Fundamental Systems Structure

Core concepts in CPU architecture, the memory hierarchy, compilers, and operating systems

Scott F. H. Kaplan
sfkaplan@cs.amherst.edu

Contents

Chapter 1

Introduction

Computing devices are pervasive. A great effort is required, in our society, to avoid their use for all sorts of activities. Some of those activities are obvious: sending email; browsing the web; using a word processor or spreadsheet; playing video games. However, computing devices are used in a much wider variety of devices: cars; airplanes; watches and clocks; mobile telephones; portable music players; televisions; cameras; bank ATM's. However, few people who use these devices understand how they work. That they are so useful to so many people, without those people understanding their structure or inner function, is a testament to their design, hiding their complex inner workings as they perform only seemingly simple tasks.

However, *some people* must understand how these systems are designed and structured. However, for many, the workings of a simple calculator are a mystery, let alone a full-fledged, modern computer system.

1.1 Why this book?

Computer systems comprise a large number of *components* or *layers*, and each of these is commonly presented separately from the others. Thus, gaining a fundamental understanding of how such a system could be constructed requires a number of texts and, perhaps, and number of academic courses. Additionally, each of these texts and courses tends to delve into one system component in detail, making it difficult to separate the essential and basic concepts and structures from various refinements and optimizations for that component.

This book, instead, provides a single, unified presentation of the fundamental structure of a complete computing system. It begins with the lowest level—digital logic—and proceeds to compose the capabilities at each level to develop the next. Ultimately, it presents the basic structure of a CPU and memory bus, the design of a compiler, and the structure of an operating system kernel with its basic components. As a whole, this book presents *one simple way* of developing a complete computing system, bottom to top. It addresses the fundamental concepts required for this design, stripping away the confusing details that come with the development of a full-featured, optimized computing system.

By reading this entire book, or by taking courses that follow the structure of this book, you will ultimately see how a complete computing system works. You will not know the details of any one real-world system, but you will understand the core concepts that will allow you to grasp those details of any such system. Moreover, you will be able to delve into each component or layer and understand not only the deeper concepts, but also how that layer interacts with others. It is this interaction between layers that is critical to system design and use, but is often lost in the one-component-at-a-time approach taken by most texts and courses.

1.2 Topic progression

This text takes a *bottom-up* approach. That is, it will begin with the most simple tools and concepts, and then build them up, one layer at a time, toward more complex and capable tools and concepts. Specifically, we should end with a complete (if simplified) computing system for which we can write complex programs and run many of them at once.

This approach is in contrast to the (predictably named) *top-down* approach. For that approach, the text would have begun with a complete, programmable computing system, and then explained its workings by incrementally delving down to the more simple and detailed layers. Each of these two approach has advantages and disadvantages. Examination of them will reveal why this book takes the bottom-up approach.

The top-down approach has the advantage that the motivation for each layer, as the presentation progresses, is clear. If you begin with the premise that you want a computer system for which you can write and run multiple, complex programs, then it is clear **why** one would begin the presentation with such a system. In order to understand how that system works, you then begin to delve down into the layers below it, carrying with you the clear motivation for understanding those layers.

The bottom-up approach lacks this ease of motivation. For example, when beginning with something so simple as *digital logic*, it is not immediately and intuitively clear to the reader **why** this topic is relevant. It is therefore the responsibility of the writer to explicitly provide the motivation for understanding this layer of the system, while deferring any understanding of how this topic relates to the entire system.

In spite of this problem with the bottom-up approach, is does have one wickedly important advantage: **there need not be any mysteries.** When beginning with the most simple layer, everything about that layer—its concepts, implementation, and applications—can all be explicitly explained. Then, with the advantage of fully understanding that lowest layer, the presentation can progress to the next layer, drawing upon the fully demystified layer below. In this way, when the presentation may reach its top level—a complete computational system—then every aspect of the system is explained and understood. No component is a mystery, and the interaction and function of all components can be seen.

In contrast, the top-down approach suffers from seemingly mysterious components, layers, and concepts until the very end is reached. When you begin with the complete system, none of its workings are understood. As you delve into the lower layers, you learn how each layer works, but you do so by making assumptions about the yet uninvestigated layers below, whose workings are a mystery. As you uncover the mystery of each layer, you must remember how it relates to the layers above it, and, *ex post facto*, piece together their relationship.

The purpose of this text and courses that follow it is to **demystify** the workings of computational systems. Therefore, it is more important that, during the journey of discovery, nothing is left as a mystery. At every step, you should see how the current topic is supported by all of the levels below it, with full knowledge of the concepts and the implementation of those levels. We therefore will follow the bottom-up approach, explicitly motivating each level, and building upon tools and concepts that have already been thoroughly developed.

1.3 Supporting software

[SFHK: Add this when the simulator/assembler/compiler are done.]

1.4 How to read this book

[SFHK: write me]

Chapter 2

Digital Logic

You have likely heard that computers work in "binary"—all 0's and 1's. But what does that **mean?** How can a collection of 0's and 1's represent numbers, or text, or pictures, or movies? How cna a program be made of nothing but 0's and 1's? That is, how can a group of 0's and 1's tell a machine **what to do** and **when to do it**?

We will answer some of these questions directly. For example, representing everyday numbers, such as $3,127$, is merely a matter of learning base-2 numeric representation, which is addressed in Section 2.1.1. However, more complex computing activities, such as using YouTube, are much more difficyult to grasp at the level of 0's and 1's. Doing so is akin to trying to understand a recipie for pumpkin pie by examining the interactions of the electrons, protons, and neutrons in a cookbook.

Although all modern computing is ultimately the manipulation of these 0's and 1's, computers cannot really be understood only at that level. Computer systems are better understood as a collection of *layers*, where the manipulation of binary numbers are at the bottom layer, and complex activities like Google Earth are at or near the top.

This chapter addresses that lowest level upon which all computing systems are built. It will build the foundation for representing and manipulating the 0's and 1's. Directly upon that foundation, we will be able to build circuits that perform basic arithmetic functions; later, we will use other, similar circuits that act as *memory* to store 0 and 1 values—*data*—for later.

2.1 A motivating example: Adding two integers

Of all the complex calculations and operations that a computing device can perform, among the most simple and intuitively understandable is *integer addition.* specifically, let us consider the addition of two *whole numbers*—non-negative integers $(0, 1, 2, \ldots)$. How can we make a machine that will add any two such numbers (e.g., $45,127 + 6,056$)?

2.1.1 Binary numbers

Knowing that we will later use devices that can manipulate 0's and 1's, we will begin by translating our problem into that form. That is, we want to represent the two values to be

decimal	**binary**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

Table 2.1: The integers 0 to 20 in both decimal and binary.

added (the *inputs*) as well as their resulting sum (the *outputs*), using only the digits 0 and 1.

Typically, people represent numbers using the set of digits from 0 to 9, also known as *base* 10 or *decimal notation*. Here, we need to represent numbers in *base* 2 or *binary notation*. We will use the *binary digits*—*bits*, for short—0 and 1.

When counting in decimal, we can imagine an odometer. When the 1's position contains the final digit, 9, and we then want to advance to the next number, the 1's position "rolls over" to a 0 again while the adjacent 10's position advances to a 1. More generally, each position advances from 0 to 9, and then it will return to 0 as the next, more significant position advances by one digit.

For binary numbers, imagine an odometer for which each position cycles not though the digits from 0 to 9, but only through the digits from 0 to 1. If the digit at a position is a 1, and we want to advance the counter to the next number, then that position "wraps around" to 0 and advances the digit at the next, more significant position. For example, Table 2.1 shows the whole numbers from 0 to 20.

To further develop an intuition about binary notation, consider how to decompose the digits of a decimal number into digits of differing significance. For example, $6,398$ can be decomposed into 6 thousands, 3 hundreds, 9 tens, and 8 ones. We can represent this decomposition as a collection of multiplications and additions, as shown in Table 2.2. Of course, the sum of the decomposed values shown in the bottom row is the original number.

thousands	hundreds	tens	ones
6	3	9	8
$\times$	$\times$	$\times$	$\times$
10^3	10^2	10^1	10^0
$\parallel$	$\parallel$	$\parallel$	$\parallel$
$6,000$	300	90	8

Table 2.2: Decomposition of the decimal number $6,398$.

thirty-seconds	sixteens	eights	fours	twos	ones
1	0	1	1	0	1_2
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
2^5	2^4	2^3	2^2	2^1	2^0
$\parallel$	$\parallel$	$\parallel$	$\parallel$	$\parallel$	$\parallel$
32	0	8	4	0	1_{10}

Table 2.3: Decomposition of the number $101101_2 = 45_{10}$.

Binary numbers are similar, but instead of there being positions for 1's, 10's, 100's, etc., there are positions for 1's, 2's, 4's, 8's, etc. Table 2.3 shows an example of decomposing a binary number. In this table, notice that, now that we are using two numerical notations, each value is tagged with a subscript of 2 or 10 to indicate whether the value is binary or decimal, respectively. Also notice that if you sum the decimal values into which the binary number is decomposed, you obtain the decimal reprensetation of the same number—that is, you will have converted the binary representation into a decimal one.

Much like each decimal number can be decomposed into a sum of power-of-10 units, binary numbers are decomposed into a sum of power-of-2 values. More importantly, **any** whole number can be represented in either decimal and binary—the two are equivalent. We should also note the difference between a *number* and its *representation*. No matter the representation of a number—45 (decimal), 101101 (binary), XLV (roman numerals)—the underlying number that these symbols represent is the same.

2.1.2 Addition, take I

In order to determine what concepts and devices we need to perform addition, we must first see how binary addition is performed. Let us review the simple mechanics of decimal addition. The two values to be added must be aligned by positions of significance: the 1's must be aligned, as must the 10's, 100's, etc. As an example, consider the additon of two four-digit decimal numbers: $x = 3,281_{10}$ and $y = 4,753_{10}$. We decompose the variables that represent these values to ease presentation of the arithmetic steps. Specifically, $x = (x_3, x_2, x_1, x_0)$, where $x_3 = 3$, $x_2 = 2$, $x_1 = 8$, and $x_0 = 1$. Similarly, $y = (y_3, y_2, y_1, y_0)$, and for this specific example, $y_3 = 4$, $y_2 = 7$, $y_1 = 5$, and $y_0 = 3$. Thus, our addition operation begins with the configuration shown in Table 2.4. On the left, we see the configuration for the addition of the specific values from our example; on the right, we see the generalized operation only on variables.

	3	2	8	1		x_3	x_2	x_1	x_0
+	4	7	5	3	+	y_3	y_2	y_1	y_0

Table 2.4: Configuration of addition for both example values (on the left) and generalized variables (on the right).

	1	1	0	0_{10}		c_3	c_2	c_1	c_0
	3	2	8	1_{10}		x_3	x_2	x_1	x_0
+	4	7	5	3_{10}	+	y_3	y_2	y_1	y_0
0	8	0	3	4_{10}	c_4	r_3	r_2	c_1	c_0

Table 2.5: The result of carrying out the addition algorithm on a specific set of values (on the left) and generalized variables (on the right).

Having formatted the input values as needed, summing x and y follows this algorithm:

1. Let k be the number of digits in the inputs (or, if they are not of the same length, prepend 0 digits to the shorter one to make it them the same length).
2. Let $i = 0$, where i indicates the position whose values to add.
3. Let $c_0 = 0$, which is the carry-in to the least significant column. Write that value above x_0.
4. Let $z_i = x_i + y_i + c_i$. This result, z_i, is a two-digit result that we decompose as $z_i = c_{i+1}r_i$.
5. Write r_i below x_i and y_i. Write c_{i+1} above x_{i+1} and y_{i+1}.
6. Increment i.
7. If $i < k$ then jump back to step 4.
8. Move c_k to the left of r_{k-1}, completing the result.

When we apply this algorithm, the result appears as shown in Table 2.5.

This same algorithm can be used for adding binary numbers as well. The only difference is in how r_i and c_i are determined—that is, how **three** binary digits (x_i, y_i, and c_i) are added.

For example, consider adding $x = 1011_2$ and $y = 0110_2$. In order to see how this addition progresses, we must consider the possible values that may occur in step 4. One advantage of binary is, given only two possible digits, one can often exhaustively list all of the possible inputs and outputs, known as a *truth table*. Here, we consider adding three one-bit values c_i, x_i, and y_i). Since each can only take the value 0 or 1, we can construct Table 2.6. Each of the eight possible sums is shown, in decimal, as z, and then as the two-bit binary value $c_{i+1}r_i$.

We can then lay out the numbers or variable using the same configuration that we do for decimal values, and then carry out the addition, using the values in Table 2.6 to determine the results in step 4. We see how the binary addition is performed in Table 2.7.

c_i	x_i	y_i	z	c_{i+1}	r_i
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Table 2.6: The truth table for adding three one-bit values.

	1	1	0	0_2		c_3	c_2	c_1	c_0
	1	0	1	1_2		x_3	x_2	x_1	x_0
+	0	1	1	0_2	+	y_3	y_2	y_1	y_0
1	0	0	0	1_2	c_4	r_3	r_2	c_1	c_0

Table 2.7: The result of carrying out the addition algorithm on a specific set of values (on the left) and generalized variables (on the right).

2.2 Propositional Logic I

We now know how to carry out a stepwise process—an *algorithm*—to add two binary numbers. However, we must develop a number of concepts in order to design a machine capable of automatically performing addition.

At the base of these concepts—indeed, at the bottom of all computational devices—is *propositional logic*. Specifically, it is the set of formal rules for handling everyday concepts like *and*, *or*, and *not*.

2.2.1 An example

As an example, consider the following statement:

> Tomorrow, if it is not raining, and if the temperature is at least 80°F, then we will go to the beach.

This statement defines two conditions that must be evaluated to determine whether a visit to the beach will occur. These two conditions are:

1. $A =$ *not raining*
2. $B =$ *temperature* $\geq$ 80°F

We can further decompose condition 1 as:

- $A' =$ *raining*

X	$\bar{X}$
F	T
T	F

Table 2.8: The truth table that defines the NOT operator.

X	Y	X AND Y
F	F	F
F	T	F
T	F	F
T	T	T

Table 2.9: The truth table that defines the AND operator.

- $A = \text{NOT}(A')$

So, we can express whether the conditions for a beach trip are met as:

$$C = A \text{ AND } B$$

A, B, and C are *Boolean variables*: they can take on one the values of *true* (T) or *false* (F). In this example, we must ultimately determine the value of C. To do that, we need to know both whether it is raining, and the temperature.

First, whether it is raining is itself a Boolean condition—either it is raining or it is not. Thus, we can assign the value T to A' if it is raining, and F to A' if it is not.

Having assigned one of those values to A', we can determine the value of A. To do so, we must define the logic operator NOT. It is a *unary* operator, which means that it takes a single Boolean variable as its input. We can define the relationship between the two possible input values that the single variable may be assigned and the possible output values of the operator with the truth table shown in Table 2.8. This table shows that the NOT operator simply inverts the input value: F becomes T; T becomes F.

