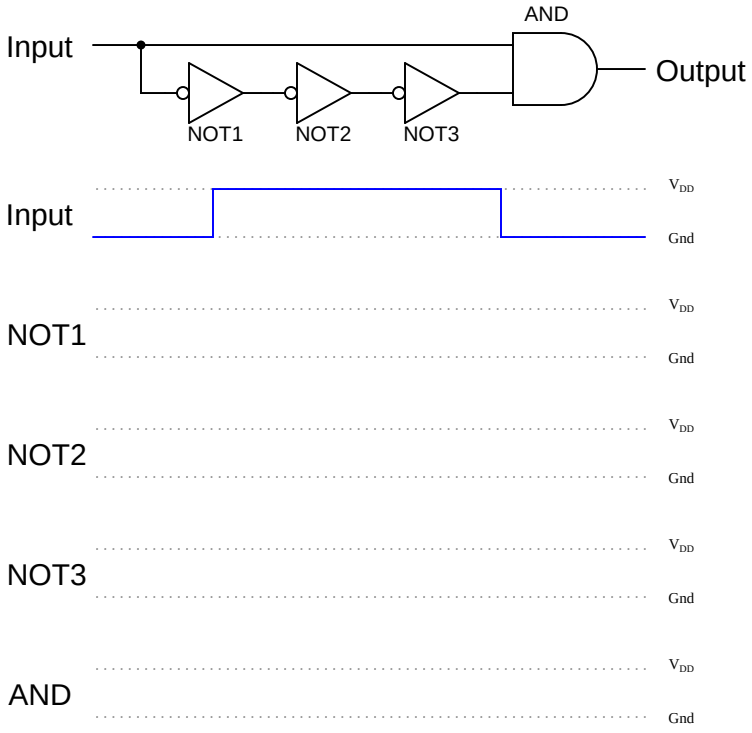
6.1.6 Simple one-shot circuit

Usually, propagation delay is considered an undesirable characteristic of logic gates, which we simply have to live with. Other times, it is a useful, even necessary, trait. Take for example this circuit:



If the gates constituting this circuit had zero propagation delay, it would perform no useful function at all. To verify this sad fact, analyze its steady-state response to a "low" input signal, then to a "high" input signal. What state is the AND gate's output always in?

Now, consider propagation delay in your analysis by completing a timing diagram for each gate's output, as the input signal transitions from low to high, then from high to low:



What do you notice about the state of the AND gate's output now?

Challenges

- Describe exactly what conditions are necessary to obtain a "high" signal from the output of this circuit, and what determines the duration of this "high" pulse.

- Explain how we might be able to modify the circuit's pulse *duration*.

6.1.7 Frequency division

One useful feature of a JK flip-flop circuit in its *toggle* mode is the ability to reduce the frequency of a clock signal. When J and K are both held high (i.e. connected to +V), the frequency of the pulse signal at either Q or $\overline{Q}$ will be one-half the clock frequency.

Knowing this, design a circuit using JK flip-flops that divides the frequency of a clock signal by a factor of *four*.

Challenges

- How could you divide frequency by one-*third*?

6.2 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.

6.2.1 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.

6.2.2 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	

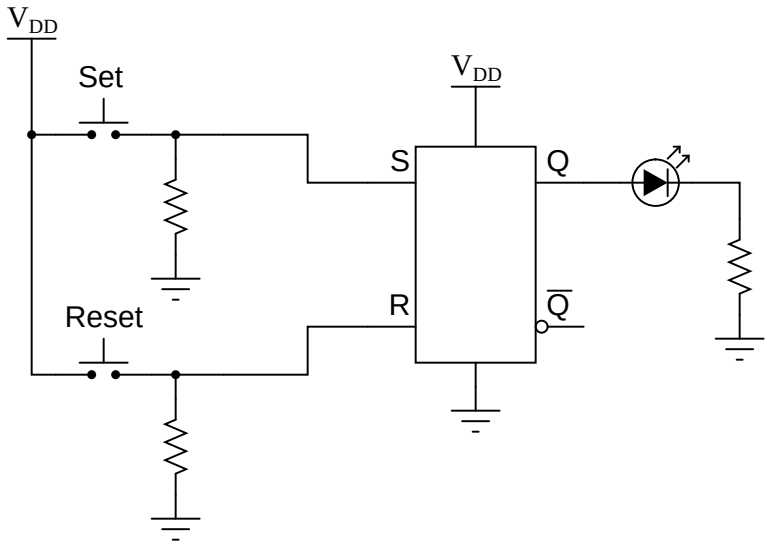
Note how the square-root term (y) is calculated in cell `C1`, and the denominator term (z) in cell `C2`. This makes the two final formulae (in cells `B1` and `B2`) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly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