Second, the temperature is **not** a Boolean value; it is, instead, a real-numbered value. However, we rely on the comparison operator *greater than or equal to* ($\geq$), which takes two real-valued inputs and produces a Boolean output value. Thus, B can be assigned the result of comparing the temperature to 80°F: either the temperature is at least 80°F ($B =$ T), or it is not ($B =$ F).

Third, given the ability to determine the Boolean values of A and B, we must determine how the logic operator AND functions to calculate C. Again, a truth table can be used to formally define the commonsense notion that $C =$ T if and only if $A = B =$ T. Table 2.9 shows this truth table.

Ultimately, to evaluate C, one must apply the comparison operator to determine B. Moreover, the NOT operator should then be applied to determine A. Finally, the AND operator must be applied to A and B in order to obtain the final result.

X	Y	X OR Y	X XOR Y	X NOR Y
F	F	F	F	T
F	T	T	T	F
T	F	T	T	F
T	T	T	F	F

Table 2.10: The truth table that defines the *inclusive or* (OR), *exclusive or* (XOR), and *neither nor* (NOR) operators.

X_3	X_2	X_1	X_0	R
F	F	F	F	T
F	F	F	T	T
F	F	T	F	F
F	F	T	T	F
F	T	F	F	F
F	T	F	T	T
F	T	T	F	F
F	T	T	T	F
T	F	F	F	T
T	F	F	T	T
T	F	T	F	T
T	F	T	T	F
T	T	F	F	T
T	T	F	T	T
T	T	T	F	F
T	T	T	T	F

Table 2.11: The truth table for an arbitrary, 4-input, Boolean logic operator.

2.2.2 Generalizing operators

Moving away from this simple example, we can generalize this type of calculation on Boolean values by defining other binary (that is, *two-input*) operators (of which AND is one). Consider Table 2.10, which is a truth table for the *inclusive or* (OR), *exclusive or* (XOR), and *neither nor* (NOR) operators. Notice that although each of these binary operators can be given a name that is a commonsense English word, that need not be the case. A logic operator can accept **any** number of Boolean value inputs, and its output values may follow no commonsense pattern. Consider the quanternary logic operator in Table 2.11, for which no simple name could be given. This example shows that any combination of input of output values listed in a truth table can define a valid operator.

2.2.3 Composing propositional functions

As we saw in the example in Section 2.2.1, propositional statements can be composed by applying additional propositional logic operators to any existing propositional expression. These compositions create new logic operations that can be expressed as truth tables of

A	B	A AND B	$C = A$ NAND B
F	F	F	T
F	T	F	T
T	F	F	T
T	T	T	F

Table 2.12: Composing AND and NOT yields the NAND operator.

their own. For example, consider composing the AND and NOT operators:

$$C = \text{NOT } (A \text{ AND } B)$$

We can evaluate C for all possible input values of A and B, yielding the truth table shown in Table 2.12.

We can also decompose known operators into compositions of other (usually "simpler") operators. Consider two of the binary operators from Table 2.10: NOR and XOR can both be expressed in terms of AND, OR, and NOT:

$$C = \text{NOT } (A \text{ OR } B) = A \text{ NOR } B$$

$$C = (A \text{ OR } B) \text{ AND } (\text{NOT } (A \text{ AND } B)) = A \text{ XOR } B$$

We will later see, in Section 2.6.3, that AND, OR, and NOT are sufficient to compose **any** logic function. For now, it is sufficient to note that any combination of input and output values define such a logic function that can be expressed using some composition of operators.

2.2.4 Proper notation

So far, we have written out the use of Boolean operators in a kind of longhand, writing AND, NOT, etc. explicitly in our expressions. However, we will heretofore use a more compact notation—one that resembles typical algebraic notation. Specifically:

- **Negation:** A bar over any variable or expression is the negation of that expression. That is:

$$C = \text{NOT}(A) \Rightarrow C = \bar{A}$$

- **Conjunction:** Rather than writing the operator AND to conjoin two expressions, they need only to be written next to one another, much like algebraic multiplication:

$$C = A \text{ AND } B \Rightarrow C = AB$$

- **Inclusive disjunction:** The inclusive OR operator is written using the plus sign, much like algebraic addition:

$$C = A \text{ OR } B \Rightarrow C = A + B$$

	0		0		1		1
+	0	+	1	+	0	+	1
0	0	0	1	0	1	1	0

Table 2.13: The four possible summations of two 1-bit values.

	x
+	y
z_1	z_0

Table 2.14: Generalizing the addition of two 1-bit values, using variables for each participating bit value.

- **Exclusive disjunction:** The exclusive XOR operator has no "normal" algebraic analog, and is written as a modified form of addition:

$$C = A \text{ XOR } B \Rightarrow C = A \oplus B$$

This notation can be composed just as the operators themselves are, thus sufficing to express other operators:

- NAND: $C = A \text{ NAND } B \Rightarrow C = \overline{AB}$
- NOR: $C = A \text{ NOR } B \Rightarrow C = \overline{A + B}$

2.3 Binary addition meets propositional logic

Now we can make good on putting propositional logic to use. More specifically, we can recast the arithmetic addition of binary numbers as logic expressions. For simplicity, let us begin with the addition of two 1-bit integers. There are only four possible combinations of two 1-bit values for addition, shown in Table 2.13.

We can generalize these additions by naming the 1-bit inputs x and y, and the 2-bit output value z. The relationships between these variable is shown in Table 2.14.

Let us recast the addition of x and y as a problem of propositional logic. Consider a binary value of 0 as thought it were a Boolean value F; likewise, the binary value 1 can be substituted for the Boolean T. thus, we can map the above listing of 1-bit addition values onto the truth table shown in Table 2.15. Examining the two output columns for z in this table, we see that each can be expressed as simple logic expressions:

$$z_1 = xy$$

$$z_0 = x \oplus y$$

Thus, this simple arithmetic calculation can be recase as a pair of propositional logic functions. For any pair of 1-bit numbers, the application of the AND and XOR logic operators will produce the arithmetic sum.

x	y	z_1	z_0
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	F

Table 2.15: The summation of two 1-bit values (x and y), producing a 2-bit result (z), cast as a truth table.

Addition of 1-bit numbers is so simple that its illustrative capacity as an example is limited. Let us consider the addition of 4-bit integers, which will provide a model for the addition of any two n-bit integers for any choice of $n > 1$. With two 4-bit values, it is impractical to enumerate all of the possible input combinations. However, we can use the model from Section 2.1.2, where the input values x and y are symbolically decomposed into their bits: $x = (x_3, x_2, x_1, x_0)$; $y = (y_3, y_2, y_1, y_0)$. Moreover, the addition produces both a set of result bits $r = (r_3, r_2, r_1, r_0)$ and a set of "carry" bits $c = (c_4, c_3, c_2, c_1)$. Specifically, recall the form, following standard pencil-and-paper addition of decimal integers shown in Table 2.4.

To cast this arithmetic problem instead as a series of propositional logic expressions, we must find a logical relationship between each of the bits of r and c in terms of the bits of x and y. Note that we can begin with the least significant input bits—x_0 and y_0—since adding those is merely adding two 1-bit values, which we have already examined. Therefore:

$$r_0 = x_0 \oplus y_0$$

$$c_1 = x_0 y_0$$

The remaining, more significant positions are slightly more complex. However, Table 2.6, which relates the input bits c_i, x_i, and y_i to the output bits r_i and c_{i+1} for position i, is exactly the correct truth table for this task, where this truth table substitutes the binary value 0 for the Boolean value F, and the binary value 1 for the Boolean value T. From that table, we can derive the following formulae for the output bits of each column.[1]

$$r_i = \bar{c}_i \bar{x}_i y_i + \bar{c}_i x_i \bar{y}_i + c_i \bar{x}_i \bar{y}_i + c_i x_i y_i$$

$$c_{i+1} = \bar{c}_i x_i y_i + c_i \bar{x}_i y_i + c_i x_i \bar{y}_i + c_i x_i y_i$$

Thus, the addition of our input bits can produce the carry and result bits by applying these logic functions. One needs only to carry out the evaluation of the propositional logic operators; yet the larger result is an arithmetic one.

[1]If the source of these formulae is a mystery, see Section 2.6.3, which presents a standardized approach for converting truth tables to logic functions.

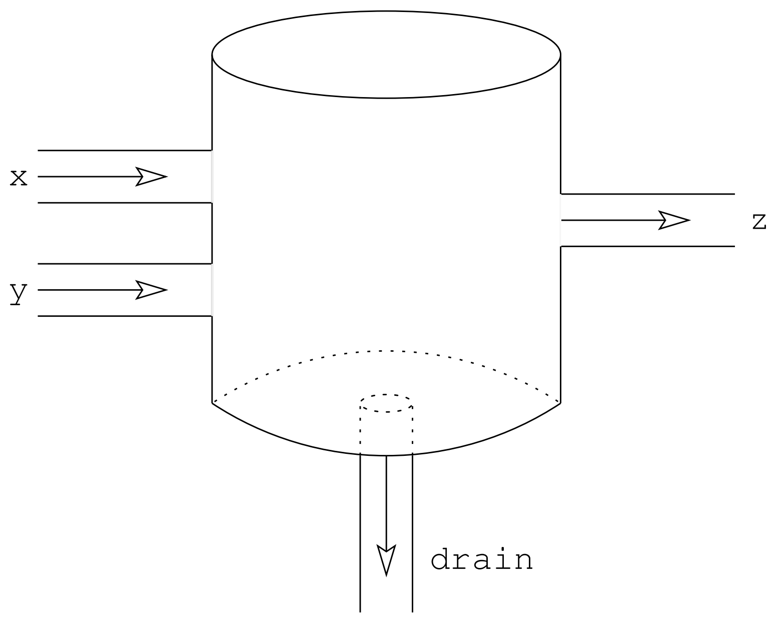


Figure 2.1: Schematic of an AND gate constructed from a drum and connected pipes through which water flows (or not).

2.4 Gates and Propositional Logic: Where the rubber meets the road

So far, we have seen how to decompose at least one simple arithmetic calculation into a collection of propositional logic operations. What has been unclear is **why** we are interested in this decomposition. We answer that question here: *gates*.

Our interest in propositional logic stems from our ability to make physical devices that carry out logic operations. These devices serve as the fundamental building blocks for all computation, because **all** computational expression can ultimately be decomposed into logic functions. Our goal here is to establish how these devices can be used to carry out **any** logic operation.

2.4.1 Water gates

AND gate: For a first attempt at creating devices that can carry out logic operations, we will employs devices that control the flow of water. how the water flows determines the result of some logic operation. As an example, let us begin with a water-flow device that carries out the AND operation, specifically: $z = xy$.

Figure 2.1 shows a drum to which there are four pipes of equal gauge attached. Each input and output is associated with one pipe each. For the inputs x and y, we can represent a value of 1 (or T) by pumping water in through their respective pipes. Likewise, we can represent 0 (or F) by **not** pumping any water in through either or both of these pipes. We also interpret a flow out through the z pipe as a 1, and no flow as a 0. The flow of water (or lack of it) trhough the drain pipe is not relevant, so we ignore it.

How does this drum carry out the logical AND operation? How does it behave as an "AND gate"? Let us consider all four possible input combinations:

- Case 00_2—$x = 0$ and $y = 0$: No flow enters the drum through the x and y pipes. Therefore, no flow exists the z pipe, implying that $z = 0$, which is the correct result for AND.

- Case 01_2—$x = 0$ and $y = 1$: No flow enters through the x pipe, but it **does** enter through the y pipe. The water that flows in through y then flows out of the drum through the *drain* pipe. Since the drain allows the water to exit the drum as fast as it enters, the water level never rises to the z pipe, and thus no water flows out through the z pipe. Thus, the device emits $z = 0$, which is correct.

- Case 10_2—$x = 1$ and $y = 0$: This case is analagous to the previous case 01_2.

- Case 11_2—$x = 1$ and $y = 1$: Flow enters the drum through both the x and y pipes. Because the water exits the *drain* pipe at half the rate of the total incoming flow, the water level will rise in the drum. After some time—a period knows as the *gate delay*—water will begin flowing out of the z pipe. This outflow represents the correct result for this case of $z = 1$.

Because of the gate delay for case 11_2, one must wait for at least that period of time after presenting new input flows (or lack of them) before the output of z is guaranteed to be correct. In any previous moment, the gate may still be altering the water level, and the output may not yet be stable. For different gate designs, the gate delay may have slightly different causes, and the duration of the delay will vary.

OR gate: Let us now consider how to construct an OR gate from this type of water drum. Constructing such a gate requires only that we swap the labels of the *drain* and z pipes from the water-drum AND gate. To see that this simple inversion works, consider the four possible input cases:

- Case 00_2—$x = 0$ and $y = 0$: Input flow from neither the x nor y pipes implies no output flow through the z pipe, implying correctly that $z = 0$.

- Cases 01_2 and 10_2—$x = 0$ and $y = 1$; or $x = 1$ and $y = 0$: The flow into the drum from one of the two inputs flows out of the z input, yielding the correct result of $z = 1$.

- Case 11_2—$x = 1$ and $y = 1$: The flow into the drum from both x and y pipes will cause both a flow out of the z pipe **and** a raising of the water level in the drum. That excess water will eventually flow harmlessly out of the *drain* pipe. The flow out of the z pipe implies the correct result of $z = 1$.

Notice that for this OR gate, the gate delay is much shorter. When new input flow is presented, one must wait only the time required for the water to fall down the drum and begin exiting the z pipe. Thus, thr output of this OR gate is guaranteed to be correct more quickly than the AND gate.

NOT gate: We are missing one critical capability—a gate that performs logical negation. We need a device that carries out $z = \bar{x}$. Notice that, unlike the other two gates that we have devised, this one must produce an output flow when there is **no** input flow. therefore, it must have a "power source"—an input flow that is not one the logical arguments to the operator. We leave it as an exercise to you, the reader, to device a water-drum NOT gate that…

- …when $x = 0$, the flow from the power source is directed to flow out of the z pipe, and …
- …when $x = 1$, the flow from both x and the power source is directed to flow out of *drain* pipes.

2.4.2 Electronic silicon gates

The water-drum gates presented in Section 2.4.1 are meant to show the simple construction of devices capable of performing propositional logic operations. Of course, these water drums would operate correctly, but slowly. Real computing devices use semiconducting[2] silicon transistors instead of drums, and flowing electricity instead of flowing water. Typically, a flow of +5 volts represents a binary value of 1, while 0 volts represents a value of 0. Other than these changes in substrate, the functions of the silicon gates and the water gates are quite similar, but the former are **much** faster.

The construction of the silicon gates is beyond the scope of this book. [SFHK: ADD REFERENCES TO DESCRIPTIONS OF TRANSISTORS, SILICON GATES, AND LITHOGRAPHY.]

2.4.3 Representing logic functions with gates

No matter the specific devices used, we will heretofore represent these logic gates as depicted in Figure 2.2. Critically, these gates can be composed analagously to the manner in which logic operators are algebraically composed. For example, Figure 2.3 shows an example of the composed logic function $z = w \oplus \overline{xy}$. The order in which a Boolean algebraic expression is evaluated is mirrored by the order in which input values flow into and through the gates, and ultimately to the circuit's output. In this example, one must first evaluate $\overline{xy}$; only then can the result of that evaluation be combined with w using the XOR operator to produce the result z. Analagously, the inputs lines x and y must have their values flow through the NAND gate first, then having the output of that gate flow as an input, along with the w line, into the XOR gate.

In closing this section, let us remember the high-level impact of composing gates in this manner: we can contruct a device that automatically computes the result of any propositional logic function for a set of given input values. Thus, any problem that can be expressed as a logic function can also be computed automatically by some arrangement of gates.

[2]That is, electrically conductive under some circumstances, but non-conductive under others.

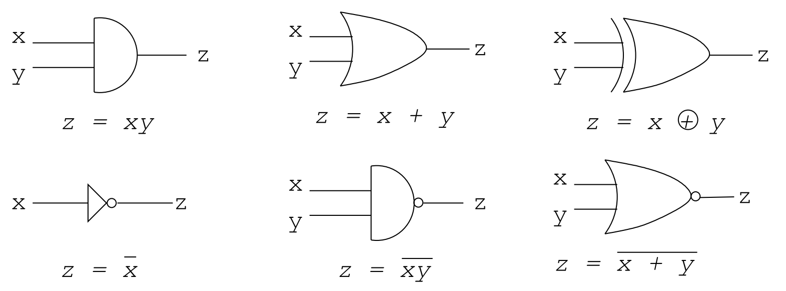


Figure 2.2: Substrate-neutral, graphical representations of gates that implement various logic operations.

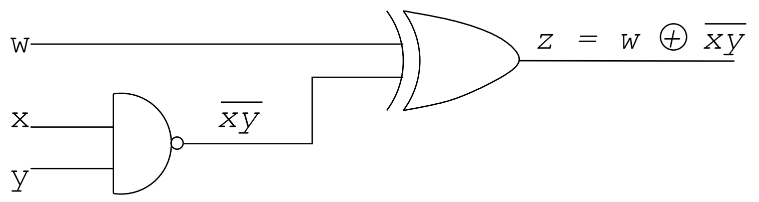


Figure 2.3: A gate-based circuit that implements the function $z = w \oplus \overline{xy}$.

2.5 Foreshadowing: Basic adder circuits

In Section 2.3, we saw how to express arithmetic addition of both 1-bit and 4-bit whole numbers as collections of logic functions. Here, we show hot o translate those logic functions into gate-based circuits, thus designing devices capable of carrying out these addition operations automatically.

2.5.1 A 1-bit half-adder

Recall that, in adding two 1-bit values—x and y—that two propositional logic functions are required to produce each of the values from the 2-bit result. Specifically, the result $z = (z_1, z_0)$ is determined by the functions:

$$z_0 = x \oplus y$$

$$z_1 = xy$$

Figure 2.4 a circuit to carry out this addition task, we can use the **same** two inputs to feed into gates that implement both logic functions and produce both output bits. Notice that any line that carries a flow can be split so that it can serve as an input to more than one gate. For example, x is divided at the dot labeled *x-split*, and then procides the same input value to both gates simultaneously. This dot is used to show the coinnection all line that meet at that point. Lines that cross without such a dot are not connected.

This *combinational circuit* implements 1-bit arithmetic addition. Present any two values as x and y, pause for the gate delay of both gates, and observe z_1 and z_0 to obtain the answer.

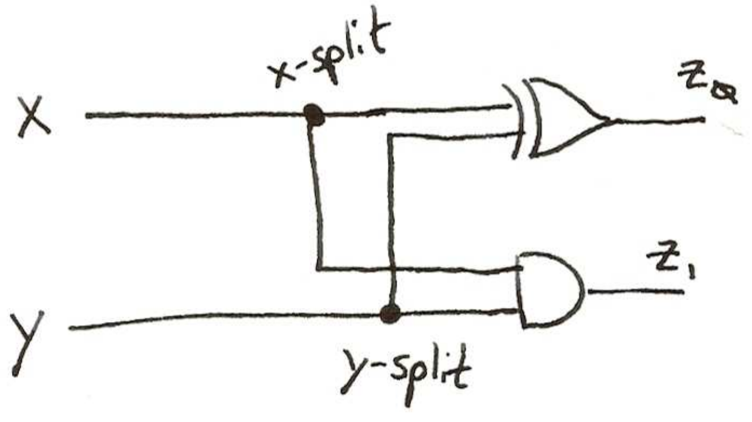


Figure 2.4: A half-adder that adds two 1-bit values.

2.5.2 A 4-bit ripple-carry adder

Adding 1-bit numbers provides a useful initial example, but it does not demonstrate well the more complex structures that can be devised to compute more complex problems. To provide a richer example, we examine the logic and circuitry for a 4-bit adder.

Recall that the addition of 4-bit values can be represented in the form shown in Figure 2.7. In that formulation, $x = (x_3, x_2, x_1, x_0)$ and $y = (y_3, y_2, y_1, y_0)$ are input values, while $z = (z_3, z_2, z_1, z_0)$ and $c = (c_4, c_3, c_2, c_1)$ are 4-bit output values produced by the process of adding x and y. More specifically, recall that adding one set of bits of the same significance is defined by the formulas:

$$z_i = \bar{c}_i\bar{x}_iy_i + \bar{c}_ix_i\bar{y}_i + c_i\bar{x}_i\bar{y}_i + c_ix_iy_i$$

$$c_{i+1} = \bar{c}_ix_iy_i + c_i\bar{x}_iy_i + c_ix_i\bar{y}_i + c_ix_iy_i$$

Figure 2.5 shows these two logic functions as combinational circuits. Collectively, we call these circuits a *full-adder*, since, unlike the half-adder, it incorporates the carry-in value. Furthermore, it produces the output for a single column's result (z_i) as well as the input for the next column (c_{i+1}). Now that we know how this full-adder is structured, we can draw it as a box with its three 1-bit inputs and its two 1-bit outputs, with no need to draw each individual gate.

Clearly, a full-adder is not, by itself, capable of adding two 4-bit values. However, by stringing together a sequence of four full-adders, we can construct a 4-bit adder. The key observation is that the carry-out of one full-adder may be connected to the carry-in of the next, thus properly carrying those value from a less significant column to the next more significant one.

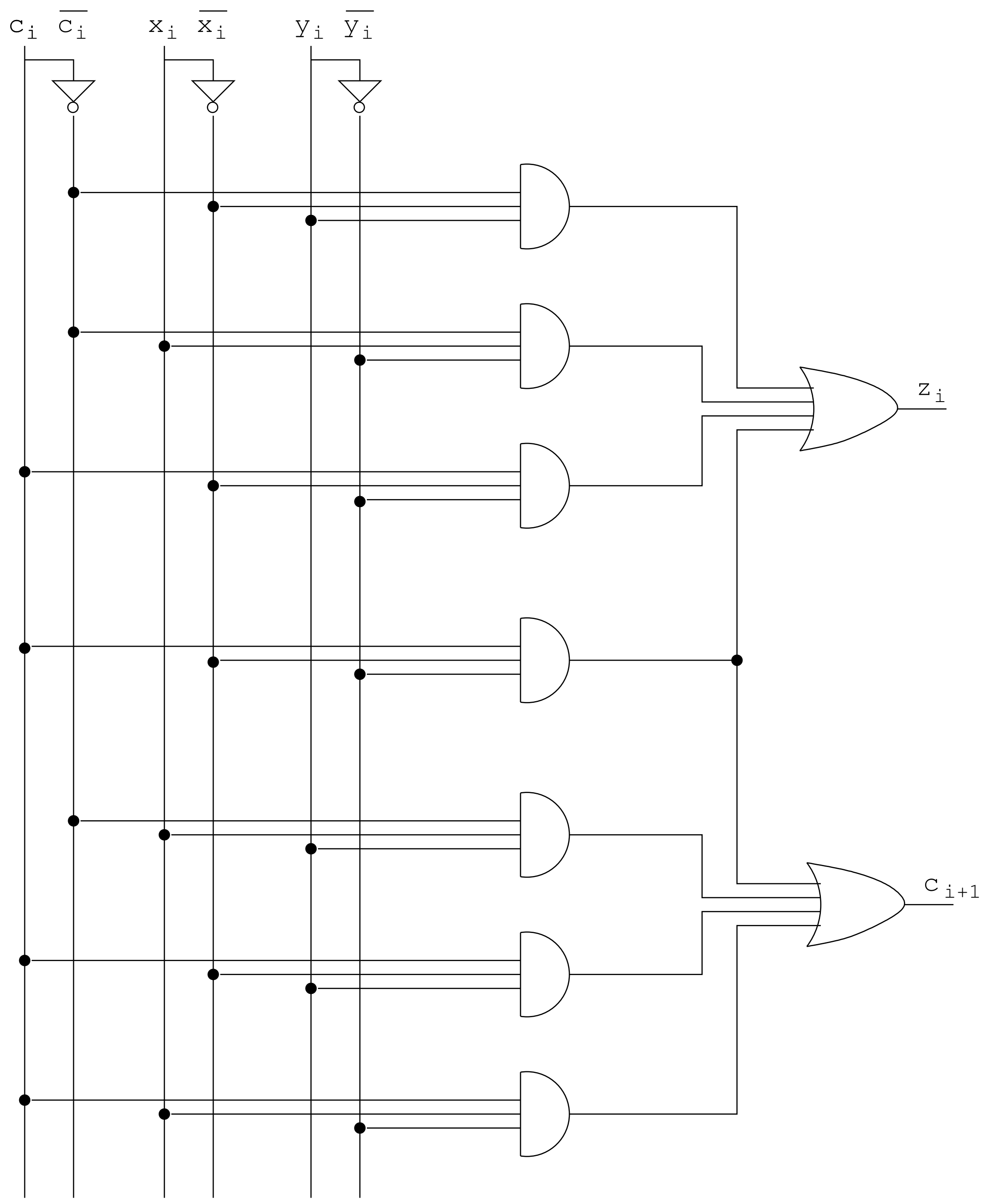


Figure 2.5: A full-adder that adds three 1-bit values.

Figure 2.6 shows how four full-adders can be arranged to add two 4-bit values. First, notice that c_0 seems to be a superfluous input. By setting $c_0 = 0$, we ensure that this extraneous input has no effect on the result. In the diagram, the triangle-like sequence of lines that serve as a source for c_0 is the symbol for connecting a line directly to *ground*—that is, this line will carry 0 *volts* to represent the input value of 0.

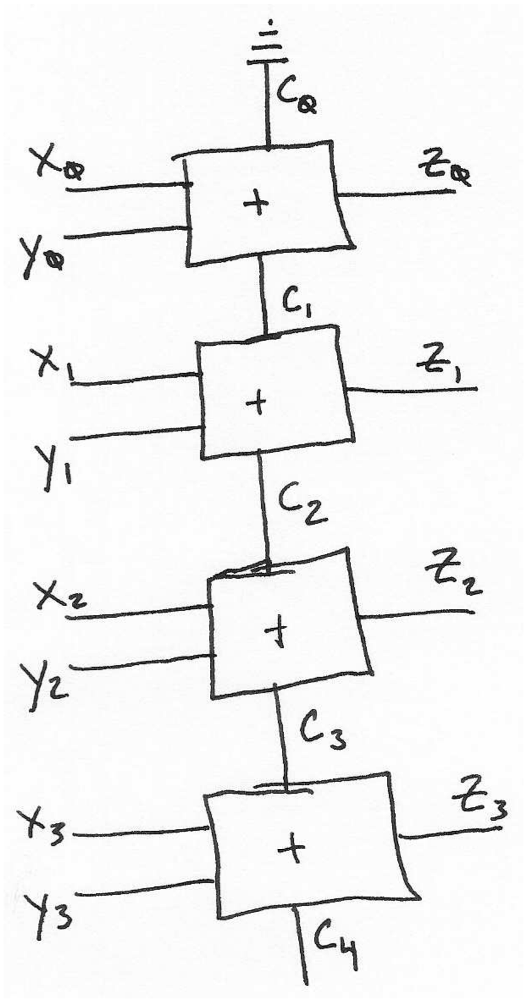


Figure 2.6: A 4-bit ripple-carry adder constructed from a chain of full-adders.

Second, notice that c_1, c_2, and c_3 are both input **and** output values. This dual role reflects directly the role of carry digits in pencil-and-paper addition—the "excess" of one column in *carried* into the next.

Third, recall the issue of *gate delay* (define in Section 2.4.1). If we present all of the inputs at time t_0, then we cannot trust the outputs of the least significant full-adder (z_0 and c_1) until the accumulated gate delay has passed. If we peek inside out full-adder circuit, we see each of the outputs is the result of waiting for a group of NOT gates, then a group of AND gates, and finally a multi-input OR gate (which may be implements as a collection of 2-input OR gates). Critically, all of the NOT gates do their work at the same time—*concurrently.* The same is true of the AND gates, and of the two OR gates. Thus, if a NOT gate's delay is d_{NOT}, an OR gate has delay of d_{OR}, and an AND gate has delay d_{AND}, then the total delay for both outputs of a full-adder is $d_{FA} = d_{NOT} + d_{AND} + d_{OR}$. Thus, at time $t_0 + d_{FA}$, we can trust that z_0 and c_1 are correct and will not change so long as the inputs x and y are unchanged.

Now for the catch: z_1 and c_2 cannot be trusted until the second full-adder has had sufficient time to process its inputs. However, although x_1 and y_1 may have been presented to their full-adder at time t_0, c_i will not be guanateed to be a correct input until $t_0 + d_{FA}$. Thus, z_1 and c_2 will not be reliable ouput values until $t_0 + d_{FA} + d_{FA} = t_0 + 2d_{FA}$.

This pattern continues: z_2 and c_3 are not reliable until time $t_0 + 3d_{FA}$; z_3 and c_4 become reliable at time $t_0 + 4d_{FA}$. Thus, the delay for the complete 4-bit adder is $4d_{FA}$. This delay is dictated by the *critical path*—a path through the circuit from some input to some output where the sum of hte gate delays along that path is maximal for that circuit. Here, the critical path from, say, x_0 to z_3 flows through a sequence of gates whose cumulative delay is $4d_{FA}$. This critical path is not unique, as others (e.g., starting at y_0 and ending at c_4) have identical gate delay sums.