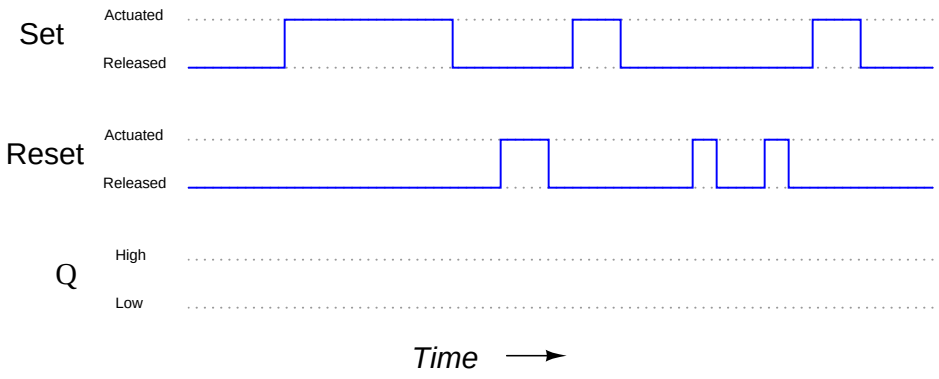
Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application 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.*

6.2.3 Timing diagram for an SR latch

Complete the timing diagram, showing the state of the Q output over time as the Set and Reset switches are actuated. Assume that Q begins in the low state on power-up:



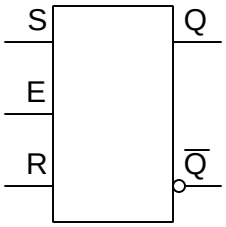


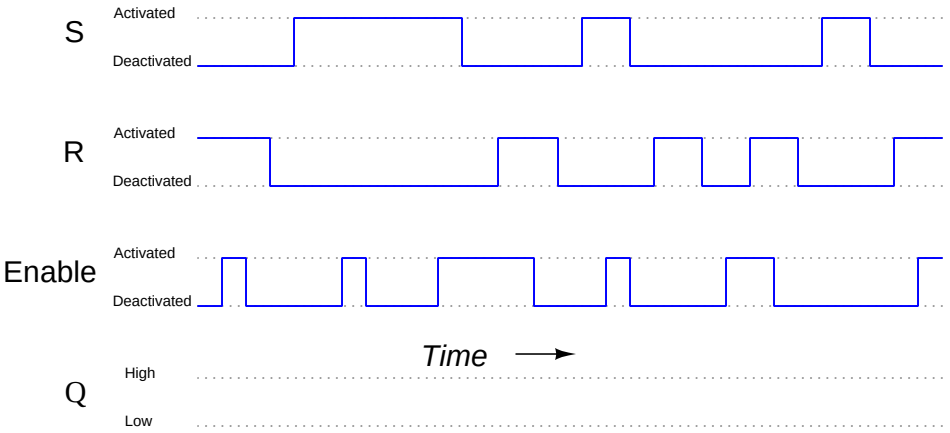
Challenges

- Describe the $\overline{Q}$ output's timing diagram.
- Describe a problem-solving strategy useful for completing timing diagrams.

6.2.4 Timing diagram for an enabled SR latch

Complete the timing diagram, showing the state of the Q output over time as the input switches are actuated. Assume that Q begins in the low state on power-up:



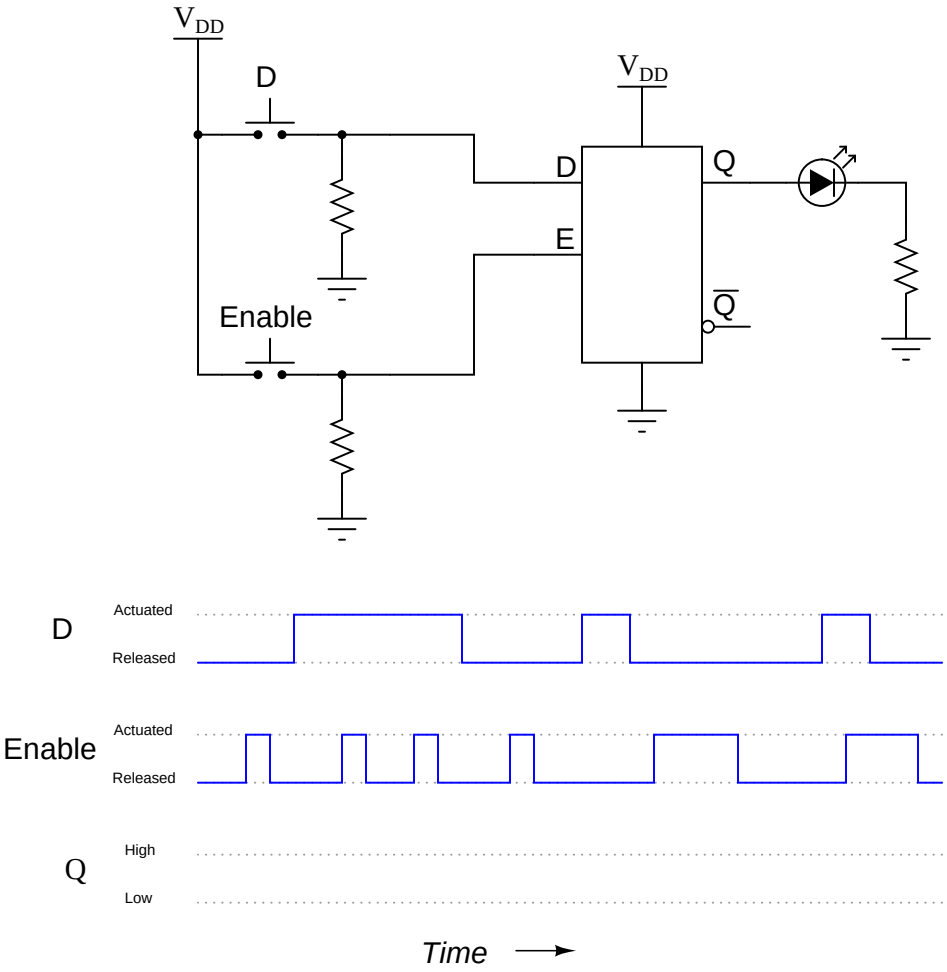


Challenges

- Describe the $\overline{Q}$ output's timing diagram.
- Describe a problem-solving strategy useful for completing timing diagrams.

6.2.5 Timing diagram for a D latch

Complete the timing diagram, showing the state of the Q output over time as the input switches are actuated. Assume that Q begins in the low state on power-up:

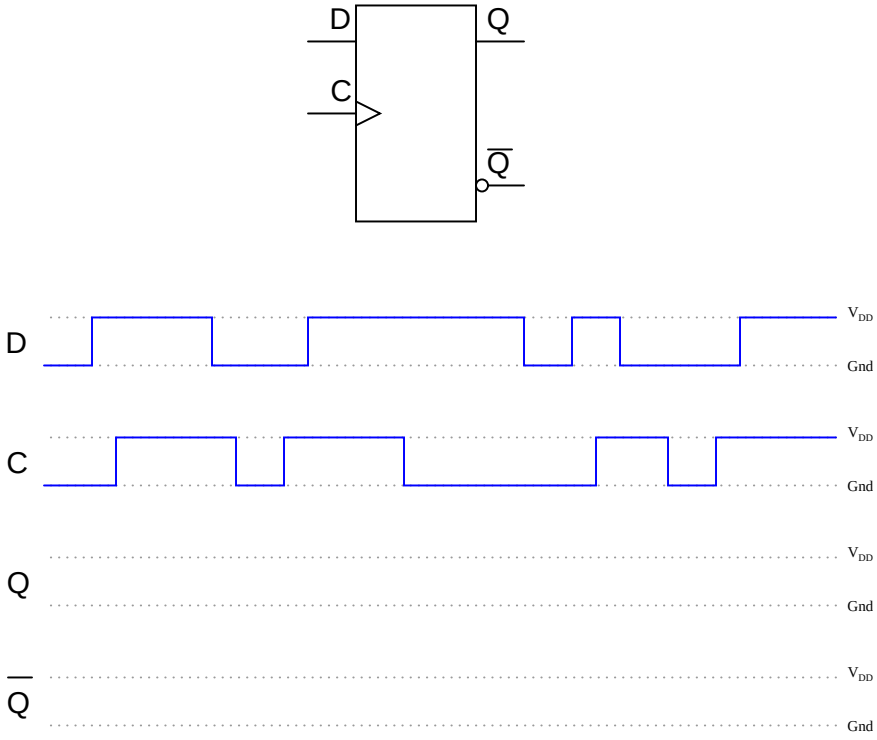


Challenges

- Describe a problem-solving strategy useful for completing timing diagrams.