We have created a circuit that, given time $4d_{FA}$ to do its work, will automatically add any two 4-bit whole numbers. This device is made of nothing but the simple gates that perform simple propositional logic operations, yet the device itself performs a task that is not itself a logic operation.

2.6 More propositional logic

We have now seen simple demonstrations of using logic operators (AND, OR, NOT) and devices that carry out those operations (*gates*) to automatically perform a task—integer addition—that is not itself propositional logic. Now that we have seen how logic can be used to perform computation, we must become somewhat more proficient in this type of logic.

2.6.1 Boolean algebra

When working with Boolean expressions, we can apply algebraic rules that are familiar to us. We can also apply a few rules that are valid for Boolean algebra, but not for traditional algebra. Specifically:

- **Identity:**
  - OR: $x + 0 = x$
  - AND: $x1 = x$
- **Commutativity:**
  - OR: $x + y = y + x$
  - AND: $xy = yx$
- **Associativity:**
  - AND: $x + (y + z) = (x + y) + z$
  - OR: $x(yz) = (xy)z$
- **Distribution:** $x(y + z) = xy + xz$
- **Zero:** $x0 = 0$

- **One:** $x + 1 = 1$

Finally, there are a pair of special rules known as *DeMorgan's Laws*:

$$\overline{xy} = \bar{x} + \bar{y}$$

$$\overline{x + y} = \bar{x}\bar{y}$$

With these rules, we can manipulate any Boolean expression. In particular, we can use these properties to establish the equivalence of expressions; we can also use the properties to simplify expressions. For example, consider the following transformation from one expression that performs the XOR operation to another:

$$\begin{aligned} & x \oplus y \\ &= (x + y)(\overline{xy}) \\ &= (x + y)(\bar{x} + \bar{y}) \\ &= \bar{x}x + x\bar{y} + \bar{x}y + \bar{y}y \\ &= 0 + x\bar{y} + \bar{x}y + 0 \\ &= x\bar{y} + \bar{x}y \end{aligned}$$

2.6.2 The generality of truth tables

No matter what combination of operators used to compose a logic function, a truth table can always be constructed that represents that function. For example, consider the following arbitrary, 3-input function:

$$z = \overline{(w \oplus \overline{xy})(\bar{w}\bar{x})}$$

This function has no intuitive or obvious "meaning." It is merely an arbitrarily constructed function. We can construct its truth table by carrying out the function on all possible inputs, as shown in Table 2.16.

This type of conversion from a logic function to a truth table is straightforward. However, it is less clear how to convert from an arbitrary truth table to a logic function. It is important to note that any different logic functions may produce the same truth table—that is, there are functions that are semantically equal but syntactically different. For example, consider these three simple functions:

$$\begin{aligned} z_A &= (x + y)(\overline{xy}) \\ z_B &= (x + y)(\bar{x}\bar{y}) \\ z_C &= \bar{x}y + x\bar{y} \end{aligned}$$

w	x	y	$\overline{xy}$	$\bar{w}\bar{x}$	$w \oplus \overline{xy}$	z
0	0	0	1	1	1	0
0	0	1	1	1	1	0
0	1	0	1	0	1	1
0	1	1	0	0	0	1
1	0	0	1	0	0	1
1	0	1	1	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	1	1

Table 2.16: Translating an arbitrarily chosen, 3-input logic function into a truth table by evaluating the components of the function and adding them to the truth table, building towards the final function.

Superficially, these are three different 2-input functions. However, they all implement the XOR operation, producing the same outputs for any given input.[3] Thus, for a given truth table, an arbitrary number of syntactically different functions will produce the correct semantic result. How can one systematically convert any truth table into a corresponding logic function?

2.6.3 Normal forms

Consider our previous, arbitrarily chosen, 3-input logic function and the corresponding truth table shown in Table 2.16. We can use that truth table to produce a function of a specific syntactic form by following a set of mechanistic steps. To do so, we begin by listing the input values of w, x, and y that results in $z = 1$. Table 2.17 shows the six cases fitting that description. Along with each such input combination, the table also shows how each such set of input values can be translated into a *conjunctive term*—that is, expression of w, x, and y such that the three are combined by the AND operator. Moreover, for each such term, if the variable in question for that case should be a 1, then the variable is written in an unmodified, "positive" form; if the variable should be a 0, then it should be inverted by the NOT operator before being conjoined with the other variables. For example, in the case where $z = 1$ when $w = 0$, $x = 1$, and $y = 0$, then the conjunctive term we seek is $\bar{w}x\bar{y}$.

Given these six conjunctive terms (in this example), we then *disjoin* them—that is, combine them with the OR operator. Theresult is a complete expression of z:

$$z = \bar{w}x\bar{y} + \bar{w}xy + w\bar{x}\bar{y} + w\bar{x}y + wx\bar{y} + wxy$$

For any assignment of values to the input variables that should yield $z = 1$, **exactly one** of the conjoined terms will evaluate to 1. Since all of those conjoined terms are then disjoined to form the whole expression, having any one conjoined term evaluate to 1 is sufficient for the entire expression also to evaluate to 1. For example, consider the evaluation of $w = 1$,

[3]Try constructing a truth table that evaluates all three functions to convince yourself of their semantic equivalence.

w	x	y	conjunctive term
0	1	0	$\bar{w}x\bar{y}$
0	1	1	$\bar{w}xy$
1	0	0	$w\bar{x}\bar{y}$
1	0	1	$w\bar{x}y$
1	1	0	$wx\bar{y}$
1	1	1	wxy

Table 2.17: For an example 3-input function, the six combinations of input values that cause a result of $z = 1$ and the corresponding conjunctive terms.

$\bar{w}x\bar{y}+$	$= \bar{1}1\bar{0}+$	$= 011+$	$= 0+$
$\bar{w}xy+$	$\bar{1}10+$	$010+$	$0+$
$w\bar{x}\bar{y}+$	$1\bar{1}\bar{0}+$	$101+$	$0+$
$w\bar{x}y+$	$1\bar{1}0+$	$100+$	$0+$
$wx\bar{y}+$	$11\bar{0}+$	$111+$	$1+$
wxy	110	110	0
			$= 1$

Table 2.18: Evaluating the example expression where $w = 1$, $x = 1$, and $y = 0$, where exactly one conjunctive term evaluates to 1, causing the whole disjoined set of terms to evaluate to 1 as well.

$x = 1$, and $y = 0$ shown in Table 2.18. Each row in this table shows one conjunctive term from the disjoined six, and then shows how each is evaluated to each a final result.

As you might expect, for any assignment of values to w, x, and y where the result is $z = 0$, **none** of the conjunctive terms will evaluate to 1. Thus, the disjoining of a collection of 0 results is always 0, yielding the correct answer for that case.

This form of expression—a collection of conjoined terms, with each such term combining each input or its inversion, with all such terms disjoined—is the *disjunctive normal form (DNF)*, or the *or-of-ands* form. As we have just seen, **any** truth table can be converted into a DNF expression. A corrolary of this observation is that three logic operations—AND, OR, and NOT—are sufficient for composing any logic function. Thus, the gates that perform this three operations are also sufficient to implement a device that carries out any logic function.

Note that there is also a *conjunctive normal form (CNF)*: a set of *disjunctive terms*, each of which contains all of the input variables (or their inversion), all conjoined to form an expression. For example, the following function is expressed in CNF:

$$z = (\bar{w} + x + \bar{y})(w + \bar{x} + \bar{y})(w + \bar{x} + y)$$

Because this form is not as intuitively derived from a truth table as DNF expressions, we will not use it. However, any DNF expression can, through algebraic manipulation, be transformed into a CNF expression and *vice versa*.

x	$\overline{xx} = z$
0	1
1	0

Table 2.19: The truth table for $z = \overline{xx} = \bar{x}$, showing NAND configured to perform the NOT operation.

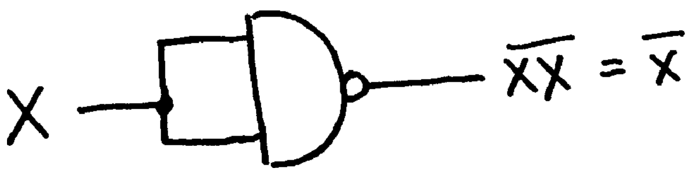


Figure 2.7: A combinational circuit using only NAND gates to carry out the NOT operation.

2.6.4 Minimal operators

The observation that AND/OR/NOT are sufficient to express any logic function poses the following question: *Do we need all three of those operators?* Can we compose any logic function from just two, or perhaps even one logic operator? Note that to prove that fewer operators are sufficient, we need only show how to compose the one or two operators such that they perform AND, OR, and NOT, whose sufficiency we have already demonstrated.

Let us consider the NAND operator, defined by the truth table shown in Table 2.12, and defined by the function $z = \overline{xy}$. Then let us see how this one operator can be composed with itself to carry out NOT, and, and OR.

not : Although NAND is a *binary operator* (that is, it operates on two 1-bit inputs), NOT is a *unary operator* (it operators on one 1-bit input). Thus, consider using the same variable for both of a NAND operator's inputs:

$$z = \bar{x} = \overline{xx}$$

That is, x is NAND'ed with itself. Since x is the only input, we can use the simplified truth table shown in Table 2.19 to prove that $\bar{x} = \overline{xx}$, and thus that NAND can be used to invert a value. Figure 2.7 shows the combinational circuit, using only a NAND-gate, that performs the NOT operation.

and : Now that we have established that NAND can perform the NOT operation, we can use both operators in determining how to perform AND using only NAND. The following expressions, as well as the truth table shown in Table 2.20, show the equivalence:

$$z = xy = \overline{\overline{xy}} = \overline{\overline{xy}\,\overline{xy}}$$

x	y	$\overline{xy}$	$z = \overline{\overline{xy}} = xy$
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.20: The truth table for $z = xy = \overline{\overline{xy}}$, showing NAND configured to perform the AND operation.

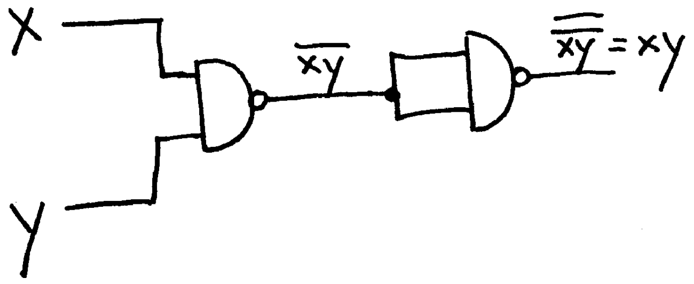


Figure 2.8: A combinational circuit using only NAND gates to carry out the AND operation.

x	y	$\bar{x}\bar{y}$	$\overline{\bar{x}\bar{y}} = x + y$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

Table 2.21: The truth table for $z = x + y = \overline{\bar{x}\bar{y}}$, showing NAND configured to perform the OR operation.

Again, we can prove this equivalence with the truth table in Table 2.20; we can also see, in Figure 2.8 the combinational circuit that implements AND using only NAND gates.

or : Finally, we have the ability, using only NAND gates, to perform NAND, AND, and NOT. With these three, we seek to perform the OR operation. The formula below shows the equivalence between an arrangement of NAND operations and the OR operation. Table 2.21 and Figure 2.9 show the truth table and combinational circuit, respectively, that reflect this formula.

$$z = x + y = \overline{\overline{x + y}} = \overline{\bar{x}\bar{y}}$$

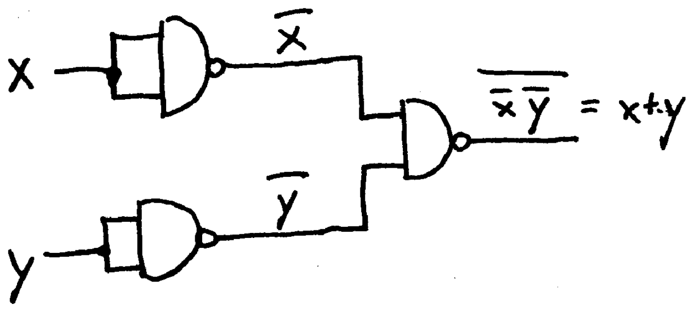


Figure 2.9: A combinational circuit using only NAND gates to carry out the OR operation.

The big picture: These three simple circuits, using only NAND gates, are sufficient building blocks to implement any logic function. Moreover, we will see that they are therefore sufficient to implement any computation. NOR is likewise sufficient; we leave it to the reader to develop AND, OR, and NOT expressions and circuits based solely on the NOR operator.

2.7 Circuit simplification

We can see from the 4-bit adder example that circuit structures for any non-trivial task are complex. For those constructing circuits by hand, each additional gate adds not only labor but also an opportunity for a miswiring error. For mass-produced circuits, each additional gate occupies valuable chip space, consumes more energy, and produces more waste heat.

Therefore, we have many reasons to simplify our circuits. Doing so requires simplifying the corresponding logic—that is, for a given truth table, finding the correspoinding logic function that uses the fewest logic operators. Since each oiperator is implemented by a gate, these minimal functions yield the smallest circuits.

2.7.1 Karnaugh maps for 2-input functions

Normal forms provide simple, mechanistic approaches top translating a truth table into a logic function. However, normal form functions are rarely minimal. Here, we examine an alternative method for t ranslating truth tables into logic functions—a method that, by finding certain patterns, allows us to create a minimal logic function.

Consider the NAND function, whose 2-input truth table is shown in Table 2.12. We can easily convert this truth table into a DNF logic function[4]:

$$z = \bar{x}\bar{y} + \bar{x}y + x\bar{y}$$

[4]Although you can likely imagine a simpler function that this DNF form, note that your ability to do so stems from the inherent smallness and simplicity of any 2-input function. Suspend your intuitions about simpler functions for the purpose of this example; later examples will be more difficult, where you not immediately identify a simpler form.

Figure 2.10: The layout of a Karnaugh map for the 2-input NAND function.

Figure 2.11: A 2-input Karnaugh map, with rectangles encapsulating simplified conjunctive terms.

In order to devise a syntactically shorter function, we begin by writing our truth table in a special form, known as a *Karnaugh map*, as shown in Figure 2.10. For this map, the vertical axis is labeled with the possible values of x ($\bar{x} \Leftrightarrow x = 0$, $x \Leftrightarrow x = 1$), while the horizontal axis is labeled analagously with the possible y values. Then, each position on the map corresponds to some specific choice of values for the two input variables. Moreover, that choice of values for the input variables corresponds to an output value for z. That z value should be placed in the aofrementioned location in the map, thus showing the function's result for those inpuit values associated with that position.

Next, one must find **rectangles of 1's**. A rectangle that encoloses 1's with no 0's included identifies a simplified conjunctive term that will be part of a completed expression for the entire function, where the conjunctive term has factored out superfluous variables. Figure 2.11 shows the rectangles for a Karnaugh map of the NAND function. These rectangles can be horizontal or vertical, and they may overlap. Using this example, we can examine what each of these rectangles represent:

- α**:** This rectangle encapsulates two 1's, and thus two input patterns that cause the function to evaluate to 1, specifically $x = 0, y = 0$ and $x = 0, y = 1$. Notice that for both of these cases, $x = 0$. Graphically, we see that commonality in that α spans $\bar{y}$ and y columns, but stays within $\bar{x}$ row.

- β**:** Similarly, the two 1's encapsulated by this rectangle are produced by the pair of input patterns $x = 0, y = 0$ and $x = 1, y = 0$. This rectangle spands the rows $\bar{x}$ and x, but stays within the column $\bar{y}$, thus graphically representing that $y = 0$.

Each rectangle represents a single conjunctive term that is simplified—that is, unlike a conjunctive term in DNF, it may not contain every input variable. Specifically, each rectangle represents a term composed of the common input variables (or their inversions), with those input variables that are not in common excluded. For example, α represents the term $\bar{x}$. We can obtain this same simplification by applying Boolean algebraic rules to a DNF expression of the input patterns to which α corresponds. That is:

$$\alpha = \bar{x}\bar{y} + \bar{x}y$$

If we factor out $\bar{x}$, apply the disjunctive ones property, and then the conjunctive identity property, we get:

$$\alpha = \bar{x}(\bar{y} + y) = \bar{x}1 = \bar{x}$$

Thus, α here represents the term $\bar{x}$, which is a substantial simplification of $\bar{x}\bar{y} + \bar{x}y$. Similarly, β can be shown to represent $\bar{y}$ as follows:

$$\beta = \bar{x}\bar{y} + x\bar{y} = \bar{y}(\bar{x} + x) = \bar{y}1 = \bar{y}$$

The terms represented by the rectangles can then be disjoined, thus forming the complete, simplified expression:

$$z = \bar{x} + \bar{y}$$

Notice that the two rectangles overlapped at $\bar{x}\bar{y}$. Such overlaps are valid. For the input pattern corresponding to an overlapping region, the consequence is that more than one of the conjunctive terms in the final expression will evaluate to 1. Here, when $x = 0, y = 0$, then $\bar{x} = \bar{y} = 1$. But since those terms are disjoined in z, whether one or both evaluates to 1 is irrelevant—the end result is that $z = 1$.

Thus, the rectangle α here ~~ϕ~~ represents the term $\bar{x}$, which is a substantially simplification of $\bar{x}\bar{y} + \bar{x}y$. Similarly, the rectangle β spans the terms that cancel, and is contained by the term that remains:

$$\bar{x}\bar{y} + x\bar{y} = \bar{y}(\bar{x} + x) = \bar{y}(1) = \bar{y}$$

The ~~te~~ terms represented by the rectangles can then disjoined, thus forming the complete, simplified expression:

$$z = \bar{x} + \bar{y}$$

~~All of t~~

Ⓑ Karnaugh maps — 3 inputs

The ~~two~~ 2-input example above shows most of the basics of using Karnaugh maps to simplify logic functions, but 2-input functions are so simple that the value of the Karnaugh map is not obvious. Let us then ~~is~~ see how to use this approach for a 3-input function. Specifically, consider this arbitrary 3-input truth table:

w	x	y	z
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Now consider the Karnaugh map for this same 3-input function:

	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
$\bar{w}$	1	1	1	1
w	0	1	0	1

First, ~~notice to~~ let us write this function in DNF:

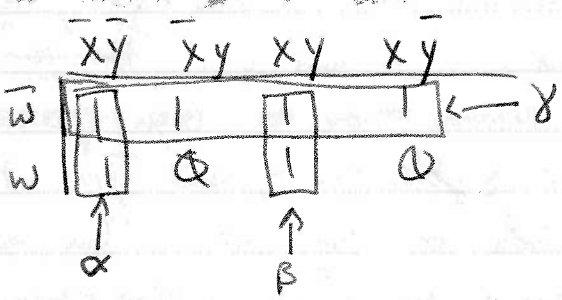
$$z = \bar{w}\bar{x}\bar{y} + \bar{w}\bar{x}y + \bar{w}x\bar{y} + \bar{w}xy + w\bar{x}y + wx\bar{y}$$

Second, ~~to~~ before simplifying this lengthy expression, ~~let~~ we should consider how the Karnaugh map is configured. We have only ~~two~~ dimensions to the table, but we have three inputs to represent. So, the ~~horizontal axis~~ column labels specify not one, but two of the input variables — $\underline{x}$ and $\underline{y}$. ~~The row label~~ Notice that there are four columns — one ~~for~~ each possible combination of ~~x, xy~~ $\underline{x}$ and $\underline{\bar{x}}$, with y and $\bar{y}$. Meanwhile, $\underline{w}$ and $\underline{\bar{w}}$ are the labels for the rows. The table therefore has $\underline{2 \times 4 = 8}$ entries — one for each possible combination of input value ~~for~~ ~~w~~ w, x, and y.

The is one additional, critical aspect of this table's configuration. Notice the progression of the column labels. Typically, the ~~countin~~ binary counting sequence follows the integers: 00 ($\bar{x}\bar{y}$); 01 ($\bar{x}y$); 10 ($x\bar{y}$); 11 (xy). However, the column labels follow a different progression: 00 ($\bar{x}\bar{y}$); ~~1~~ 01 ($\bar{x}y$); 11 (xy); 10 ($x\bar{y}$).

Why the altered progression? ~~Karnaugh maps req~~ For Karnaugh maps to work — that is, for the rectangles ~~to provide~~ [illegible] correctly to identify factorizations of superfluous terms — then each ~~label~~ column label must have ~~at one~~ a term in common with the labels on each adjacent term. That is, from one column to the next, the labels must have either an $\underline{x}$, an $\underline{\bar{x}}$, a $\underline{y}$, or a $\underline{\bar{y}}$ in common. ¶ The standard counting sequence lacks this property. Specifically, if $\underline{\bar{x}y}$ and $\underline{x\bar{y}}$ are adjacent to one another, as the would be for the standard sequence, then those two columns have no term in common. In contrast, notice that the sequence of column labels on the example Karnaugh map has this property; every column has a term in common with those adjacent.

Third, let us use this Karnaugh map to simplify the algebraic expression:



Again, we find rectangles of $\underline{1}$'s. Note that each rectangle must have a power-of-2 ~~number~~ size ~~for~~ for each dimension. That is, the rectangles may be 1×2, 2×1, ~~4×~~ 1×4, 4×~~4~~1, or 2×2. A 1×3 ~~or~~, 3×1, 2×3, or 3×2 rectangle is invalid [illegible] because the $\underline{1}$'s in such a rectangle would not ~~have any~~ all have terms in common. However, the valid rectangle sizes contain $\underline{1}$s all ~~of~~ of which will have terms in common. For example, consider our three rectangles above:

α) This rectangle spans $\overline{w}$ and w, so that term can be eliminated. The two 1's contained by α ~~both hav~~ are contained in the column $\overline{x}\,\overline{y}$, and thus have those terms in common. This rectangle represents ~~the~~, in ~~our~~ the simplified expression, the conjunctive term $\overline{x}\,\overline{y}$.

β) Like α, $\overline{w}$ and w can be eliminated. Here, though, ~~x~~ ~~and y are the terms~~ xy is the column that contains this rectangle, so β represents: xy.

γ) This 4×1 rectangle spans all combinations of x, $\overline{x}$, y, and $\overline{y}$. However, the retangle is contained by the row $\overline{w}$, and thus γ represents: $\overline{w}$. Notice that larger rectangles imply a larger number of ~~ef~~ factored terms, and thus shorter ~~ex~~

Disjoin the three ~~terms~~ conjunctive terms, and our simplified form is:

$$z = \overline{x}\,\overline{y} + xy + \overline{w}.$$

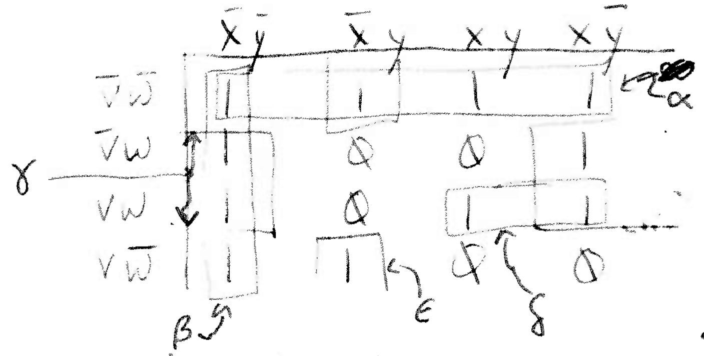
ⓒ 4-input Karnaugh maps

Now that we see how to arrange two input variables along a single dimension of a Karnaugh map, we can apply the same approach to both axes of the map, allowing for four input variables. ~~I~~ Let us ~~agn~~ again consider an arbitrary truth table and then its corresponding Karnaugh map.

v	w	x	y	z
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

The truth table represents no immediately or intuitively re-cognizable function. Its DNF is lengthy:

$$
\begin{aligned}
z = {} & \bar{v}\bar{w}\bar{x}\bar{y} + \\
& \bar{v}\bar{w}\bar{x}y + \\
& \bar{v}\bar{w}x\bar{y} + \\
& \bar{v}\bar{w}xy + \\
& \bar{v}w\bar{x}\bar{y} + \\
& \bar{v}wx\bar{y} + \\
& v\bar{w}\bar{x}\bar{y} + \\
& v\bar{w}\bar{x}y + \\
& vw\bar{x}\bar{y} + \\
& vwx\bar{y} + \\
& vwxy
\end{aligned}
$$



	$\bar{x}\bar{y}$	$\bar{x}y$	xy	$x\bar{y}$
$\bar{v}\bar{w}$	1	1	1	1
$\bar{v}w$	1	0	0	1
vw	1	0	1	1
$v\bar{w}$	1	1	0	0

The 4-input map is configured much like the 3-input, except that ~~both~~ each dimension of the map is labeled with two input variables: ~~to~~ v and w for the rows, and x and y for the columns. Again, each row or ~~tabel~~ column label must have ~~at least~~ one term in common with ~~t~~ its ~~advace~~ adjacent labels. So, both rows and colums follow the special progession: 00, 01, 11, 10.

In this example, we find five rectangles: two 4×1 rectangles (α and β); one 2×2 rectangle (γ); and two 2×1 rectangles (δ and ϵ). Again, each rectangle encapsulates a set ~~of 1's~~ ~~that represent~~ input values that evaluate to 1 for this function, and that can be expressed as a single conjuctive term. So:

α) This rectangle spans every column, and thus every combination of x, ~~z~~ y, and their negations. However, it is also contained within the row $\overline{v}\,\overline{w}$. Thus, if $v=0$ and $w=0$ then $z=1$ irrespective of the values of x and y. So, this rectangle represents the term: $\overline{v}\,\overline{w}$.

β) ~~Like α~~ This rectangle is analagous to α. Here, the map shows that, if $x=0$ and $y=0$, then $z=1$ no matter the values of ~~x~~ v and w. Thus: $\overline{x}\,\overline{y}$.

γ) This 2×2 rectangle is an interesting one in that it "wraps around" from one edge of the map to the other. ~~This~~ That is, column $x\overline{y}$ is adjacent not only to xy, but also to $\overline{x}\,\overline{y}$. ~~Likewise~~ ~~Thus,~~ γ spans $\overline{v}$ and v (vertically) as well as $\overline{x}$ and x (horizontally by ~~wr~~ wrapping around.) The rectangle is contained within the w ~~columns~~ rows and the $\overline{y}$ columns — that is, if $w=1$ and $y=0$, the values of v and x have no effect, since $z=1$ in all such cases. So: $w\overline{y}$.

δ) This 2×1 rectangle is much like those of smaller maps. It is contained by the row $\underline{vw}$ and within the colums $\underline{xy}$ and $\underline{x\bar{y}}$. Thus, the $\underline{1}$'s in this rectangle occur when $\underline{v=w=x=1}$, irrespective of the value of y. So: $\underline{vwx}$.

ε) This 2×1 rectangle wraps around vertically, showing that $\underline{v\bar{w}}$ and $\underline{\bar{v}\bar{w}}$ are adjacent rows because of their common term $\underline{\bar{w}}$. This rectangle is also contained within the column $\underline{\bar{x}y}$. So, all $\underline{1}$'s is this rectangle occur when $\underline{w=0}$, $\underline{x=0}$, and $\underline{y=1}$, regardless of the value of $\underline{x}$: $\underline{\bar{w}\bar{x}y}$.

When we disjoin the terms implied by these five rectangles, we obtain the simplified expression:

$$z = \underset{\alpha}{\bar{v}\bar{w}} + \underset{\beta}{\bar{x}\bar{y}} + \underset{\gamma}{w\bar{y}} + \underset{\delta}{vwx} + \underset{\epsilon}{\bar{w}\bar{x}y}$$

(D) Karnaugh maps are not easily extended beyond four inputs. For ~~a larger~~ higher arity functions, there are two options, neither of which we explore in depth:

(1) Algebraic simplification: Like any algebra, the various properties of Boolean algebra allow expressions to be ~~so~~ manipulated and simplified. However, this method of simplification requires practice and creativity, and thus is beyond the scope of this book.

② Circuit simplification algorithms! More advanced techniques, including those that perform extensive but intelligent searches of the possible solutions, ~~have been~~ are performed by ~~caref~~ complex software. The most advanced of these tools are proprietary trade secrets. Because our goal is to understand how a simple system can be constructed, and not how complex systems are optimized, this book will not address such tools.

(2) MEMORY

I. Motivation

~~Ex~~ ~~We have~~ ~~drawn~~ created ~~circuits~~ For each circuit that we have created, we have assumed that some input values—some collection of $\underline{1}$'s and $\underline{0}$'s—would be provided as the inputs ~~to~~. From whence come these input values? So far, there are two choices:

1. ~~Direct input~~ External value: By connecting an input to a flow (e.g., a water pump for water-based gates; a 0V ~~&~~ or +5V electrical source for silicon gates). Thus, the value is provided by something external to the circuit and the logic it represents.

2. Output+input chain: The output of some ~~other~~ other gate may be used as the input to another — a fundamental characteristic that allows ~~gates to~~ us to implement composed logic functions using gates. Here, the value is provided internally, as one component of a larger circuit.

So far, all of our logic and circuits have been combinational, in that they are a functional, many-to-one mapping of $\underline{n}$ input values to $\underline{1}$ output value. That value is computed by a direct flow from the inputs, through the gates, and, after the delay dictated by the critical paths, to the outputs.

~~We introd~~ This chapter introduces another possibility ~~so~~: ~~an~~ input values provided by memory elements. Here, some ~~Boolean~~ binary value will be recorded ~~by a~~ at some earlier time, and then used as an input by some gate. That input value will persist ~~until~~ until a new value is recorded in that same memory element.