6.2.6 Timing diagram for a D flip-flop

Determine the output states for this D flip-flop, given the pulse inputs shown:

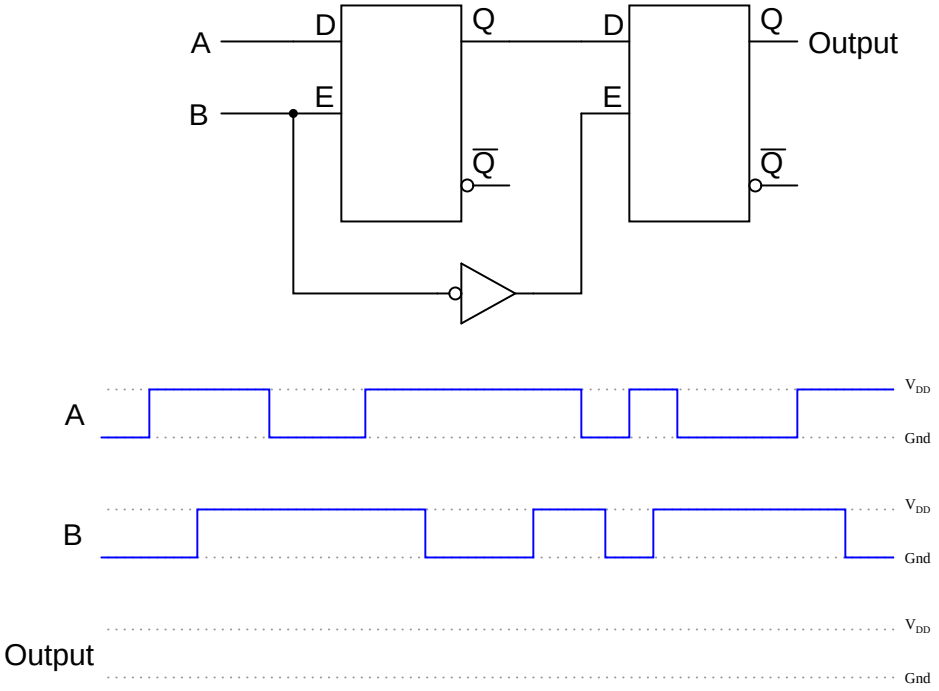


Challenges

- Describe a problem-solving strategy useful for completing timing diagrams.

6.2.7 Cascaded D latches

Determine the final output states over time for the following circuit, built from D-type gated latches:



At what specific times in the pulse diagram does the final output assume the input's state? How does this behavior differ from the normal response of a D-type latch?

Challenges

- How would the behavior of this circuit change if the inverter were swapped out for a buffer?

6.3 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.

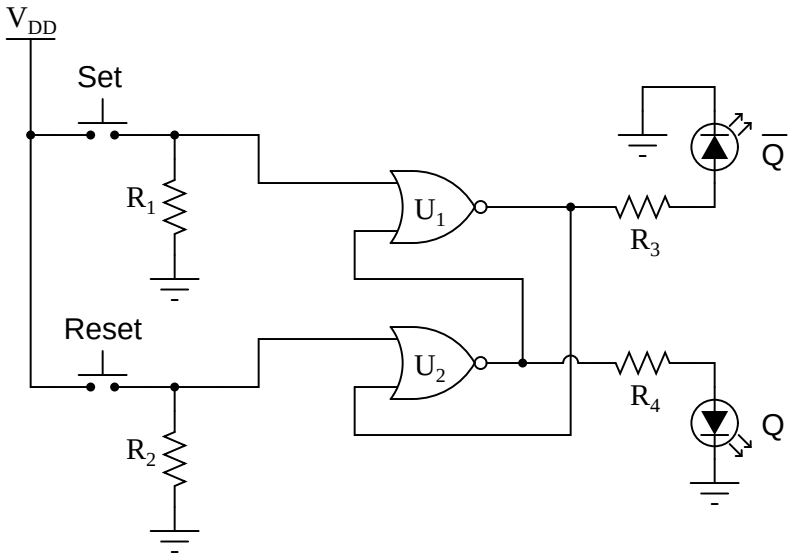
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of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

6.3.1 Identifying fault in a NOR-based SR latch circuit

Identify at least one component fault that would cause the Q LED to always stay on, no matter what was done with the input switches.



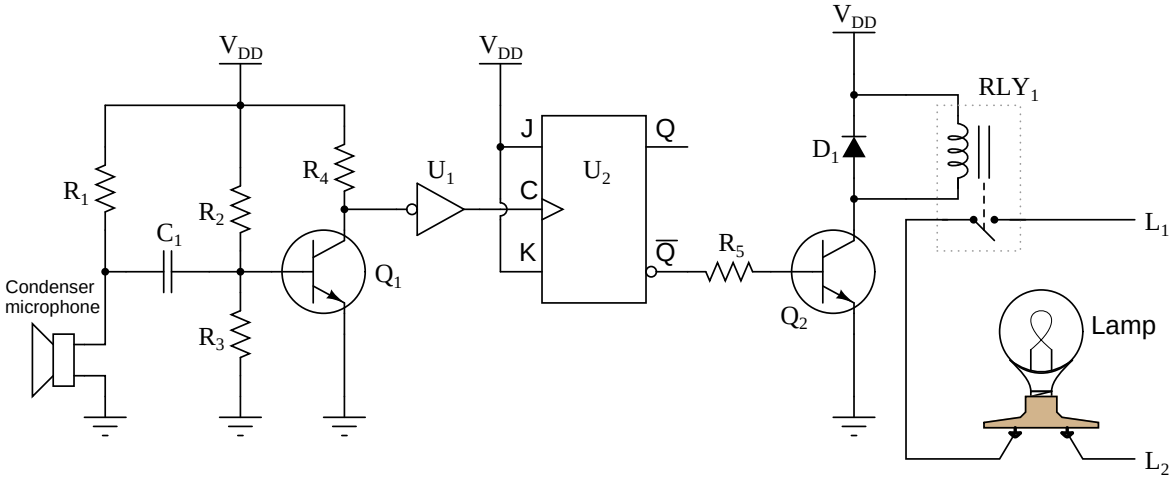
For each of your proposed faults, explain *why* it will cause the described problem.

Challenges

- Describe the function of the two resistors R_1 and R_2.

6.3.2 Sound-controlled lamp

Predict how the operation of this sound-activated lamp circuit[11] will be affected as a result of the following faults. Consider each fault independently (i.e. one at a time, no coincidental faults):



- Resistor R_1 fails open:
- Resistor R_3 fails open:
- Diode D_1 fails open:
- Transistor Q_2 fails shorted between collector and emitter:
- Solder bridge past resistor R_5:

Challenges

- Describe how this circuit is supposed to function.
- How would this circuit's function be affected by connecting the base of Q_2 to the flip-flop's Q output rather than its $\overline{Q}$ output?
- For each of these faults, describe how you could pinpoint the fault using diagnostic tests such as meter or oscilloscope measurements.

[11] This circuit is designed to turn a lamp on and off whenever a loud sound is detected by the microphone, such as a hand clap.

Appendix A

Problem-Solving Strategies

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

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

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

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

Appendix B

Instructional philosophy

B.1 First principles of learning

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

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

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

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

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

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

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

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

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

B.2 Proven strategies for instructors

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

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

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

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

B.3 Proven strategies for students

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

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

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

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

B.4 Design of these learning modules

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

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

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

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

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

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

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

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

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

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

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

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

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

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

The `GNU/Linux` computer operating system

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

Bram Moolenaar's `Vim` text editor

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

Donald Knuth's TEX typesetting system

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

Leslie Lamport's LATEX extensions to TEX

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

Tim Edwards' `Xcircuit` drafting program

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

`Gimp` graphic image manipulation program

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

`SPICE` circuit simulation program

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

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

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

`gnuplot` mathematical visualization software

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

`Python` programming language

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

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

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

Section 1 – Definitions.

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

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

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

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

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

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

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

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

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

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

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

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

Section 2 – Scope.

a. License grant.

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

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

B. produce, reproduce, and Share Adapted Material.

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

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

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

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

5. Downstream recipients.

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

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

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

b. Other rights.

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

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

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

Section 3 – License Conditions.

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

a. Attribution.

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

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

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

ii. a copyright notice;

iii. a notice that refers to this Public License;

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

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

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

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

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

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

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

Section 4 – Sui Generis Database Rights.

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

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

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

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

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

Section 5 – Disclaimer of Warranties and Limitation of Liability.

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

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

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

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

Section 6 – Term and Termination.

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

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

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

2. upon express reinstatement by the Licensor.

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

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

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

Section 7 – Other Terms and Conditions.

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

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

Section 8 – Interpretation.

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

be made without permission under this Public License.

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

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

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

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

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

Appendix E

References

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

Appendix F

Version history

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

29 April 2026 – typographical error correction courtesy of Wyatt Gleason.

20 November 2024 – added a new Case Tutorial section showing how to create quadrature pulse waveforms using JK flip-flops.

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

29 April 2024 – consolidated latch and flip-flop Case Tutorial examples into a single file to be included in this and other modules.

29 November 2023 – added a comment to the Tutorial about "registered" logic, and also more questions to the Introduction chapter.

20 July 2023 – added a new Case Tutorial section on the phenomenon of switch bounce as well as mitigation techniques.

8 May 2023 – clarified "invalid" states for latch circuits. Also added some Case Tutorial examples with cascaded flip-flops, and added a line of text explaining the assumption of Q's initial state to the other latch and flip-flop examples.

25 April 2023 – edited Challenge questions in "Wrong way siren circuit" Conceptual Reasoning question. Also added a Case Tutorial section on timing diagrams applied to simple combinational logic circuits.

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

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

27 July 2022 – edited image_6204 in the "Example: enabled SR latch timing diagram" Case Tutorial section to not say "Actuated" and "Released" which apply to mechanical switch states.

25 May 2022 – added a $\overline{Q}$ trace to image_6204 within the "Example: enabled SR latch timing diagram" Case Tutorial section.

2 May 2022 – added a Tutorial section on preset and clear inputs, and also added a Case Tutorial section with timing diagram examples.

28 April 2022 – added a commutating diode in parallel with the relay coil on the "Wrong way siren circuit" Conceptual Reasoning question.

6 December 2021 – added timing diagrams to instructor notes for the "Wrong way siren circuit" Conceptual Reasoning question.

26 November 2021 – added master-slave flip-flop technology to the Tutorial, as well as divided the Tutorial into sections.

9 May 2021 – commented out or deleted empty chapters.

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

5 June 2020 – added mention of digital "busses" to the Tutorial.

23 March 2020 – added more problems.

12 March 2020 – minor edits to wording in the tutorial, including some elaboration on flip-flops and latches as *registers* for digital words.

7 March 2020 – added Technical Reference section on digital pulse criteria.

5 March 2020 – minor edit to schematic diagram for the "UV lamp control circuit" Conceptual Reasoning problem.

27 February 2020 – added Foundational Concepts to the list in the Conceptual Reasoning section.

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

2 January 2020 – added C++ programming example simulating a latch.

20 November 2019 – commented that the D-type latch is often referred to as a *transparent* latch. Also added index entries for each type of latch and flip-flop.

14 November 2019 – typeset all input labels as mathematical variables (e.g. J instead of J). Also, some minor edits made to the text.

13 November 2019 – added more questions.

13 September 2019 – added mention of set-up and hold times to Tutorial.

16 June 2019 – minor edits to diagnostic questions, replacing "no multiple faults" with "no coincidental faults".

27 March 2019 – added questions to Conceptual, Quantitative, and Diagnostic Reasoning sections.

26 March 2019 – added more discussion on the topic of clock pulses and edge-triggering of flip-flops to the Tutorial, as well as the application of D-type flip-flops to quadrature signal direction.

15 March 2019 – added more content to the Tutorial.

10 March 2019 – document first created, Tutorial incomplete.

Index

4000/14000 CMOS family, 44
5400/7400 TTL family, 44

Active-high, 30
Active-low, 30
Adding quantities to a qualitative problem, 102
AND function, 25
Annotating diagrams, 8, 101
Asynchronous, 50

Bipolar, 26
Bistable, 31
Boolean algebra, 25
Bus, 37

C++, 52
Capacitance, parasitic, 47
Checking for exceptions, 102
Checking your work, 102
Clear, 41
Clock signal, 34
CMOS, 26
Code, computer, 111
Compiler, C++, 52
Computer programming, 51

D flip-flop, 36
D latch, 36
Diagram, timing, 8
Digital signal integrity, 50
Digital word, 37
Dimensional analysis, 101
Diode-Transistor Logic family, 44
Disallowed state, 30, 31
Discrete, 25
DTL family, 44

ECL family, 44
Edwards, Tim, 112
Emitter-Coupled Logic family, 44
Enable input, 33
Encoder, rotary, 38

Fall time, 49
Family, logic gate, 44
Flip-flop, 34
Flip-flop, D, 36
Flip-flop, JK, 39
Flip-flop, SR, 35

Graph values to solve a problem, 102
Greenleaf, Cynthia, 71

Hold time, 35, 48
How to teach with these modules, 109
Hwang, Andrew D., 113

IC, 31
Identify given data, 101
Identify relevant principles, 101
Inductance, parasitic, 47
Integrated circuit, 31
Integrity, signal, 50
Intermediate results, 101
Interpreter, Python, 56
Invalid state, 30, 31
Inverted instruction, 109

Java, 53
JK flip-flop, 39

Knuth, Donald, 112

Lamport, Leslie, 112
Latch, 3, 28
Latch, D, 36

Latch, SR, 30
Limiting cases, 102
Logic function, 25
Logic gate family, 44
Logic gate sub-family, 45
Logic level, 25
Logic state, 25
Logic, registered, 37

Master-slave flip-flop, 40
Medium-scale integration, 37
Metacognition, 76
Monostable multivibrator, 34
Moolenaar, Bram, 111
Motorola, 44
Multivibrator, 34
Murphy, Lynn, 71

NAND function, 25
Negative edge triggering, 39
NOR function, 25
NOT function, 25

Octal flip-flop, 37
One-shot, 34
Open-source, 111
OR function, 25

Parallel, 20
Parasitic capacitance, 47
Parasitic inductance, 47
Positive edge triggering, 39, 49
Power supply rail, 26
Preset, 41
Problem-solving: annotate diagrams, 8, 101
Problem-solving: check for exceptions, 102
Problem-solving: checking work, 102
Problem-solving: dimensional analysis, 101
Problem-solving: graph values, 102
Problem-solving: identify given data, 101
Problem-solving: identify relevant principles, 101
Problem-solving: interpret intermediate results, 101
Problem-solving: limiting cases, 102
Problem-solving: qualitative to quantitative, 102
Problem-solving: quantitative to qualitative, 102
Problem-solving: reductio ad absurdum, 102
Problem-solving: simplify the system, 101
Problem-solving: thought experiment, 8, 101
Problem-solving: track units of measurement, 101
Problem-solving: visually represent the system, 101
Problem-solving: work in reverse, 102
Programming, computer, 51
Propagation delay, 49
Python, 56

Quadrature signals, 21, 38
Qualitatively approaching a quantitative problem, 102

Race condition, 31
Rail, power supply, 26
Reading Apprenticeship, 71
Reductio ad absurdum, 102, 108, 109
Register, 37, 50
Registered logic, 37
Reset, 28, 41
Resistor-Transistor Logic family, 44
Resonance, 47
Rise time, 49
Rotary encoder, 38
RTL family, 44

Schmitt trigger logic gate, 20
Schoenbach, Ruth, 71
Scientific method, 76
Seal-in, 28
Serial, 20
Set, 28, 41
Set-up time, 35, 40, 48
Shift register, 20
Signal integrity, 50
Signal, discrete, 25
Simplifying a system, 101
Socrates, 108
Socratic dialogue, 109
Source code, 52
SPDT, 32
SPICE, 71
SR flip-flop, 35

SR latch, 30
Stallman, Richard, 111
Sub-family, logic gate, 45
Synchronous, 50

Thought experiment, 8, 101
Timing diagram, 8, 29
Toggle mode, 39, 49
Toggle switch, 3
Torvalds, Linus, 111
Transition time, 49
Truth table, 25, 29

Units of measurement, 101

Visualizing a system, 101

Whitespace, C++, 52, 53
Whitespace, Python, 59
Word, 37
Work in reverse to solve a problem, 102
WYSIWYG, 111, 112

XOR function, 25