In order to see why ~~the~~ we would want such memory elements, we will consider, in Section ②.I.Ⓐ, ~~a simple~~ the most simple of sequential logic circuits. This type of design is ~~cove~~ addressed more thoroughly in Chapter ③, which is why we must address memory elements themselves first. Note, however, that memory elements will be just as fundamental to computation as ~~com~~ processing itself! Without data ~~and~~ stored in memory elements, stepwise computing as we know it is impossible

Ⓐ ~~II.~~ A~~n~~ ~~motivating~~ example: The toggle

Imagine that you want a device that will ~~remember one of two~~ alternate between one of two possible states. For example, ~~three~~ two small children are sharing a toy, and you want a device that remembers ~~a~~ which of the two children's turn it is. Or, for those familiar with ~~the~~ collegiate basketball, you want to ~~make the~~ create the core of the possession arrow indicator. For both~~a~~ devices, there is a single ~~but~~ button that you would press to toggle the state of the device: from Jeffrey's turn to Ephraim's turn, ~~or~~ or vice versa; from ~~the~~ Amherst's possession at the next jump-ball to Williams', or vice versa.

This device can also be viewed as a 1-bit counter. ~~Imagi~~ All odometers ~~represent a fi~~ employ a fixed number of digits. Thus, their range is bounded, and eventually, ~~it~~ they must "roll-over", resetting to ~~Ø~~. zero. A 1-bit counter is an odometer with a single binary digit. After incrementing from Ø to 1, the next incrementation must force a reset to Ø ~~again~~.

~~Therefore, to~~ Given this description of a ~~one~~ device ~~that~~ whose output oscillates between Ø or 1, how ~~to~~ can we structure a circuit to implement the device? ~~In p~~ We must imagine that the one external input — the button that toggles the ~~s~~output — is provided as an ~~inp~~ line that carries Ø when the button is not being pressed, and 1 when it is. This input will serve as a trigger to a memory element ~~& which, open~~. This memory element will have ~~two inputs~~ the following inputs and outputs:

- ~~In Data input:~~
- Data ~~output~~ input: The value that the memory element will adopt when the trigger is activated.
- ~~Da~~ Trigger input: When this input is Ø, the data input is ignored. When it ~~is~~ becomes 1 [footnote this detail], the ~~output~~ memory element adopts the data input's value. ~~That value.~~
- Data output: This line emits the memory element's current value. That is, it is the value of the data input at the time that the trigger input was last asserted.

Thus, ~~we can~~ a ~~memo~~ memory element holds and emits a single bit value. We can set the value of the element at any time by placing the value on the data input line and then cycling — changing from Ø to 1 and again to Ø — the trigger input.

So, with this description of a memory element, how can we construct our device? The heart of this 1-bit counter is a feedback loop centered on the memory element. Specifically, the output of the memory element is the current state — the value of the counter, the child's turn, or the ~~jump ball pose~~ possession arrow's direction. Thus, we want the input to the memory element to ~~contain~~ be the ~~value~~ next state — the state of the device after the button is pressed. We can make a state transition table that lists every possible current state and matches it to the corresponding next state. ~~Here:~~ For this example, that table is:

current state	next state
Ø	1
1	Ø

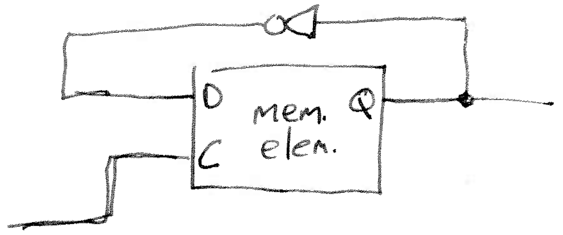
~~If we~~ Given that the data output of our memory element, which we will name Q, is the current state, and that the data input, heretofore D, is the next state, we can rewrite this table:

~~Q~~ Q	D
0	1
1	0

~~That is,~~ We can now express D as a function of Q:

$D = \overline{Q}$

We now use this ~~table~~ function to construct our counter circuit:



In this circuit, ~~the~~ we rename the trigger as the clock line, and label it C. (This renaming will seem more important later; for now, it is the line that controls the rate of progression of the counter.) By connecting our external button to C, we now have our toggle ~~FF~~ device, which shows its output on Q, to which we could attach some output device (e.g., a light) to observe the current state. ~~This circuit is~~

Now that we see ~~a~~ the simplest of sequential logic circuits, we must investigate how the heart of such circuits — the memory elements — are constructed.

II. Memory element structures

There is more than one way to construct a ~~device~~ memory element. Some involve ~~fundamental~~ components that are fundamentally different from the gates we have been using. ~~In fact~~, The ~~[illegible]~~ types of computer memory with which you may be familiar do not use gates at all: RAM; hard disks; CD's and DVD's; flash drives. ~~However~~ ~~Those~~ The construction of ~~a~~ the memory elements that compose those devices are beyond the scope of this book. ~~Their existence has less to do with fundamental & a~~ They exist because they make for economic forms of memory, especially at large scales (e.g., ~~gigabyt~~ gigabyte ~~&~~ memories and larger). ~~How~~

However, memory can be constructed ~~from~~ using the gates with which we are familiar. Doing so makes for the fastest types of memory available. Our goal is to use the components with which we are familiar — these logic gates — to construct memory elements. We will move through a progression of constructs, each ~~but~~ building upon and improving the previous design.

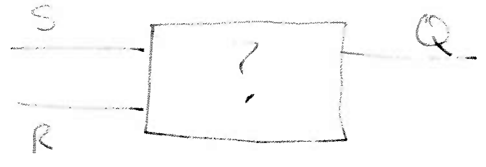
Ⓐ <u>S-R latches</u>

Let us begin with a memory element whose inputs are slightly different. Specifically, let the two inputs be:

① <u>Set</u> (<u>S</u>): Set the memory element to store (and <u>Q</u> to become) <u>1</u>.

② <u>Reset</u> (<u>R</u>): Set the memory element to store (and <u>Q</u> to become) <u>0</u>.

Schematically, then, we have:



~~To~~ ~~the~~ ~~How do we~~ More specifically, we expect an assertion of S to cause Q to become 1, and to remain 1 after S is no longer being asserted. Similarly, Q becomes 0 ~~after~~ when R is asserted, and remains 0 after R is no longer asserted. Thus, this memory element, known as an S-R latch, will ~~exit~~ have an output Q that is determined by the most recent assertion of S or R. \footnote{The behavior of an S-R latch is undefined if both S and R are asserted concurrently.}

We can try to express the logic of this memory element using a truth table~~, but we must~~:

S	R	Q
0	0	~~[illegible]~~ ?
0	1	0
1	0	1
1	1	~~0~~ —

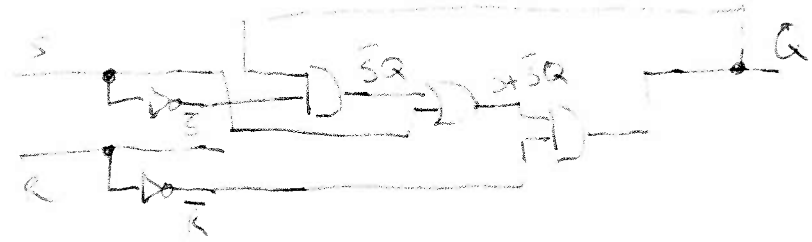
Notice that the normal truth table format does not quite work: ~~While~~ ~~it~~ Any combinational circuit may have undefined outputs for some inputs~~, as~~ as we have here for S=1 and R=1. What is unusual is that for one input combination — ~~S=Q and~~ $\underline{S=0}$ and $\underline{R=0}$ — those inputs are insufficient to determine the output ~~Q~~ $\underline{Q}$. ~~We~~ ~~Depending~~ ~~In this~~ For this case, where neither S nor $\underline{R}$ is being asserted, $\underline{Q}$ depends on which of S ~~and~~ or $\underline{R}$ was most recently asserted. Such cases indicate that this circuit is <u>not</u> combinational, since the output is not merely a logical ~~comb~~ composition of the inputs.

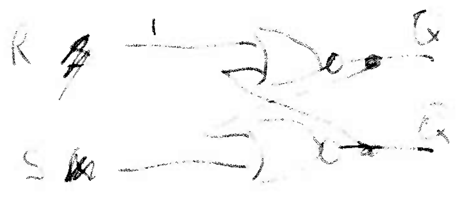
<u>NOTES</u>

$$Q = S\overline{R} + \overline{S}\,\overline{R}Q = \overline{R}(S + \overline{S}Q)$$

$S+Q$

$S + SQ + \overline{S}Q$





$Q = \overline{\overline{S} + \overline{Q}}$

$= \overline{S} + \overline{(\overline{R + Q})}$

~~$= \overline{S} + R + Q$~~

$= \overline{S}(R+Q)$

$= \overline{S}R + \overline{S}Q$

$\therefore Q = \overline{R + \overline{(\ldots)}}$

~~$= R$~~

$= \overline{R + \overline{(S+Q)}}$

$= \overline{R}(S+Q)$

~~$(S+\overline{S})(S+\overline{S}Q)$~~

~~$SS + S\overline{S}Q + \overline{S}S +$~~ $\overline{S}\overline{S}Q$ ~~$S +$~~ ~~$\overline{S}Q + \overline{S}\overline{Q}$~~

$S +$

$S + \overline{S}Q$

~~$(\overline{S}Q + \overline{S}\overline{Q}) + \overline{S}Q$~~

~~SQ $SS + \overline{S}S + \overline{S}Q$~~

~~$SS + (S+Q)\overline{S}$~~

~~$S + S + Q$~~

~~$S + Q$~~

$S(1+Q) + \overline{S}Q$

$S + SQ + \overline{S}Q$

$S + (S+\overline{S})Q$

$S + Q$

~~What is missing from this~~ Note that at any given moment, there is a value that indicates whether S or R was most recently asserted! Q. What if we use Q as an output and an input? Consider a new truth table:

S	R	input Q	output Q	
0	0	0	0	} determined by Q
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	—	} undefined
1	1	1	—	

Here, we still have an undefined output when S=1 and R=1; but we no longer have an output that cannot be ~~det~~ determined by the inputs. So what is the function that this truth table defines? In DNF:

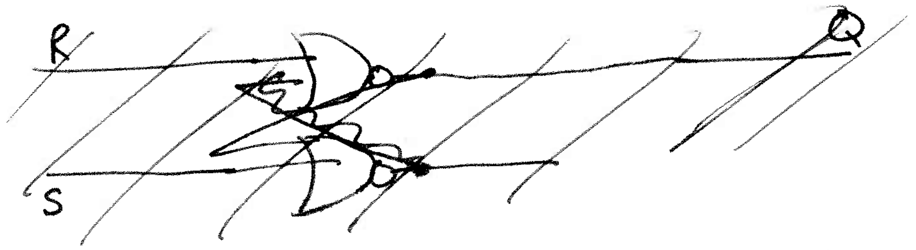
$$Q = \bar{S}\bar{R}Q + S\bar{R}\bar{Q} + S\bar{R}Q$$ [SFHK: Add the circuit.]

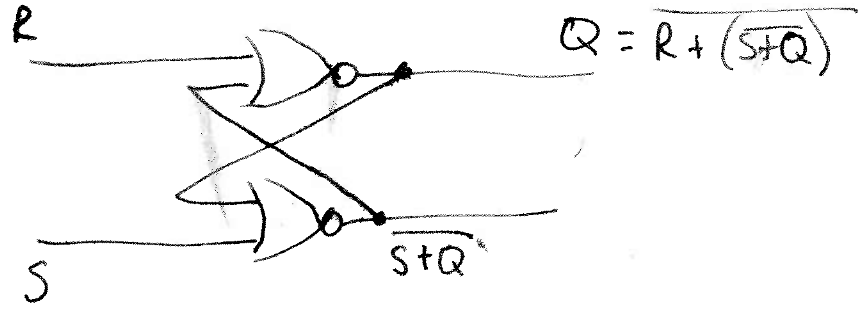
What may seem odd is that Q is defined in terms of itself. That is ~~thus,~~ Q is defined recursively. However, it is this very characteristic that allows the circuit to implement memory — to emit values based on its own value, ~~in,~~ (somewhat) independent of any external inputs.

With the application of some Boolean algebraic transformations, this ~~formula can~~ function can be substantially simplified:

$$Q = \bar{S}\bar{R}Q + S\bar{R}\bar{Q} + S\bar{R}Q$$
$$= \bar{S}\bar{R}Q + (\bar{Q}+Q)S\bar{R}$$
$$= \bar{S}\bar{R}Q + S\bar{R}$$
$$= \bar{R}(\bar{S}Q+S)$$
~~$= \bar{R}(\bar{S}Q+S(S+\bar{S}))$~~ $= \bar{R}(\bar{S}Q+S(1+Q))$
~~$= \bar{R}(\bar{S}Q+SS+S\bar{S})$~~ $= \bar{R}(\bar{S}Q+S+SQ)$
~~$= \bar{R}($~~ $= \bar{R}(S+\bar{S}Q+SQ)$
$$= \bar{R}(S+(\bar{S}+S)Q)$$
$$= \bar{R}(S+Q)$$
$$= \overline{R+(\overline{S+Q})}$$

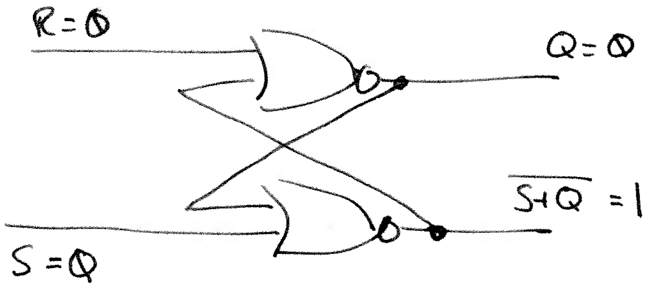
Let us draw the circuit that this ~~sim~~ transformed function suggests: Note that the only operator needed is NOR:



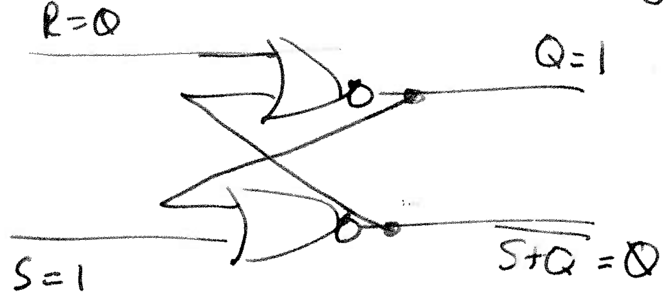


This elegant construction is the canonical one for an S-R latch, which is the most fundamental of memory elements.

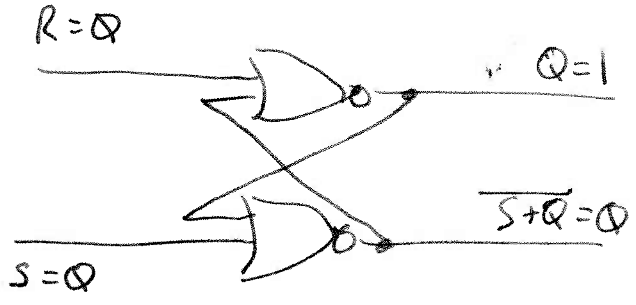
To see how it works, play out the values on this circuit. Make the initial assumption that <u>S = R = Q = 0</u> — that is, the memory element is storing the value 0, and neither the set nor the reset inputs is being asserted. With the assumptions, we can label the values on the circuit:



Notice that $\overline{S+Q} = 1$, making $\overline{R + (\overline{S+Q})} = 0$, and thus making these values <u>stable</u>. Now consider an attempt to set the memory element, by asserting <u>S = 1</u>:



Critically, when we return to <u>S = 0</u>, <u>Q</u> has been changed and will <u>remain</u> as <u>Q = 1</u>:

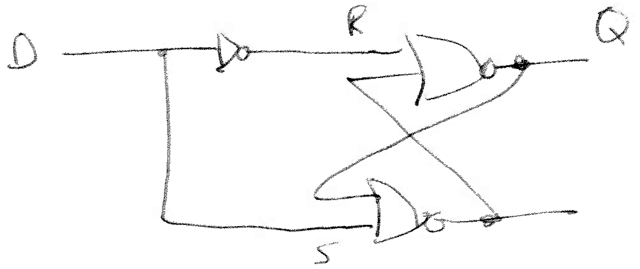


We leave it to the reader to explore the effect of asserting R=1 to reset the output to Q=0. Also, if you attempt to assert both S=R=1, you will see that Q will oscillate, as quickly as the gate delays allow, between 0 and 1, leaving the result unpredictable and thus undefined.

(B) D-latch

The S-R latch successfully "remembers" whether it was most recently set or reset. However, its inputs and outputs—that is, its interface—is not quite what we sought. Rather than assert one line to store the value 0, and assert a different line to store the value 1, we had hoped to have a memory element that adopted whichever value a single binary input, D, was carrying at a given moment.

To achieve this new interface, we could expand ~~the d~~ on the design of the S-R latch. Specifically, if D=0, we want to reset the latch, by having R=1; similarly, if D=1, we want to set the memory element by asserting S=1. So:



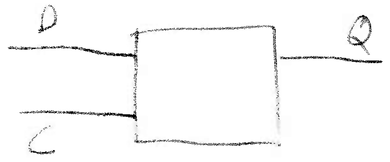
~~Here, if~~ D. This simple design ~~achieves what we sought~~ can be expressed as the following truth table:

D	S	R	Q
0	0	1	0
1	1	0	1

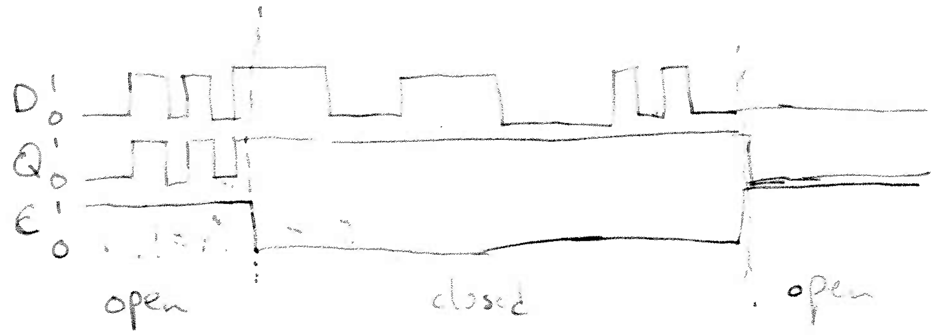
~~Notice that~~ This table reveals a problem with the design. Specifically, <u>Q = D</u>. That is, this design functions no differently than this "circuit":

The fundamental problem with our design is that there is ~~not~~ state in which neither <u>S</u> nor <u>R</u> is being asserted. ~~In our~~ Under this design, the memory element is <u>always</u> being actively set or reset. Thus, it never ~~rem~~ "remembers" ~~that~~ which value was last assigned to the memory element because a new value is continuously being assigned.

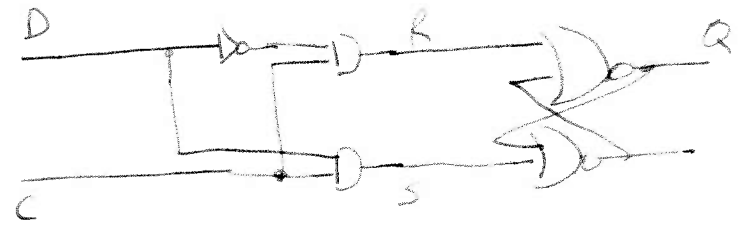
To fix this problem, we must recall the ~~in~~ memory element interface that we originally imagined in Section ~~2A~~ ~~2(A):~~ 2.I.(A):



This interface adds the ~~inpu~~ <u>clock</u> (<u>C</u>) input. This additional input specifies <u>when</u> the memory element should ~~a~~ adopt the value <u>D</u> ~~as~~. Specifically, when <u>C=1</u>, we consider the latch to be <u>open</u>, and <u>Q=D</u> (as above). However, when <u>C=0</u>, the latch is <u>closed</u>, and <u>Q</u> remains unchanged from its value at the last moment where <u>C=1</u>. We can see how <u>C</u> controls the relationship between <u>Q</u> and <u>D</u> using a <u>timing diagram</u>:



How can we alter our initial but flawed augmentation of the S-R latch to include a clock input? Consider that ~~to~~ either S ~~and~~ or R should be asserted if and only if ~~C=1~~ the latch is open (C=1), accepting a new value ~~for~~ from D. ~~Only~~ When the latch is closed (C=0), ~~both~~ ~~both~~ S=R=0, leaving Q unmodified. So:

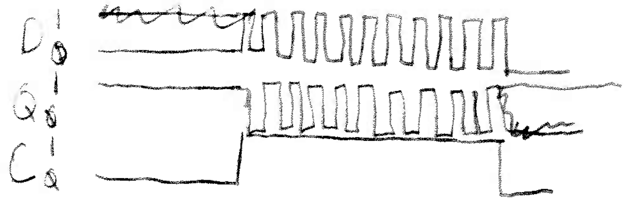


Here, the AND gates allow S or R to be asserted only when C=1, as desired. This design successfully gives us our D-latch.

Ⓒ D flip-flop

If we examine our example in Section 2.I.Ⓐ carefully, we may notice a problem ~~to~~ with our intended use of these memory elements and the structure of our sequential logic circuit. Specifically, notice that $D = \overline{Q}$: that is, the output from the memory element at the core of our 1-bit counter is ~~passed through a NOT gate, and then it~~ the input to a NOT gate, whose output in turn is the input to our memory element. ¶ The problem here is one of timing. Consider what happens when ~~t~~ the latch is opened (C=1). Assume ~~that~~, at the first moment at which the latch is open, that Q=1. The time required, using silicon gates, for this value to flow into the NOT gate ~~g~~ and emerge as the input D=0 to the memory element is miniscule. If the clock for the memory element does not return to C=0 very rapidly, then ~~the news~~ that input value will be adopted by the memory element, ~~m~~, making Q=0. ~~If C rema~~ the latch remains open, ~~this~~ the traversal of Q=0 into ~~the NOT~~ and through the NOT gate, setting D=1, and again changing Q=1 again, will all occur rapidly.

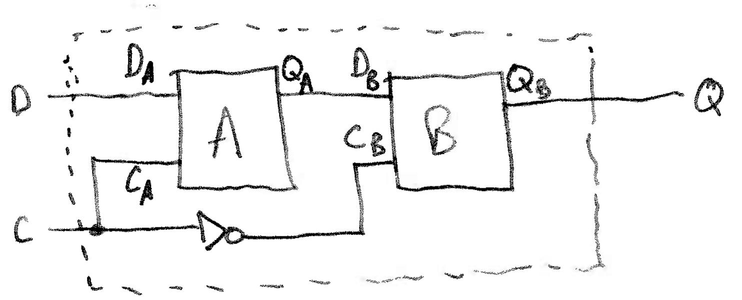
In short, while the ~~g~~ latch is open, D and Q will ~~oscillate~~ ~~to~~ alternate between 0 and 1 so rapidly that is will be difficult — ~~or~~ in practice, impossible — to predict the value of Q at the moment that ~~the to~~ C=0, closing the latch. Depicted as a timing diagram, the behavior of our 1-bit counter using a D-latch as ~~the~~ the memory element would be:



Therefore, for ~~our the~~ structures like our 1-bit counter cannot rely on D-latches as the memory elements. \footnote{We will see, ~~in~~ that this 1-bit counter ~~is~~ is not only a common structure, but the base structure for general purpose processing devices.}~~E~~ A D-latch is open for the duration defined by C=1 ~~te~~; it is therefore known as being level triggered, where the level is the portion of the timing diagram where C=1 and the latch is open.

We require a memory element whose latch opening is of a much shorter duration. Specifically, it must open and then close again before the new output Q, can change, by flowing through the intermediate circuit, D.

To achieve that goal, the problem is less a logical one, and more one of structure and timing. The following structure, known as a D-flip-flop, ~~is~~ comprises two D-latches:

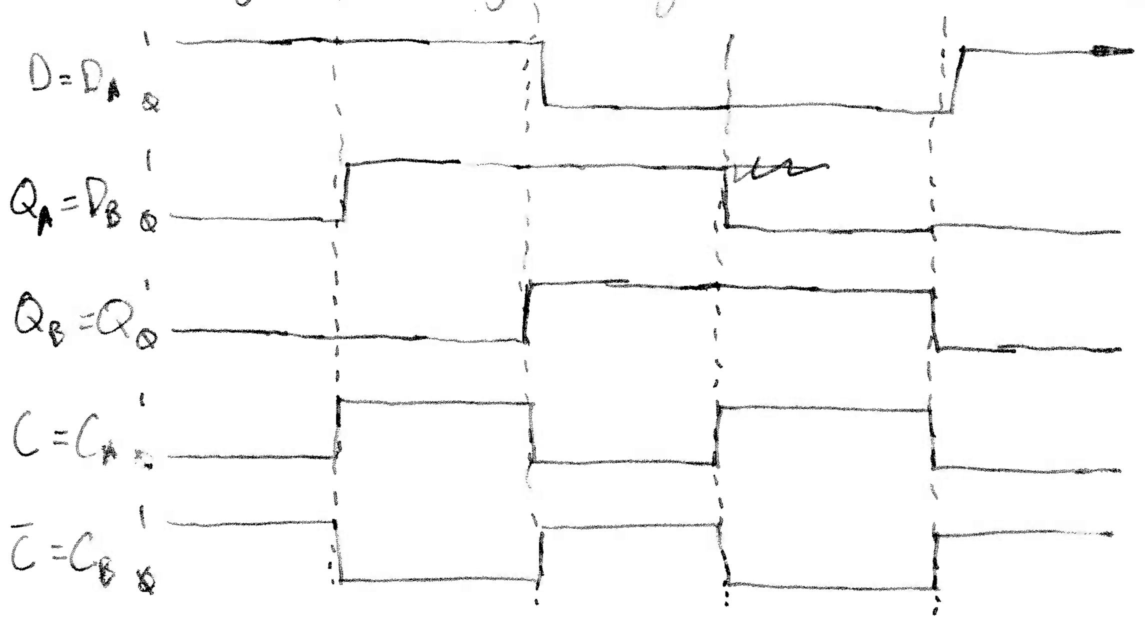


We see that these two latches are connected in sequence. Moreover, their clock input ~~is that of~~ one is the inverse of the other. Specifically, if D is the original input, C is the externally provided clock line, Q is the final output, and $D_i / C_i / Q_i$ are the inputs and outputs of ~~a~~ internal latch i:

$$D_A = D \qquad Q = Q_B \qquad Q_A = D_B$$

$$C_A = C \qquad C_B = \overline{C}$$

Determining that this structure provides the behavior that we seek requires careful consideration. Let us again consider our 1-bit counter, this time using ~~t~~ the flip-flop as the memory element. Assume that, initially, the memory element stores a 0 (that is, ~~$Q=0$)~~ $Q = Q_B = 0$. Consequently, $D = D_A = 1$. Finally, assume that we begin considering the sequence of events while $C=0$.



While $C=0$, $C_B = \bar{C} = 1$, implying that latch B is open. ~~However,~~ $C=C_A=0$, implying that latch A is closed. Therefore, since latch A is closed, then its output remains ~~unchaned~~ unchanged, so $Q_A = D_B = 0$. This value "passes through" the open latch B, appearing as $Q=0$.

When C transitions from 0 to 1, latch A opens $(C_A=1)$ just as latch B closes $(C_B=0)$. ~~Ti~~ ~~The timing of this step is essential.~~ The opening of latch A allows the new ~~vale~~ input value presented on $D=1$ to "pass through" that latch and reach $Q_A=D_B=1$. However, latch B closes before this new value reaches its input D_B. Thus,

$Q = Q_B$ remains 0.

While $C=1$, Q remains unchanged. For a different larger circuit (that is, not the 1-bit counter), any changes to D while $C=1$ will pass through ~~such~~ the open latch A such that $D = D_A = Q_A = D_B$. However, the closed latch B will continue to prevent any changes to Q.

The critical moment for this flip-flop occurs when C transitions from 1 to 0. Specifically, latch A closes. Consequently, ~~what~~ whichever value is assigned to D at the moment that latch A closes is the value that latch A will emit from that moment onward, until latch A opens again later. Since latch B will open, ~~$Q_B = D$~~ $Q_A = D_B = Q_B = Q$, thus carrying the ~~value~~ output of latch A as the output of the entire flip-flop. That is, ~~whatever~~ the value of D at the moment that C transitions from 1 to 0 is the ~~value~~ new value that the flip-flop adopts until the clock cycles ~~again~~ and C transitions from 1 to 0.

This moment of the transition from $C=1$ to $C=0$ is known as the falling clock edge. Likewise, the transition from $C=0$ to $C=1$ is called the rising clock edge. We say that D-latches are level triggered — that it adopts new values during the time that C has a level value (in our construction, while $C=1$. In contrast, a D-flip-flop is edge triggered, adopting a new value exactly at the moment that C transitions ~~from 1 to 0~~ one value to another. In our ~~con~~ construction this moment is on the falling edge, as C transitions from 1 to 0, which we will write as C↓. ~~T~~ By examining our timing diagrams, we see that the duration of the clock edges is much smaller than the duration of the levels. Thus, flip-flops are open to new values for a very short duration, addressing our problem with the 1-bit counter ~~where~~ by having a memory element open and close again before the ~~new v~~ memory element's new output values can affect its input value.

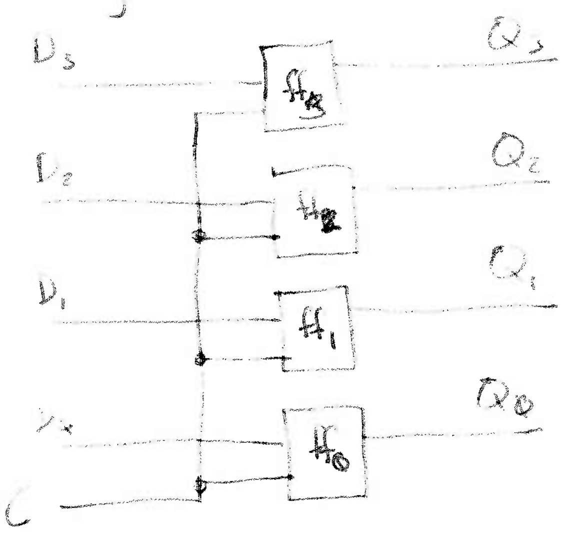
Because the 1-bit counter is ~~a~~ the most simple example of ~~a~~ an essential structure known as a sequential logic circuit, and because ~~these so much of~~ CPUs ~~design is~~ are large, complex sequential logic circuits, we will heretofore assume that any memory element is a D flip-flop.

(2) ~~Registers~~ ~~Addressible memory~~ & Registers

Now that we are able to store a 1-bit value, we must consider how to group those 1-bit memory elements together so that we can store K-bit values. ~~Moreover, for many interesting computations, we may wish to store more than one K-bit value. Thus, we need~~ Luckily, doing so requires no logic, and no ~~explicit~~ connections between memory elements.

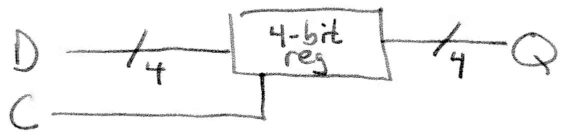
To store ~~K bits of data~~ a K-bit value, we need only to store each of the K bits in its own 1-bit memory element. ~~T~~ Such a collection of K 1-bit memory elements to store a K-bit value is known as a register. Although K can take on any value, it is typically a power of 2. Moreover, ~~because 8-bit values~~ a ~~known~~ ~~the most~~ ~~#~~, common size ~~include~~ is 8 — the number of bits in a byte, used to store a single character of text. ~~(as well as 32~~

To make a register behave as a single device, ~~there is one~~ it must be able to adopt all K bits of a K-bit value at the same moment. Thus, all of the 1-bit memory elements that compose a register must share a clock line. That is, a 4-bit register would have the following internal structure:



~~Any~~ A 4-bit input value $D = (D_3, D_2, D_1, D_0)$ is adopted into the four flip-flops (ff_3, ff_2, ff_1, ff_0) when the clock C cycles (transitions from 0 to 1 and back to 0 again). Once D is clocked into the register, ~~the~~ its value at the moment that the clock's falling edge ~~will~~ becomes the output value $Q = (Q_3, Q_2, Q_1, Q_0)$.

Now that we are using multibit values, we will ~~[illegible]~~ mark any line in a circuit diagram with the number of bits it represents. For example, we can redraw the above 4-bit register example:



Here, we ~~sepa~~ condense the four wires carrying the values of D and Q to single lines marked with a 4 to indicate the width of the value. We also encapsulate the four 1-bit flip-flops in a single box labeled 4-bit reg, since we now know its inner structure.

(E) Addressible memory

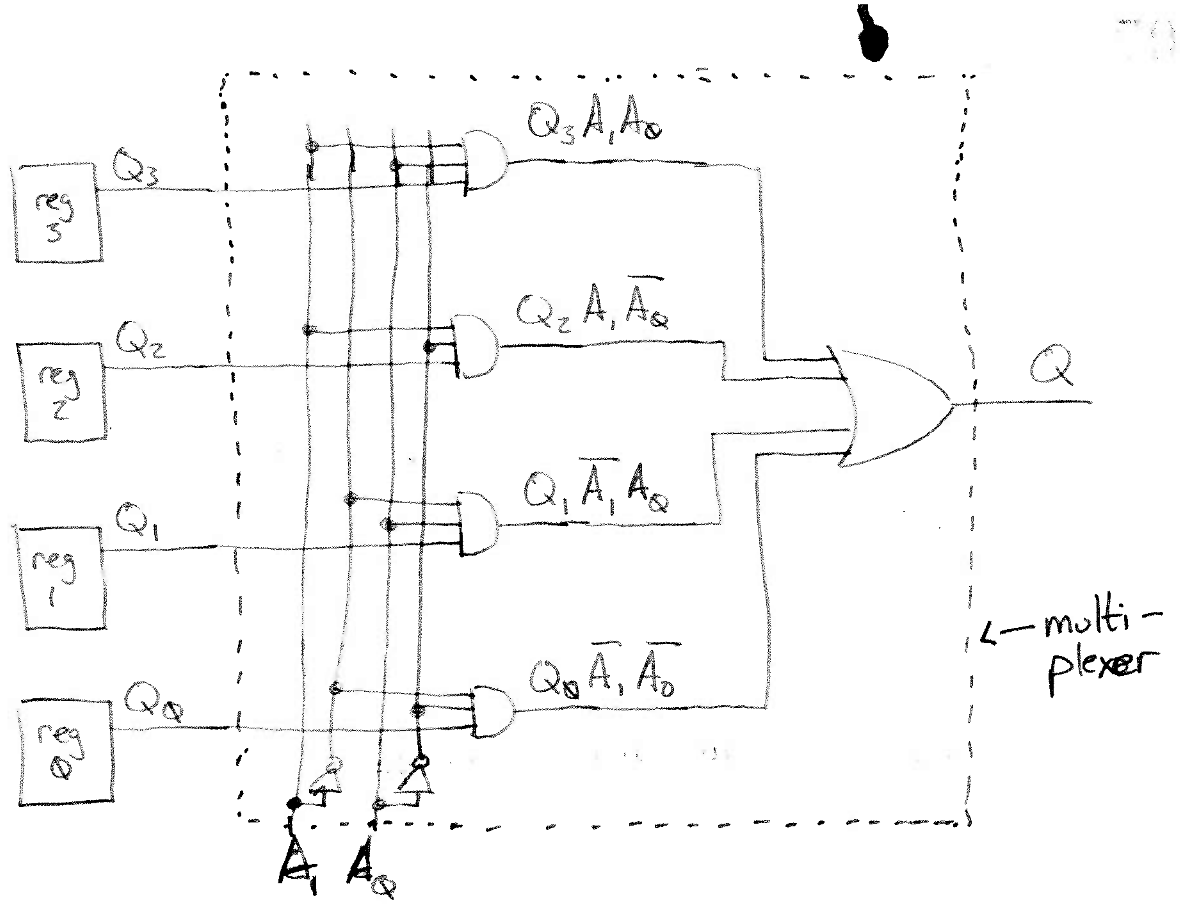
Interesting computations require the storage, in memory elements, of not just one K-bit value, but many of them. We would like a structure that allows a circuit to access any one of m K-bit registers. That is, ~~we want~~ with a given K-bit value, we want the ability to store it in ~~any one~~ the register, chosen from m of them, of our choice. Likewise, given those m registers, each storing and emitting a K-bit values, we want ~~th~~ a circuit that allows us to select one of those registers and its value.

We are describing addressible memory: a collection of registers where each is assigned a unique number — a memory address — by which it can be selected for reading (using its currently stored value) or for writing (assign it a new value).

To explore the structures necessary to create this type of addressible memory, we will assume, for illustrative purposes, that we want to store ~~4~~ 2-bit values. Moreover, we will assume that we have four 2-bit registers whose access we want controlled by our addressing scheme. Thus, we assign unique identifiers to each register, $\underline{0}$ through $\underline{3}$. Furthermore, we assume that there is a single 2-bit input value $\underline{D}$, and a single 2-bit output value $\underline{Q}$ for the entire addressible memory structure. Finally, there will be a 2-bit address input $\underline{A}$ that will select one of the four registers from which to read or write.

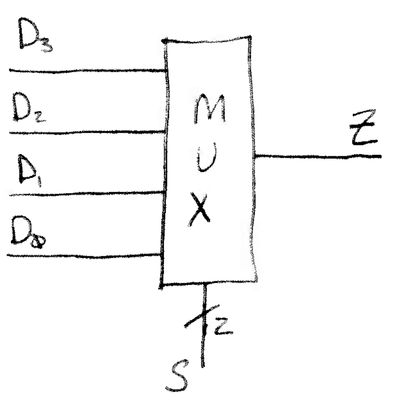
READING FROM ADDRESSIBLE MEMORY: First, we ~~consider~~ seek a circuit that will ~~select~~ allow us to select the output from one of the four registers. To do so, we will initially simplify our problem by assuming that we are using not 2-bit registers, but rather 1-bit registers. Once we have established a structure for selecting the output of one out of four 1-bit registers, then we will expand the structure to handle our original 2-bit registers.

The central tool in building this "selective reader" circuit is the AND gate. More specifically, we combine the bits of $\underline{A}$ (or their negation) along with the output Q_i of ~~regi the re~~ register $\underline{i}$ in a way that the output of the AND gate is $\underline{1}$ if and only if $\underline{Q_i = 1}$ and $\underline{A = i}$. We can then inclusive-OR the outputs of these AND gates — one per register ~~A~~. ~~Since~~ Since $\underline{A}$ selects only one register, then exactly one of the AND gates can emit a $\underline{1}$, thus affecting the final, disjoined output. The structure looks like: ~~this~~:



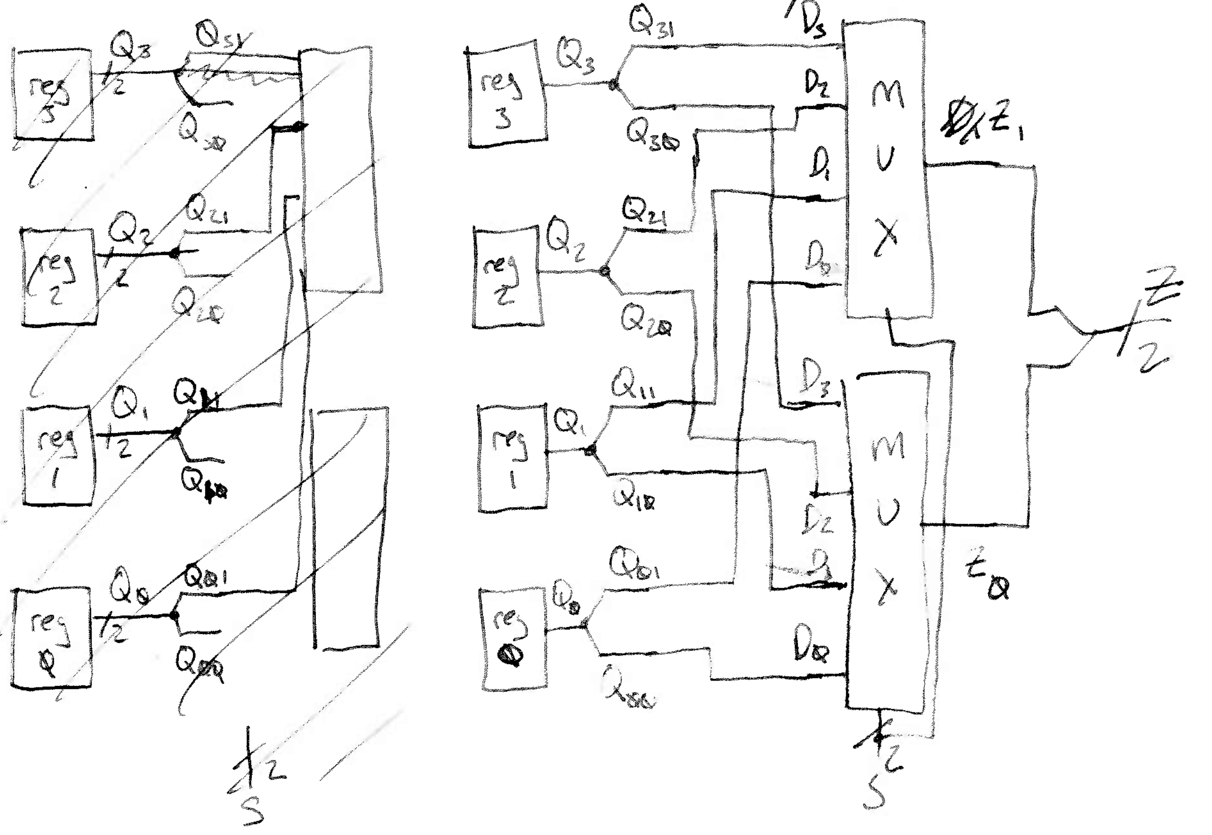
This structure, known as a <u>multiplexer</u>, selects one of $\underline{Q_3}$, $\underline{Q_2}$, $\underline{Q_1}$, or $\underline{Q_0}$ to become the output $\underline{Q}$. The address $\underline{A = (A_1, A_0)}$ selects which ~~of the Q_i~~ such that $\underline{Q_A = Q}$. Thus, by presenting as $\underline{A}$ the number of the desired register, the value stored in that register is emitted as the output of the multiplexor, $\underline{Q}$.

How do we expand this structure to select <u>2-bit</u> values? Assume that each register stores two bits, and thus that each $\underline{Q_i}$ is a 2-bit value $\underline{(Q_{i1}, Q_{i0})}$. For the multiplexer to work, we must replicate the <u>multiplexer</u> structure on a per-bit basis. That is, we need a multiplexer for the more significant $\underline{Q_{i1}}$ bits, and then we need a second multiplexer for the less significant $\underline{Q_{i0}}$ bits. So, ~~if we~~ assume a high-level representation of the multiplexer, ~~[illegible]~~ like so:



Here, the four $\underline{D_i}$ inputs are 1-bit values; $\underline{S}$ is a 2-bit value, identical to the 2-bit ~~A~~ input $\underline{A}$ shown above; and $\underline{Z}$ is the 1-bit output. The internal structure is identical to what is shown above as the multiplexer component of readable memory.

Given this high-level representation of a multiplexer, we can structure a 2-bit readable, addressible memory like so:



That is, the multiplexer structure is replicated for each of the $\underline{k}$ bits when selecting $\underline{k}$-bit values, where each ~~m~~ 1-bit multiplexer takes inputs from ~~k~~ each of the $\underline{m}$ k-bit input values, specifically the group of $\underline{m}$ bits of the same significance within those $\underline{m}$ values. The ~~reso~~ outputs from each 1-bit mux can be joined to construct the single $\underline{K}$-bit selected value.

<u>WRITING TO ADDRESSIBLE MEMORY</u>: To write a new 2-bit value into one of the four availabile registers, we begin with a few critical observations: