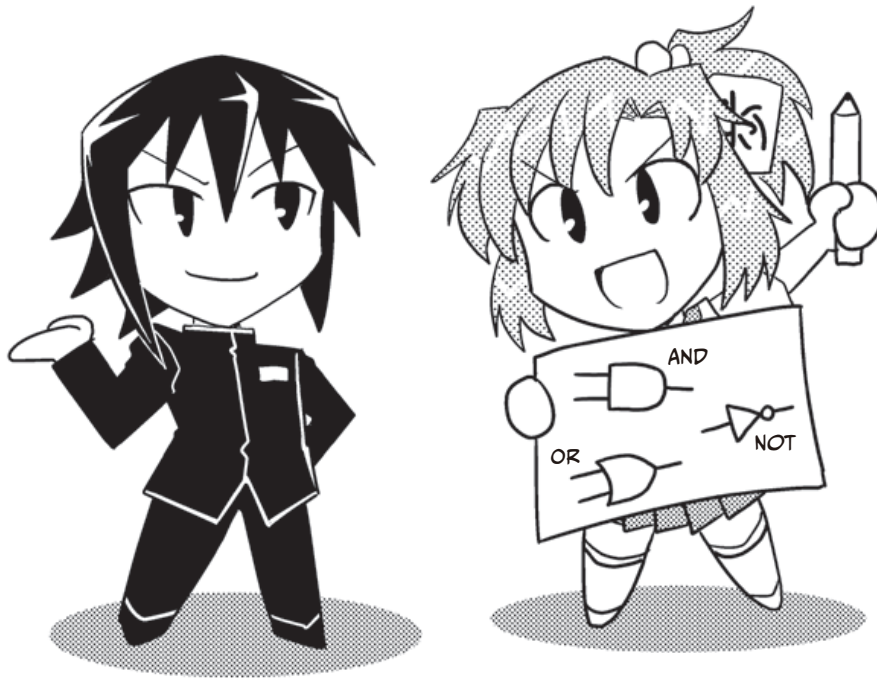
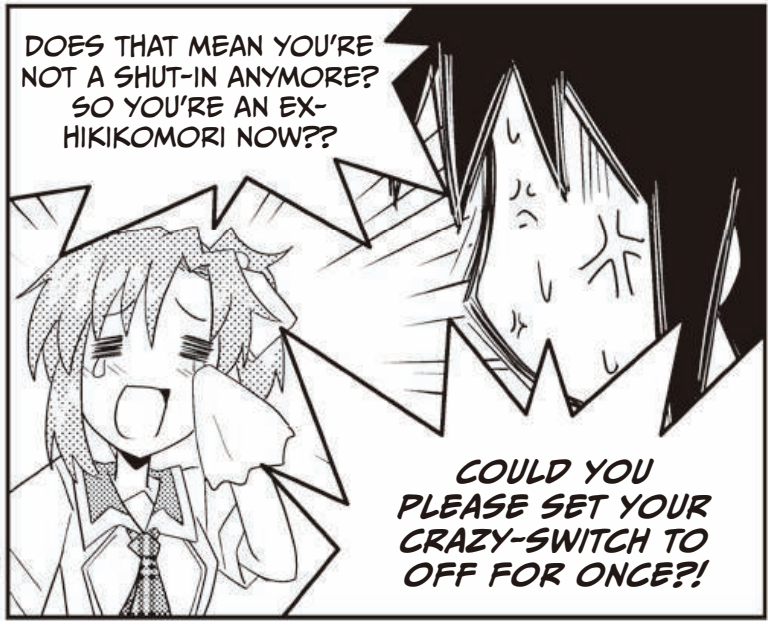
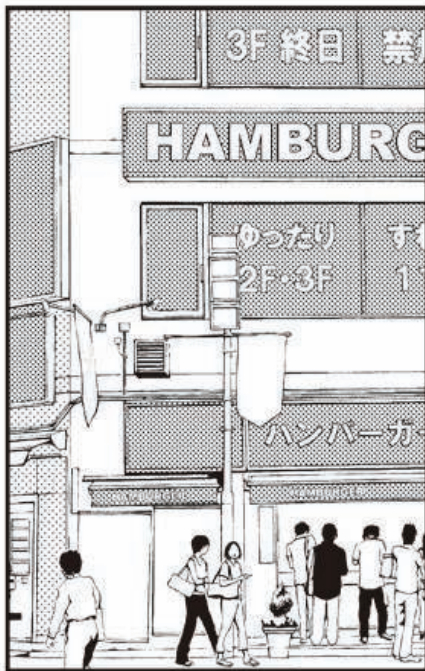




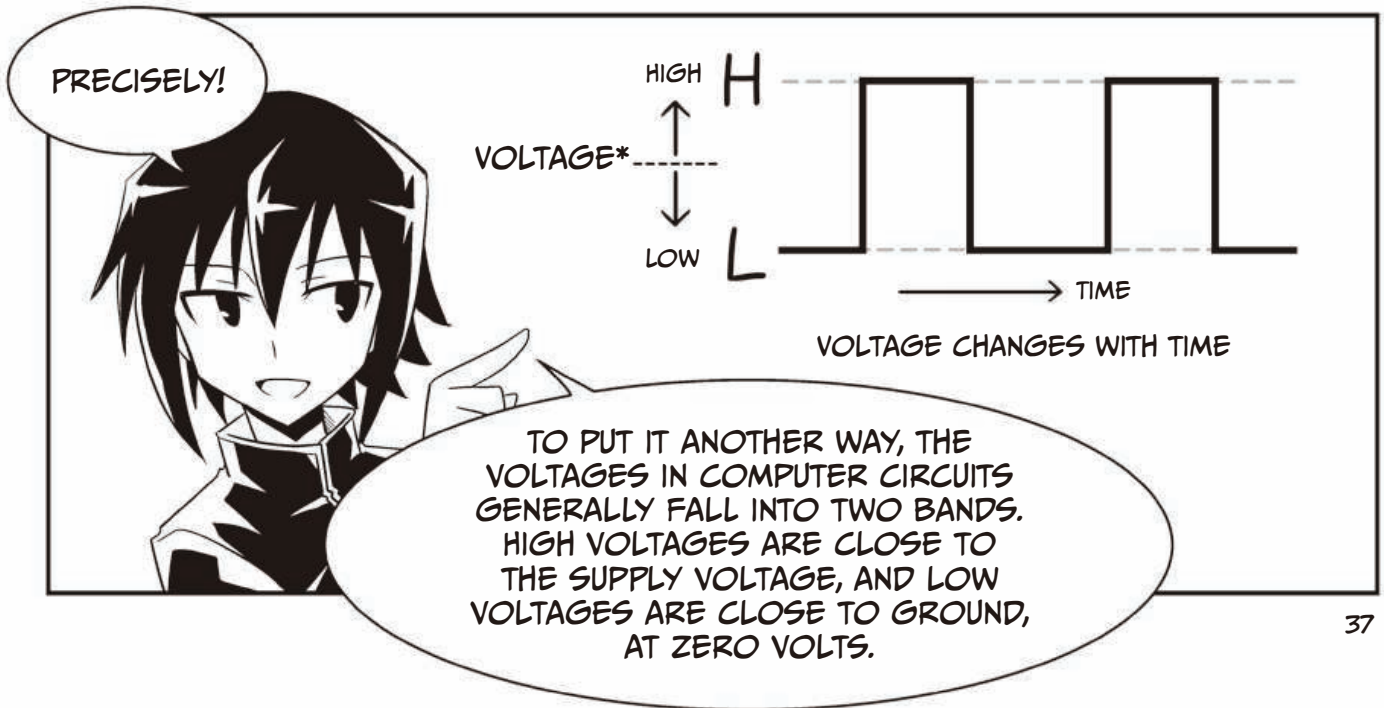
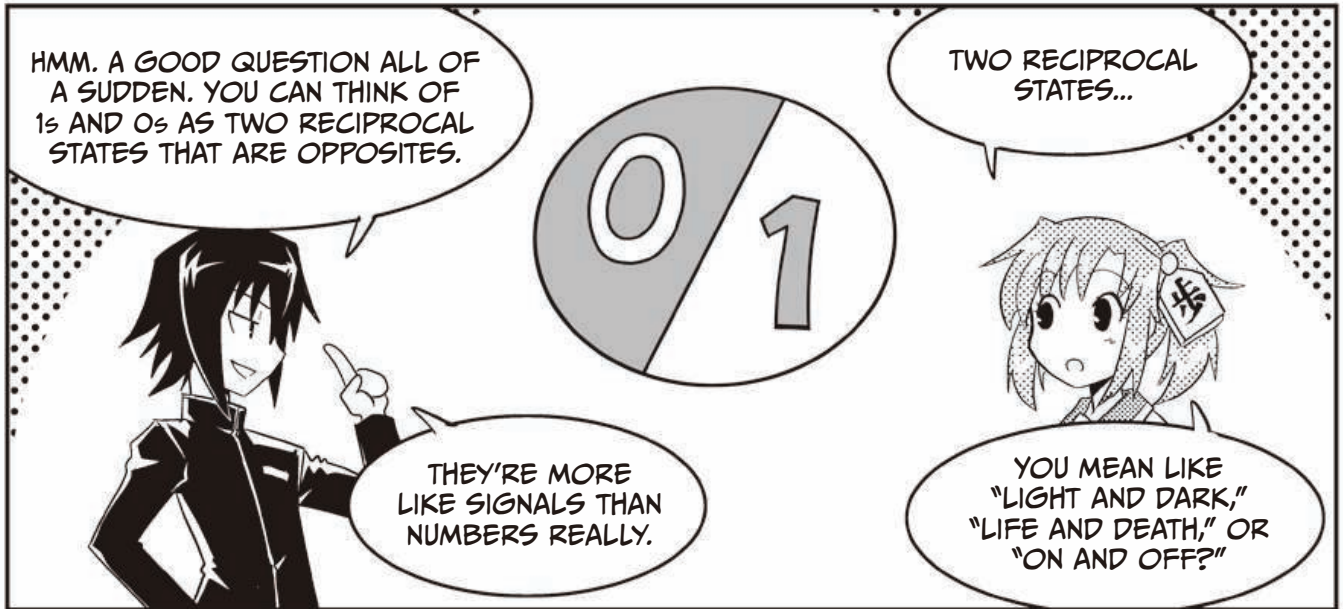
DIGITAL OPERATIONS

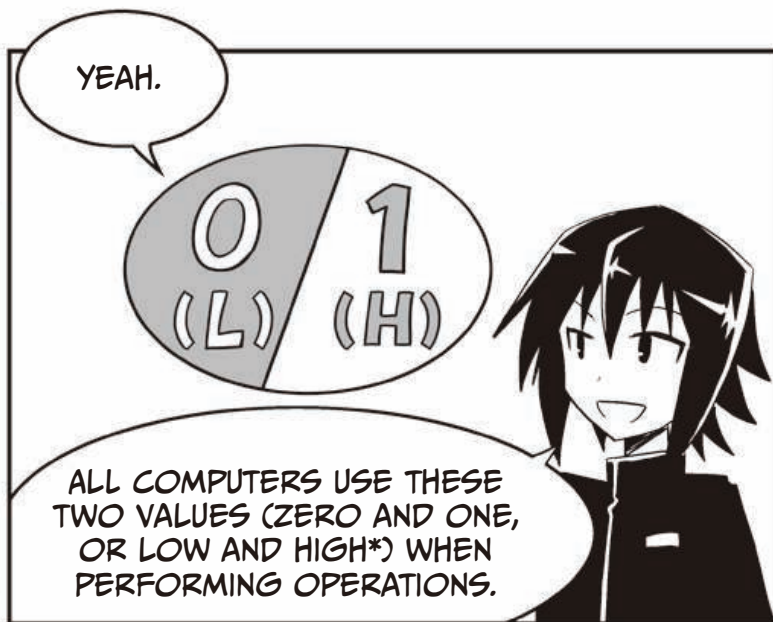
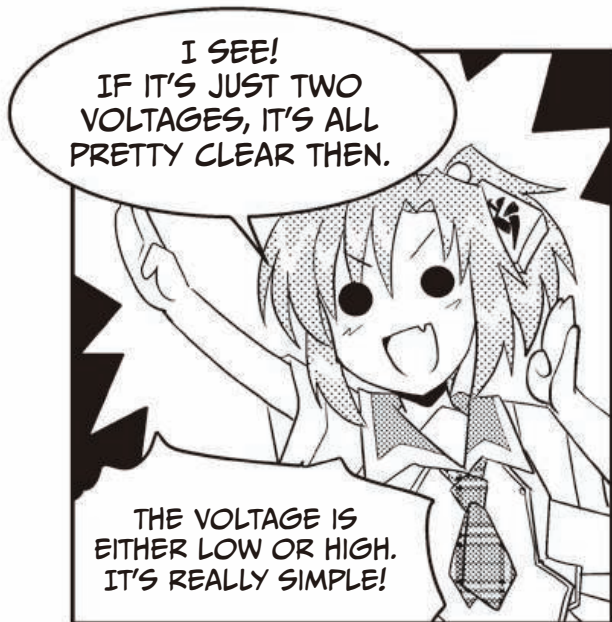


The Computer's World Is Binary



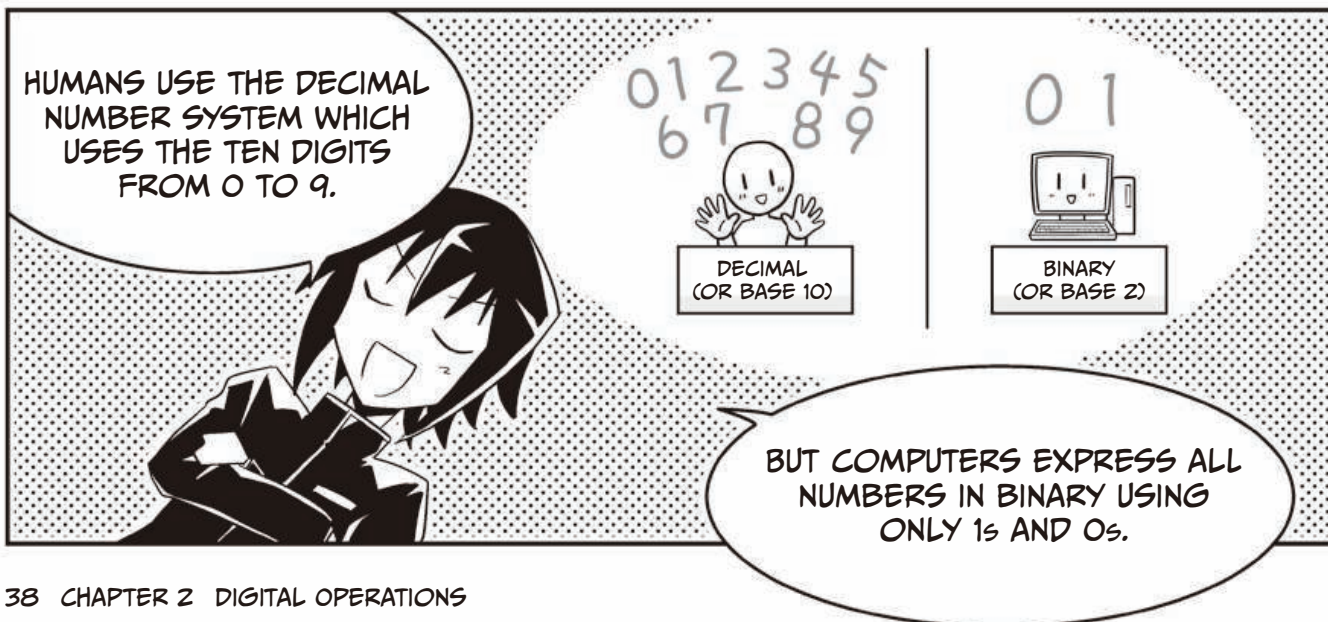
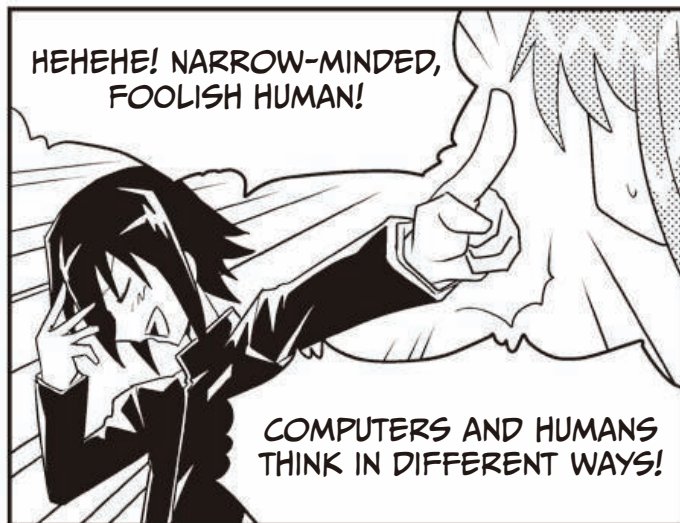
THE RECIPROCAL STATES OF 1 AND 0





* IN THIS BOOK, WE'LL TREAT LOW AS ZERO AND HIGH AS ONE,
BUT IT'S POSSIBLE TO DO IT THE OTHER WAY AROUND AS WELL.
IT'S UP TO THE SYSTEM DESIGNER WHICH ASSIGNMENT TO USE.

DECIMAL AND BINARY



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
⋮	⋮

AS YOU CAN SEE, YOU DON'T NEED MORE THAN 1s AND 0s!!

ANOTHER DIGIT!

ANOTHER DIGIT!

ANOTHER DIGIT!

ANOTHER DIGIT!

WOW, IT REALLY IS ONLY 1s AND 0s! BUT THE NUMBER OF DIGITS INCREASES REALLY FAST IN BINARY...

BY THE WAY, A BINARY DIGIT (A ONE OR A ZERO) IS ALSO CALLED A BIT IN COMPUTER TERMINOLOGY. THAT'S REALLY IMPORTANT, SO DON'T FORGET IT!

COMPARING DECIMAL AND BINARY

1bit

1001

FOUR DIGITS, SO FOUR BITS

A FOUR-DIGIT BINARY NUMBER IS FOUR BITS... SO, TO EXPRESS THE DECIMAL NUMBER 9, WE WOULD NEED FOUR BITS (1001), RIGHT?

COME NOW, ARE YOU PREPARED TO DIVE INTO THE WORLD OF 1s AND 0s?!

SWISH

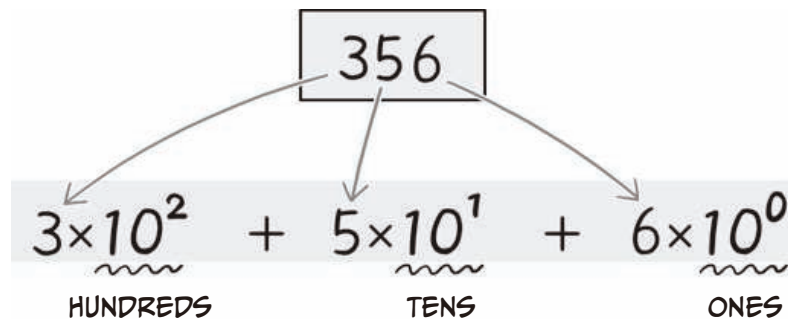
AH, SURE!

I WONDER IF HE'S ALWAYS THIS HYPER...

EXPRESSING NUMBERS IN BINARY



Well then, let's learn the basics of binary, or *base 2*, math! Let's start by thinking about the decimal, or *base 10*, system that we use every day.



Any number to the power of zero is equal to one. For example, $10^0 = 1$, and $2^0 = 1$.

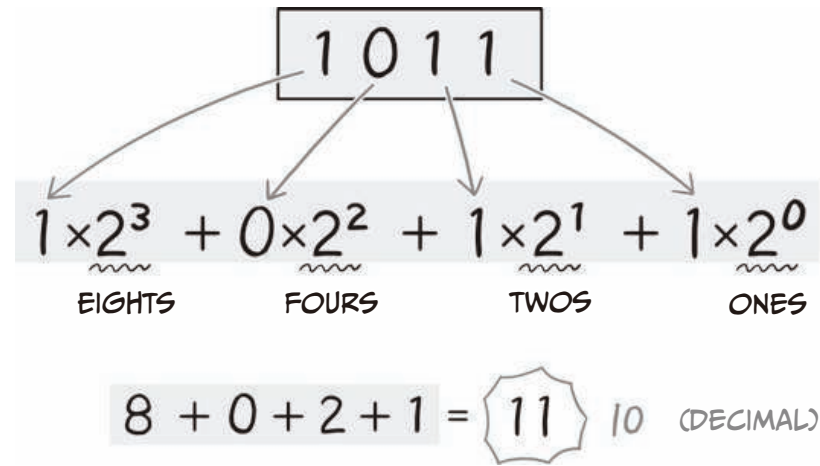
For example, the number 356 is divided up as in the illustration above. Each digit is multiplied by successive powers of ten to get the final value.



Okay! It's really easy if I just think of it like different coin denominations: 356 yen is just three 100-yen coins (10^2), five 10-yen coins (10^1), and six 1-yen coins (10^0) added together.



That's right. The next step is to apply that same logic to binary. We just swap the 10 in our decimal calculations for a 2 in the binary case to get the appropriate factors for each digit. Take a look at this picture.

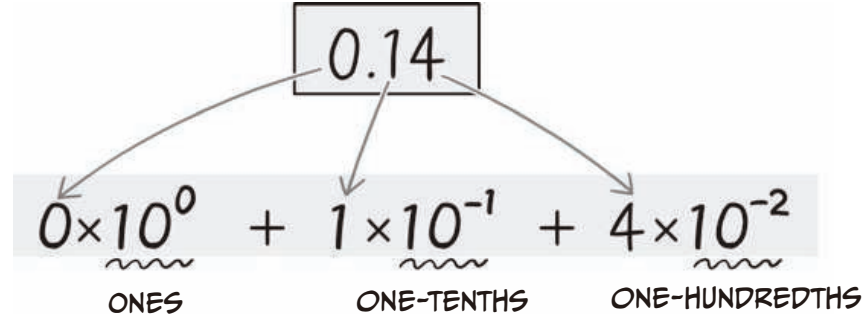


Uh-huh! I don't think anyone uses coins like this though... but if someone did, I would just take either 1 or 0 of each of the 8 yen, 4 yen, 2 yen, and 1 yen coins, right?

This means that the binary 1011 would translate into $8 + 0 + 2 + 1 = 11$. As soon as you understand the basic principle, it's easy!



By the way, this also works for fractional expressions. Take a look at this.

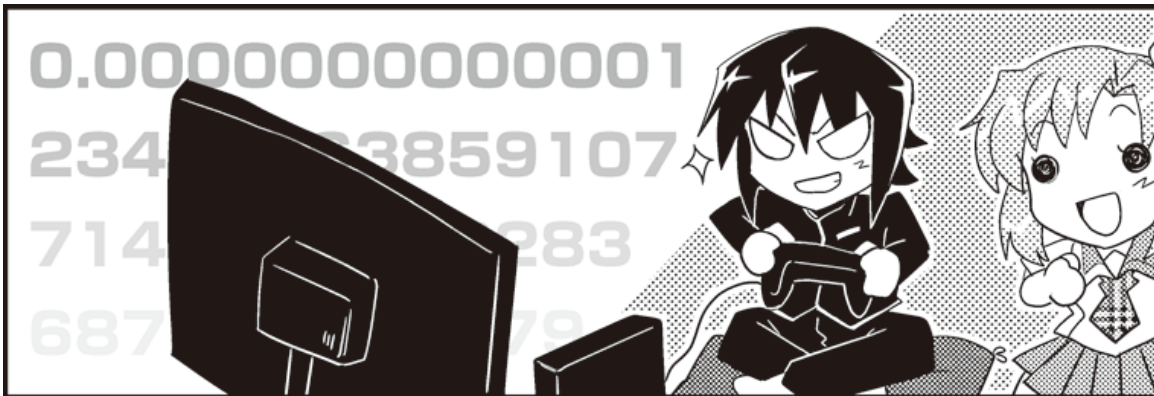


In decimal, each digit after the decimal point has factors using negative powers (10^{-1} , 10^{-2} etc.). So we have one-tenth (0.1), one-hundredth (0.01), and so on.



Sooo, it's the same reasoning with binary, right? That means that it would be (2^{-1} , 2^{-2} , 2^{-3}) and so on after the decimal point as we add more digits. So the factors would be one-half (0.5), one-fourth (0.25), one-eighth (0.125), and so on. It seems a bit cumbersome, but I think I get it.

FIXED-POINT AND FLOATING-POINT



Next up, I'll teach you a really important concept. In computers, there are two ways to store fractions—either fixed-point or floating-point.

When using extremely small values like 0.0000000000000000...001 or very large values like 10000000000000000..., it is a lot more practical to use floating-point fractions.



Hmm... Why is that? What's the difference?



Well, for example, instead of writing a billion in decimal, as 1,000,000,000, you could write it as 10^9 to save some space, right? We call this form *scientific notation* or *standard form*, where the n in 10^n is called the exponent. *Floating-point* fractions use scientific notation when storing values.

In contrast, *fixed-point* fractions express values the way we're used to, with a decimal point. When expressing integers with this method, you can imagine the decimal point being at the far right of the number. Here's a comparison of the two.

FIXED-POINT	FLOATING-POINT
123. ← DECIMAL POINT	1.23×10^2
1230000.	1.23×10^6
0.00000123	1.23×10^{-6}



Oh, okay. So if you're using fixed-point fractions to express really large or really small numbers, the number of digits you need increases by a lot. But if you're using floating-point, only the exponent gets bigger and smaller, while the number of digits stays the same. Yeah, that's really useful!



That's right. That last example was in decimal, but since computers use binary, the principle becomes even more relevant. The most common variant used is this one.

$$\begin{array}{ccc} \text{AN EXAMPLE} & & \\ \text{SIGNIFICAND} & & \\ 1.69 & \times & 2^n \text{ EXPONENT} \\ \underbrace{\hspace{2cm}} & & \underbrace{\hspace{2cm}} \\ \text{SIGNIFICAND} & & \text{BASE} \end{array}$$

An example of floating-point representation inside a computer
(using a base 10 number for illustration)



I made our example here, 1.69, decimal just to make it easier to understand. The number would be in binary in a computer. The important part here is that this significand always has to be greater than 1 and less than 2.



Hmm... So this representation makes it easy for computers to handle extremely small and extremely large numbers, right? They're also easy to use in calculations, huh.



Yes! And it's also important to understand that the speed with which you can calculate using floating-point numbers is a very important question and ties in deeply with CPU performance. (See page 139 for a more detailed explanation.)

Generally, scientific calculations require an accuracy of only around 15 digits, but in some cases, 30 are used. Some modern encoding algorithms even use integers of up to 300 digits! It's worth mentioning that gaming systems that process real-time, high-fidelity graphics use floating-point arithmetic extensively.



Ugh... I don't think I could do those calculations in my head. I hate to lose to computers, but I hope they're at least advancing some fields of science!

ADDITION AND SUBTRACTION IN BINARY



It's finally time to talk about binary arithmetic. Let's start by thinking about addition. First off, adding two bits works like this!

$$0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10$$



Okay, that's easy! The last equation, $1 + 1 = 10$, means that we carried the 1 to the second place value and the first digit became 0, right?

$$\begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$

CARRIED TO THE NEXT PLACE VALUE PLACE



Yeah. If you understand how to add one bit to another, you should be able to understand calculations with more digits as well. For example, when adding the binary numbers $(1011)_2 + (1101)_2$, you just need to start from the right and work your way to the left, carrying digits as you go.* Take a look here.

$$\begin{array}{r} 1011 \\ + 1101 \\ \hline 11000 \end{array}$$

CARRY



Uh-huh, I just have to be careful with the carries, right? Binary addition is pretty simple! Or, it might just be my genius shining through.



Hey! Okay then, let's take a look at subtraction next. When doing subtraction, it is important to learn how to create negative values using a technique called *two's complement*.

Adding the two's complement (a number that corresponds to the negative version of a number) of a binary number A to another number B is the same as subtracting A from B!! What do you think—pretty cool, right?

* $()_2$ means the number is in binary representation and $()_{10}$ means decimal representation.



Ahh... I'm sorry to stop you when you're on a roll, but I didn't understand that at all... What are you talking about?



Let's start out slow in decimal. First off, let's agree that subtracting 15 is the same as adding -15. But what would you do if you weren't allowed to use the minus sign at all? Is there some other number that we can use to represent the number -15?



I... I have no idea. Stop putting on airs and just teach me already!



Where did your genius go? Well, have a look at these two equations then.

EQUATION A		EQUATION B
$\begin{array}{r} 15 \\ + (-15) \\ \hline 0 \end{array}$		$\begin{array}{r} 15 \\ + (85) \\ \hline 100 \end{array}$
		↑ IGNORE!

Looking at just the final two digits of these equations, we see that the result of equation A is 0 and the result of equation B is 00. We could therefore say that for the last two digits, the results of $15 + (-15)$ and $15 + 85$ are the same!



Whaaa...? You're right, 0 and 00 are the same! But what happens to the 1 in 100 of the equation B result?



Hah! Since we're doing two-digit math at the moment, we don't care about digits that carry over beyond those two. Just pretend you can't see them! We call those *overflow*, and we just ignore them.



What kind of twisted reasoning is that? Is that even allowed?



Heh heh heh! Surprised? In situations like this, we say that 85 is the ten's complement of 15. In other words, we say that a number's *complement* in some base is the smallest number you have to add to the original number to make the number's digits overflow. As the name suggests, you can think of the numbers "complementing" each other to reach the next digit.

And this complement corresponds to the original value's negative form. So in this case, 85 is essentially equal to -15.

Let's take another example. When calculating $9647 - 1200 = 8447$, we might as well calculate $9647 + 8800 = 18447$ and ignore the carry. That's because in the result, we see that the lower four digits are the same. Therefore, we can use 8800 as the ten's complement of 1200 during addition to get the same result as we would get using subtraction.



Uhh... This is getting pretty hard to grasp! So using complements, we can perform subtraction using addition instead. I suppose that might be useful? So what happens if we try this farfetched solution with binary numbers?



It's not farfetched—it's awesome! It's logical!! Anyway, let me show you how to do it in binary.

$$\begin{array}{r}
 1010\ 1000 \\
 +\ 0101\ 1000 \\
 \hline
 \text{IGNORE! } 1\ 0000\ 0000
 \end{array}$$

ADD THE TWO NUMBERS: IF THE RESULT IS 0 (IGNORING THE OVERFLOW), IT MEANS THE NUMBERS ARE COMPLEMENTARY.

As you can see, when you add two binary numbers and ignore the overflow, if the result equals 0, it means the two numbers are complementary. To a number, simply add its complement instead.



Okay... But finding the complement seems kinda hard...



Don't worry, there is a really easy way to find a two's complement. Just follow these steps.

LET'S FIND THE TWO'S COMPLEMENT TO DO SUBTRACTION!

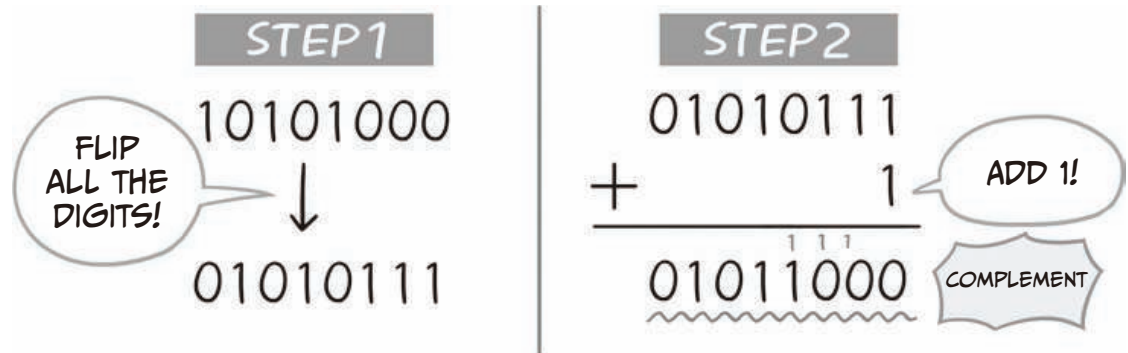
Step 1: Invert all the digits of the first number from 1 to 0 and vice versa. (This is also called finding the one's complement.)

Step 2: Add 1 to this inverted version of the number.

And you'll end up with the two's complement!



Sweet! I tried finding the complement of that last example. Using this method, it was easy.



Computers (actually the ALUs) use this type of reasoning all the time for arithmetic operations (addition and subtraction). The only difference is that most ALUs perform subtraction by adding the first number and the inverted second number. Then finally, they add 1 to that sum. The order of operations is different, but the end result is the same, right?

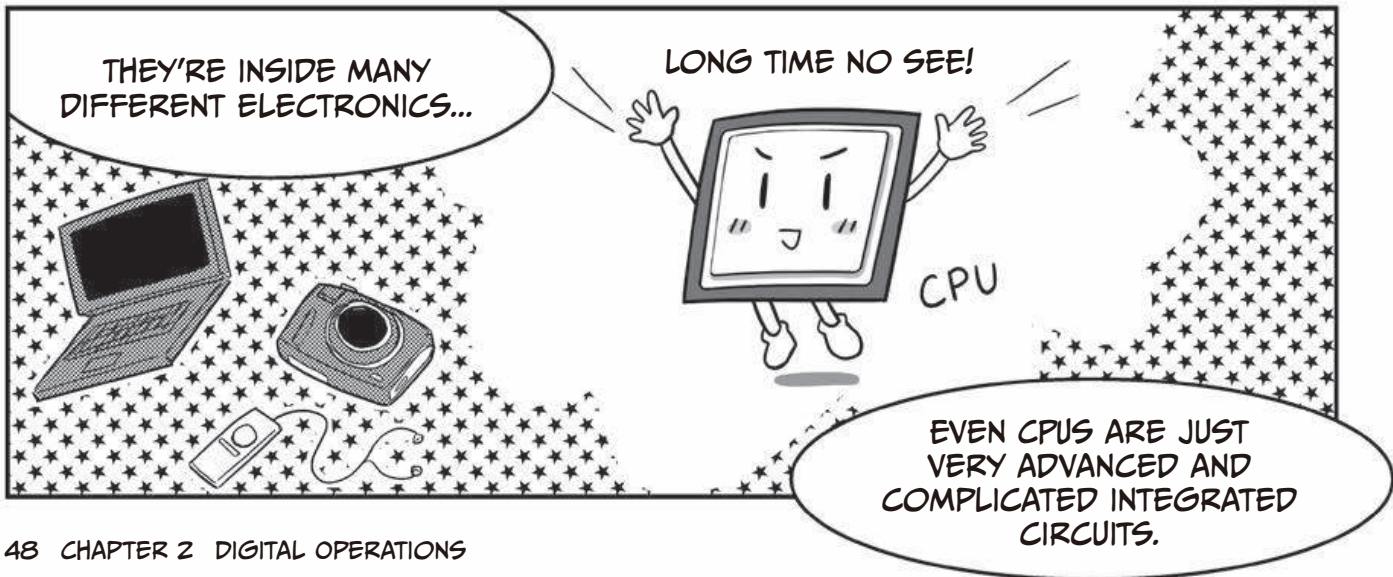
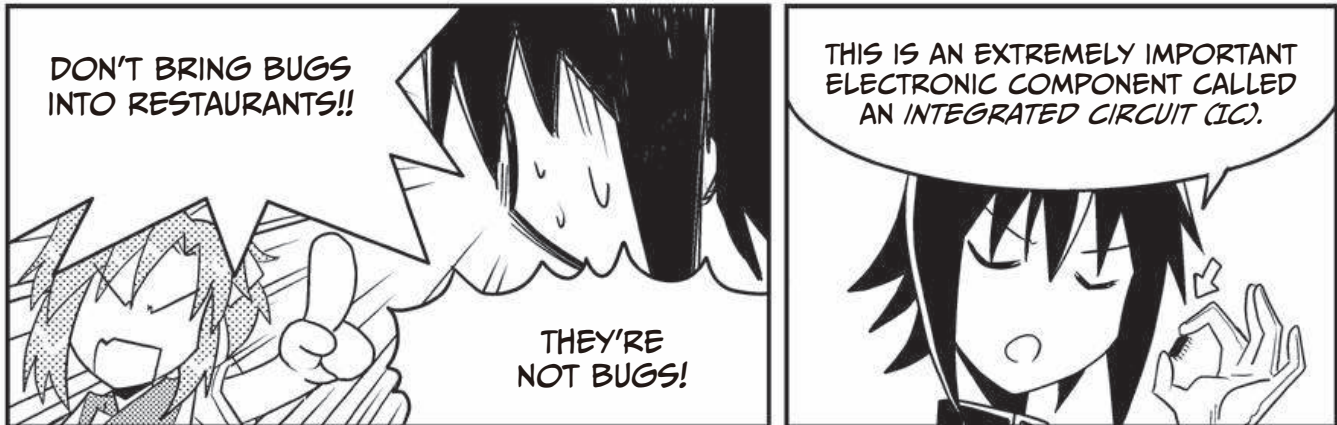
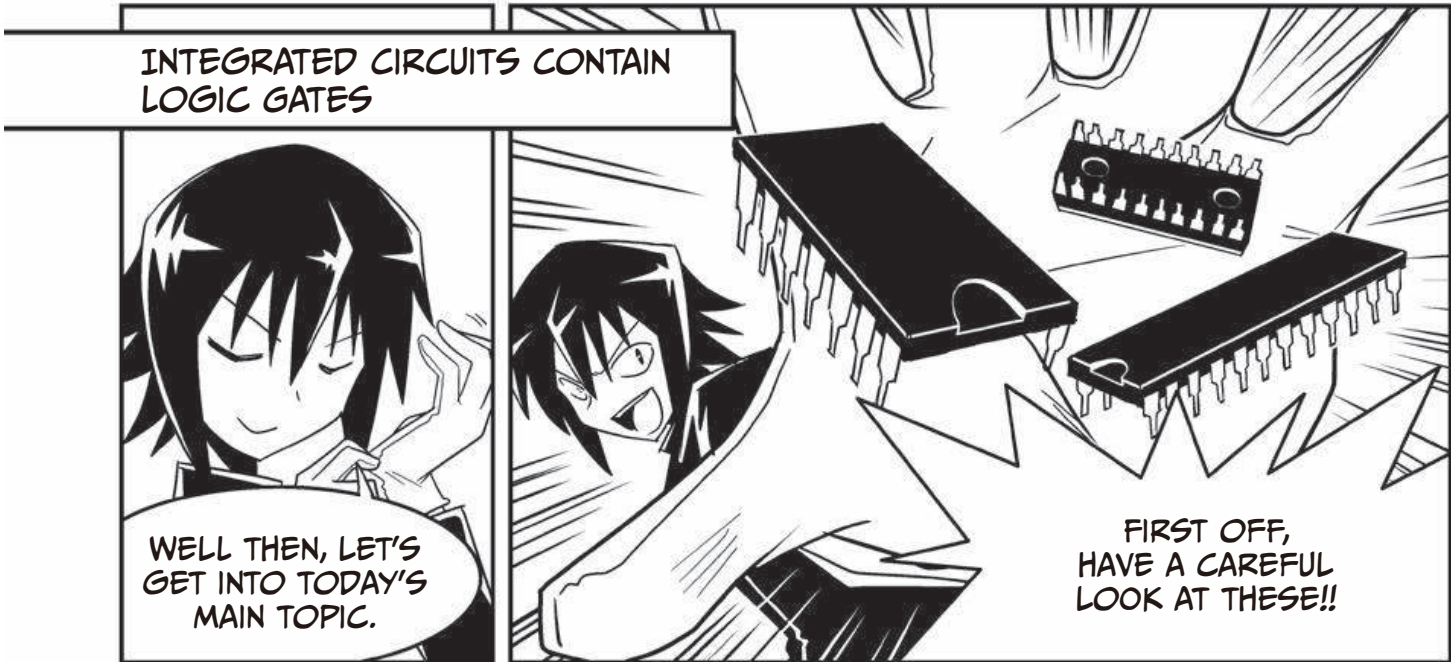
And since the computer calculations only deal with 1s and 0s, this method is both really simple and incredibly fast at the same time.

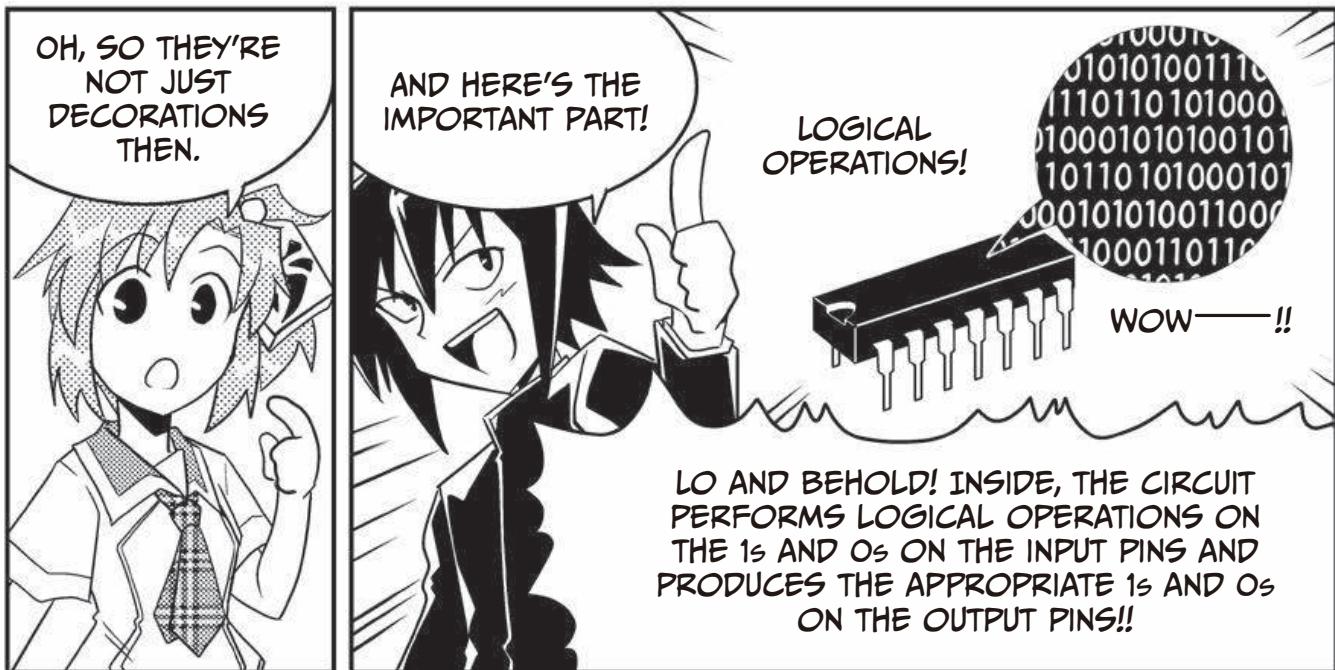


I see. So there are some merits to binary, I suppose!

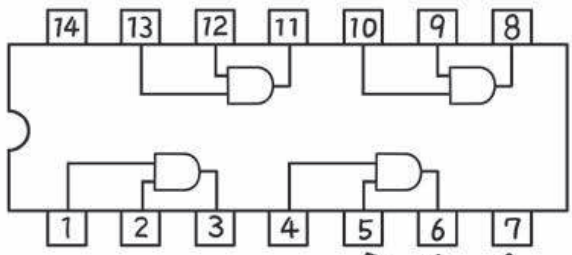


What Are Logical Operations?





FIRST, I WANT YOU TO GET THE GENERAL IDEA. THE INSIDE OF AN INTEGRATED CIRCUIT LOOK SOMETHING LIKE THIS...



THIS IS A LABELED DIAGRAM OF THE INSIDE OF THIS CHIP.

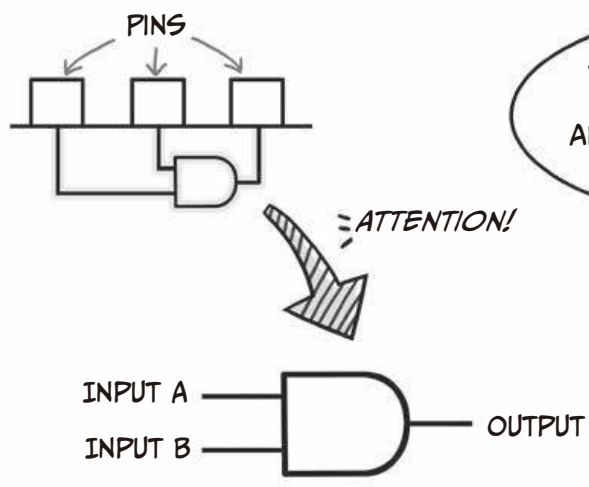
↑ ↑ ↑
PINS
SCRITCH

THIS IS A 74LS08 INTEGRATED CIRCUIT.



HMM. YEAH, I CAN SEE THAT THERE ARE FOUR SYMBOLS THAT LOOK THE SAME, AND THEY SEEM TO BE CONNECTED TO THREE PINS EACH...

NOW LET'S FOCUS ON ONE OF THOSE SYMBOLS.




LOOKING CLOSELY, YOU CAN SEE THAT THEY EACH HAVE TWO INPUTS AND ONE OUTPUT. WE CALL EACH OF THESE PINS A LOGIC GATE.



I SEE, SO THAT MEANS... LOOK AT THE NEXT PART!

EACH LOGIC GATE IS LIKE A MAGIC BOX WHERE YOU GET SOME OUTPUT IF YOU PUT THINGS INTO THE INPUTS!




AND THE INPUTS AND OUTPUTS ARE, OF COURSE, 1s AND 0s... YEAH.

EACH INPUT AND THE OUTPUT CAN EITHER BE 1 OR 0.

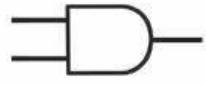

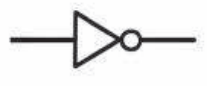
YEAH, THAT'S RIGHT.

THE THREE BASIC LOGIC GATES (AND, OR, AND NOT)

THEN LET ME USE YOUR MAGIC BOX ANALOGY AS WE GET INTO THE SPECIFICS.



AMONG THE LOGIC GATES, THE MOST BASIC 1s ARE THESE: THE AND GATE, THE OR GATE, AND THE NOT GATE.

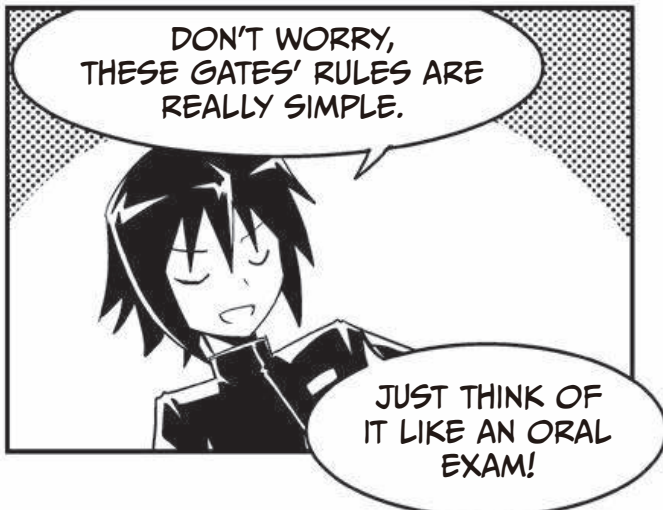
AND	OR	NOT
		

MEMORIZE ALL OF THEM TOGETHER!!

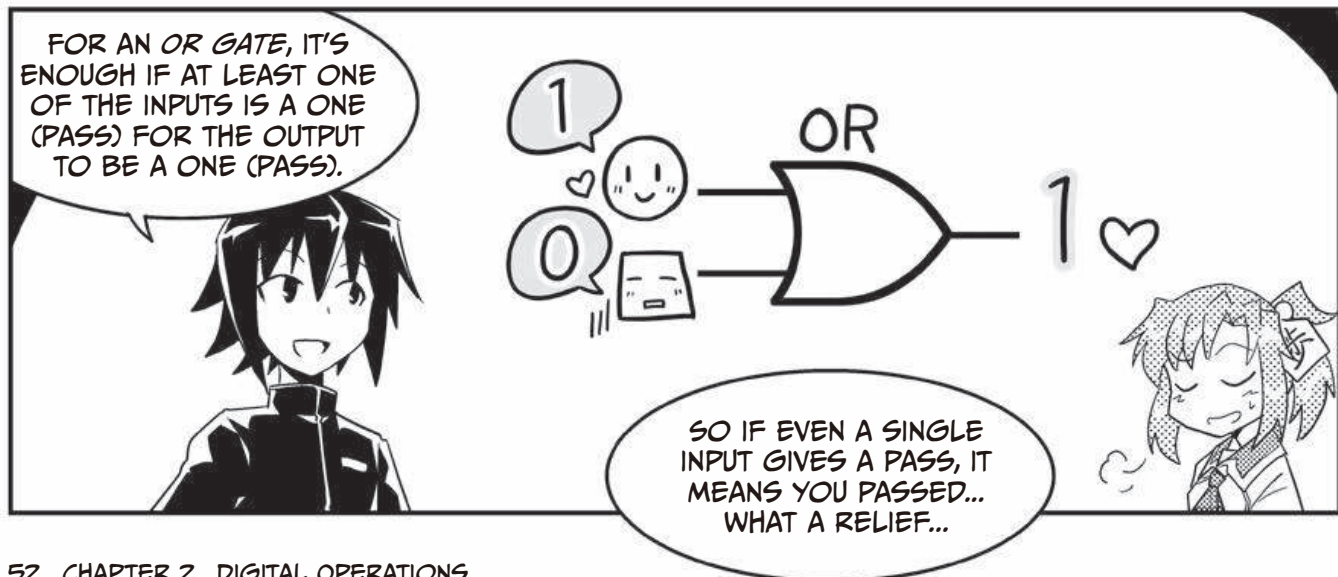
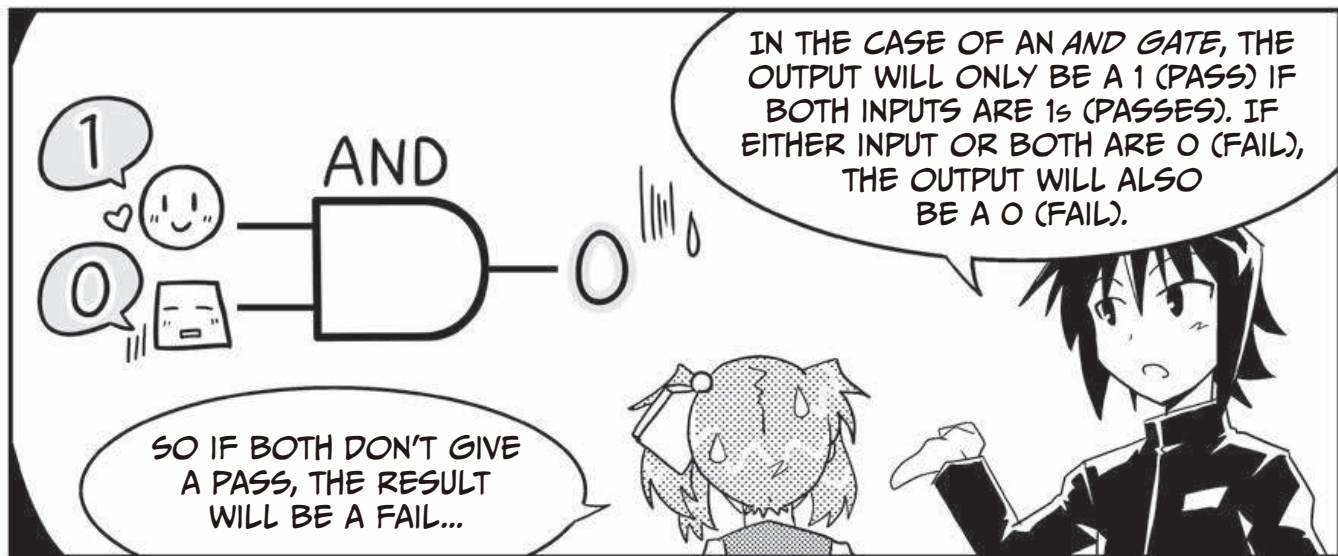
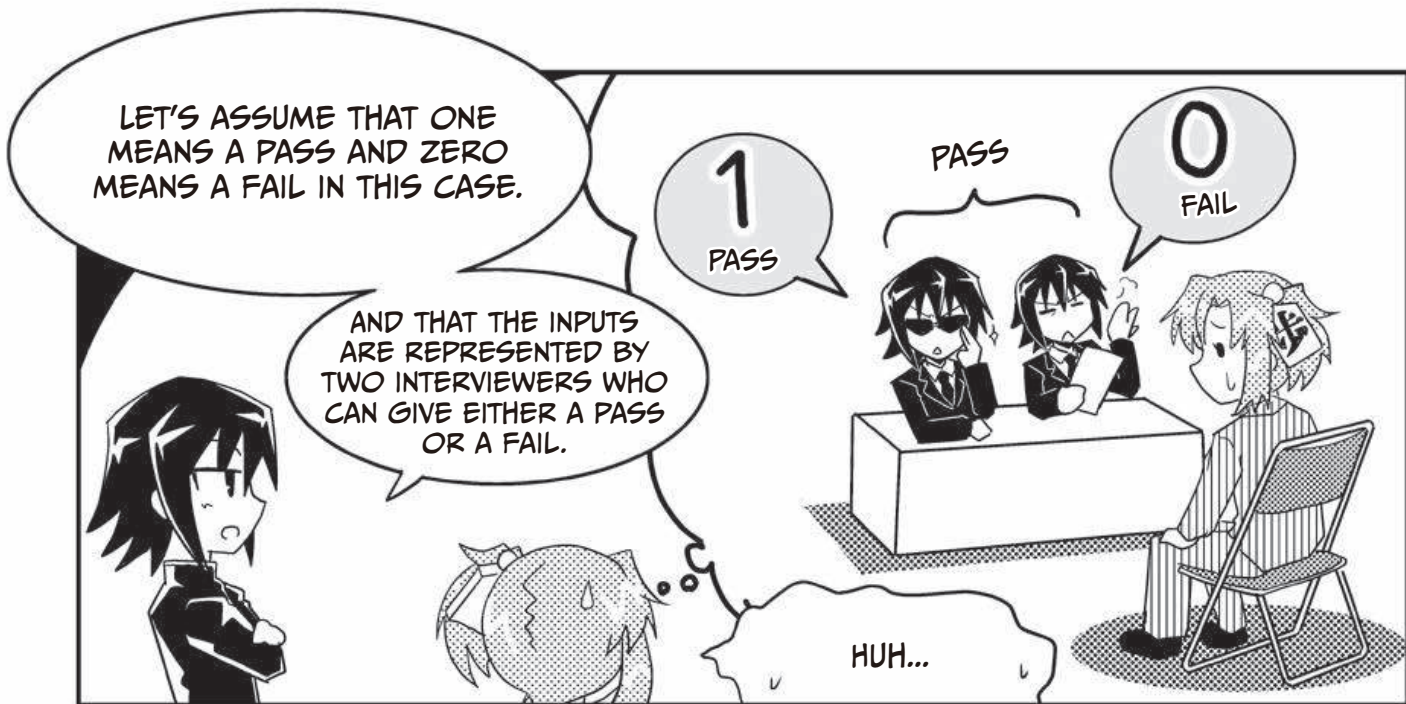


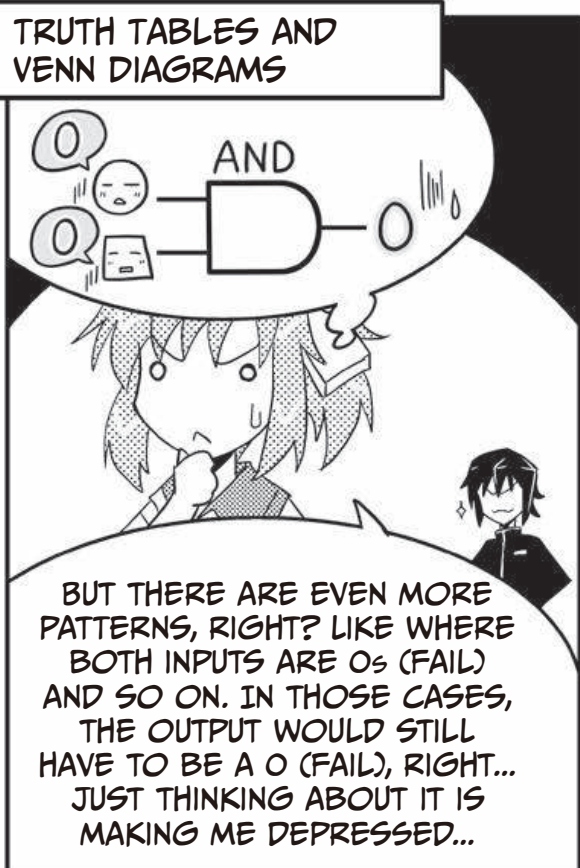
ALL OF THEM?? IS THIS A BOOTCAMP?!

DON'T WORRY, THESE GATES' RULES ARE REALLY SIMPLE.



JUST THINK OF IT LIKE AN ORAL EXAM!






THIS IS IT!
BURN IT INTO
YOUR MIND!!!

SWAT-

TRUTH TABLE FOR
THE AND GATE

INPUT		OUTPUT
A	B	Z
0	0	0
1	0	0
0	1	0
1	1	1



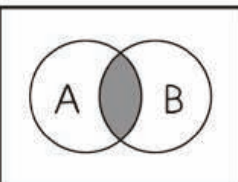
IF BOTH A AND B ARE 0,
THE OUTPUT IS 0.

IF A IS 1 AND B IS 0,
THE OUTPUT IS 0.

IF A IS 0 AND B IS 1,
THE OUTPUT IS 0.

IF A AND B ARE BOTH 1,
THE OUTPUT IS 1.

OOOH! YOU CAN SEE ALL THE
INPUT AND OUTPUT POSSIBILITIES.
THAT'S SUPER USEFUL!!



ALSO, WHEN THINKING
ABOUT LOGIC GATES,
VENN DIAGRAMS ARE
REALLY HANDY.

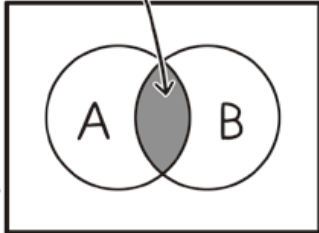
OH, I REMEMBER
THOSE FROM
JUNIOR HIGH.

YES, BUT THE IMPORTANT
THING HERE IS THAT THESE
VENN DIAGRAMS ILLUSTRATE
TWO STATES.

THE AREA INSIDE
THIS RECTANGLE IS A
WORLD THAT CONSISTS
ONLY OF REGIONS
WITHOUT COLOR (0) OR
WITH COLOR (1), OKAY?

SO USING VENN
DIAGRAMS, WE CAN
VISUALIZE THE 1s
AND 0s. NICE!

IN THIS EXAMPLE, THERE
IS ONLY COLOR (1) WHERE
A AND B INTERSECT.



THE AREA INSIDE THE
RECTANGLE IS A WORLD
OF ONLY 1s AND 0s.

THAT'S RIGHT.
THEN LET'S USE
THIS TO TAKE
A LOOK AT THE
THREE LOGIC
GATES AGAIN
ALL AT ONCE,
SHALL WE?

A SUMMARY OF THE AND, OR, AND NOT GATES



Let's summarize the first three basic gates. Let's look at the symbols, truth tables, and Venn diagrams as sets!

AND GATE (LOGIC INTERSECTION GATE)

SYMBOL	TRUTH TABLE	VENN DIAGRAM															
<p>INPUTS OUTPUT</p>	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	A	B	Z	0	0	0	0	1	0	1	0	0	1	1	1	<p>$Z = A \cdot B$</p>
A	B	Z															
0	0	0															
0	1	0															
1	0	0															
1	1	1															

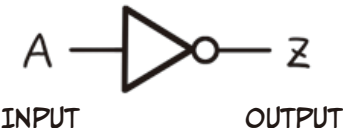
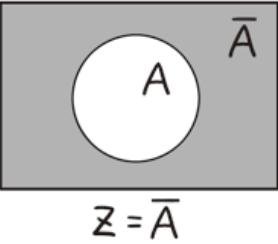
AND gates output 1 only when both inputs are 1 and are sometimes expressed in equation form as $Z = A \cdot B$. The symbols used to represent AND are those for logical intersections: \cdot or \cap .

OR GATE (LOGIC UNION GATE)

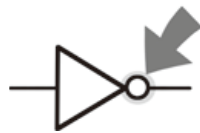
SYMBOL	TRUTH TABLE	VENN DIAGRAM															
<p>INPUTS OUTPUT</p>	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	A	B	Z	0	0	0	0	1	1	1	0	1	1	1	1	<p>$Z = A + B$</p>
A	B	Z															
0	0	0															
0	1	1															
1	0	1															
1	1	1															

OR gates output 1 when either input is 1 and are sometimes expressed in equation form as $Z = A + B$. The symbols used to represent OR are those for logical unions: $+$ or \cup .

NOT GATE (LOGIC NEGATION GATE)

SYMBOL	TRUTH TABLE	VENN DIAGRAM						
 <p>INPUT OUTPUT</p>	<table border="1"> <tr> <td>A</td> <td>Z</td> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	Z	0	1	1	0	 <p>$Z = \bar{A}$</p>
A	Z							
0	1							
1	0							

NOT gates output 0 only when the input is 1 and are sometimes expressed in equation form as $Z = \bar{A}$. The symbol used to represent NOT is the one for logical negation (complement): $\bar{\quad}$.



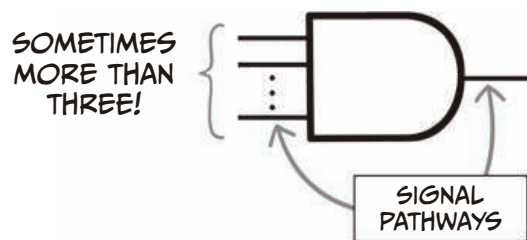
THIS WHITE CIRCLE INDICATES THAT 0 AND 1 SHOULD BE FLIPPED!



Ohh! So you can also write them as $A \cdot B$, $A + B$, or \bar{A} . I think I understand all these forms now.



Okay. Be extra careful about this though! In the examples here, we showed AND and OR gates having only the two inputs A and B, but it's not uncommon for these gates to have three or more inputs.



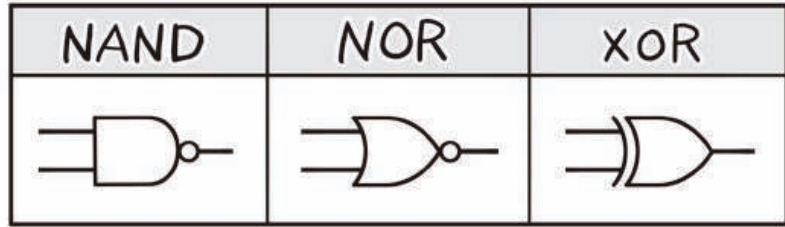
In these cases, we require that all inputs of the AND gate be 1 for the output to be 1. In the case of OR gates, we require that at least one input be 1 for the output to be 1.



So these input and output lines are called *signals* and can either be a 1 or 0. That's easy to remember.

OTHER BASIC GATES (NAND, NOR, AND XOR)

OKAY, LET'S TAKE A LOOK AT NAND, NOR, AND XOR* GATES NEXT.



* XOR IS WRITTEN AS EOR OR EXOR IN SOME CASES.

YOU JUST SAID THAT AND, OR, AND NOT WERE THE THREE BASIC GATES...

TOTTERING

YOU'RE JUST GOING TO TAKE THAT BACK? LIAR! THERE'S EVEN MORE OF THEM?!

STOP WHINING AND CALM DOWN!!

YOU SHOULD KNOW ABOUT NAND, NOR, AND XOR TOO.

AND THE REASON IS...

SOMETHING YOU'LL REALIZE AFTER YOU LEARN ABOUT THEM!!!

EVEN MORE ZEALOUS THAN USUAL!

LET'S DO IT!

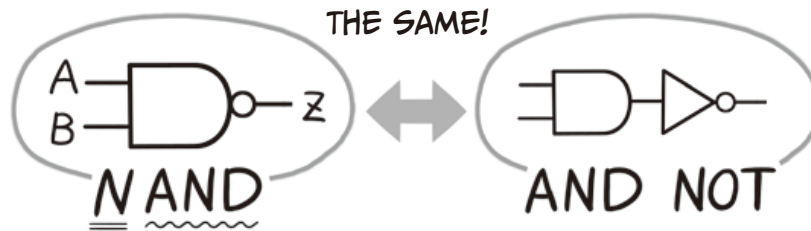
A SUMMARY OF THE NAND, NOR, AND XOR GATES



Okay, let's talk about the other basic gates. Actually, these gates are really just combinations of AND, OR, and NOT gates!

NAND GATE (LOGIC INTERSECTION COMPLEMENT GATE)

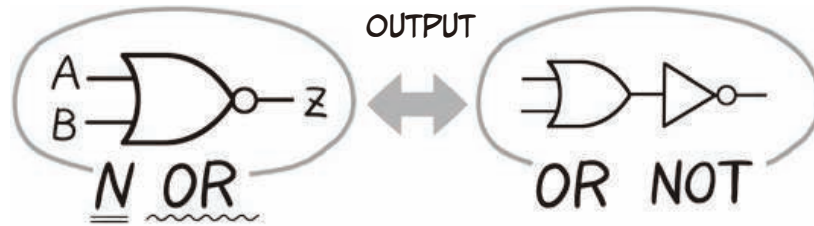
SYMBOL	TRUTH TABLE	VENN DIAGRAM															
	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Z	0	0	1	0	1	1	1	0	1	1	1	0	
A	B	Z															
0	0	1															
0	1	1															
1	0	1															
1	1	0															



The *NAND gate* is an AND gate wired to a NOT gate. The NAND gate's output is therefore the output of an AND gate run through a NOT (negation) gate. It's sometimes written as the equation $Z = \overline{A \cdot B}$.

NOR GATE (LOGIC UNION COMPLEMENT GATE)

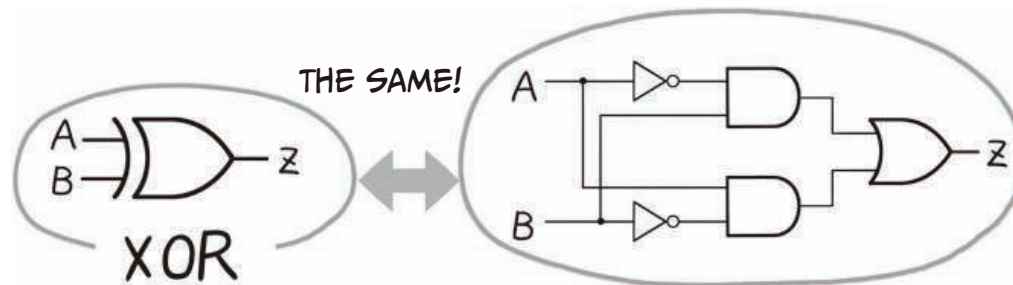
SYMBOL	TRUTH TABLE	VENN DIAGRAM															
	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Z	0	0	1	0	1	0	1	0	0	1	1	0	
A	B	Z															
0	0	1															
0	1	0															
1	0	0															
1	1	0															



The *NOR gate* is an OR gate wired to a NOT gate. The NOR gate's output is therefore the output of an OR gate run through a NOT (negation) gate. It's sometimes written as the equation $Z = \overline{A+B}$.

XOR GATE (EXCLUSIVE LOGIC UNION GATE)

SYMBOL	TRUTH TABLE	VENN DIAGRAM															
	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	A	B	Z	0	0	0	0	1	1	1	0	1	1	1	0	<p>$Z = A \oplus B$</p>
A	B	Z															
0	0	0															
0	1	1															
1	0	1															
1	1	0															



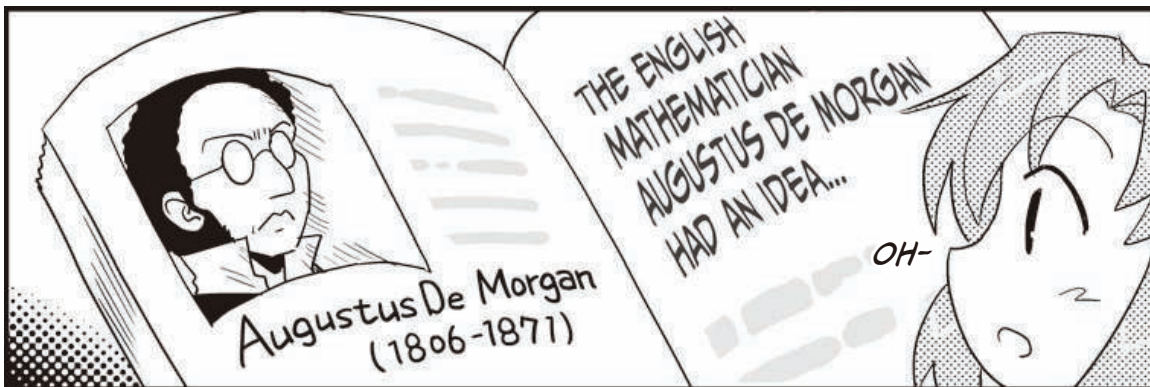
XOR gates output 1 only when the inputs A and B are different. Such a gate is sometimes written as the equation $Z = A \oplus B$.

The XOR gate's function is shown in the schematic above, where you see a combination of AND, OR, and NOT gates. The X in XOR stands for *exclusive*.



Oho! You were right. These gates really are just combinations of basic gates.

DE MORGAN'S LAWS



This might be kind of off topic, but don't you feel a certain fascination whenever you hear the word *theorem* or *law*? It's so charming and cool, I can't help but feel my heart throb wistfully every time... Well, let me tell you about an important theorem! Here it is! De Morgan's indispensable laws for logical operations.

DE MORGAN'S THEOREM

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$



Aah, I might have eaten a little bit too much today. But fast food can be really good sometimes, don't you think?



Stop ignoring me! Well I suppose formulas like this can look complicated at first glance... Let's start with the important part. This law basically just says that you can swap AND for OR operators and vice versa. Does that make it clearer?



Yeah! I can see that the left and right sides have big differences in how they use \cdot (AND) and $+$ (OR). Is it like this?



That's it! It also means that we can use De Morgan's laws to show our circuits in different ways. Using this technique, it's easy to simplify schematics when necessary.



Conversions using De Morgan's laws



But they're completely different! There's really no problem even though the left and right side look nothing alike?



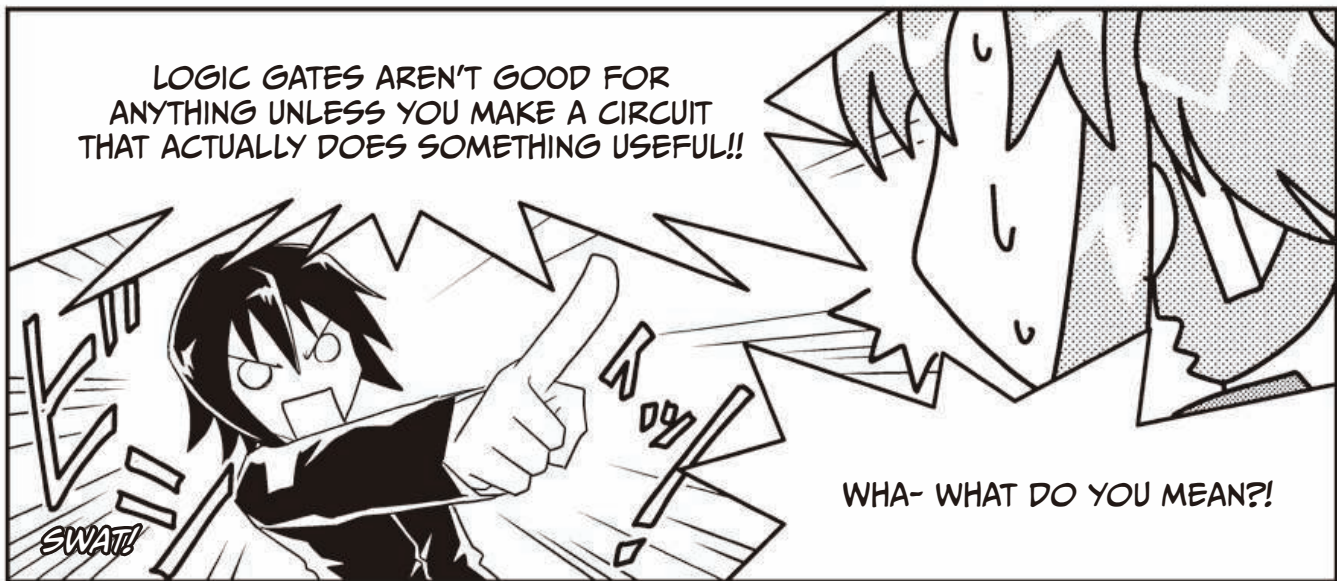
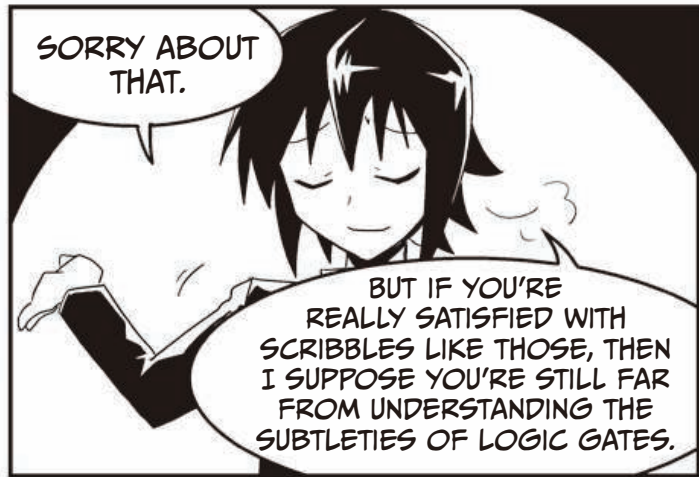
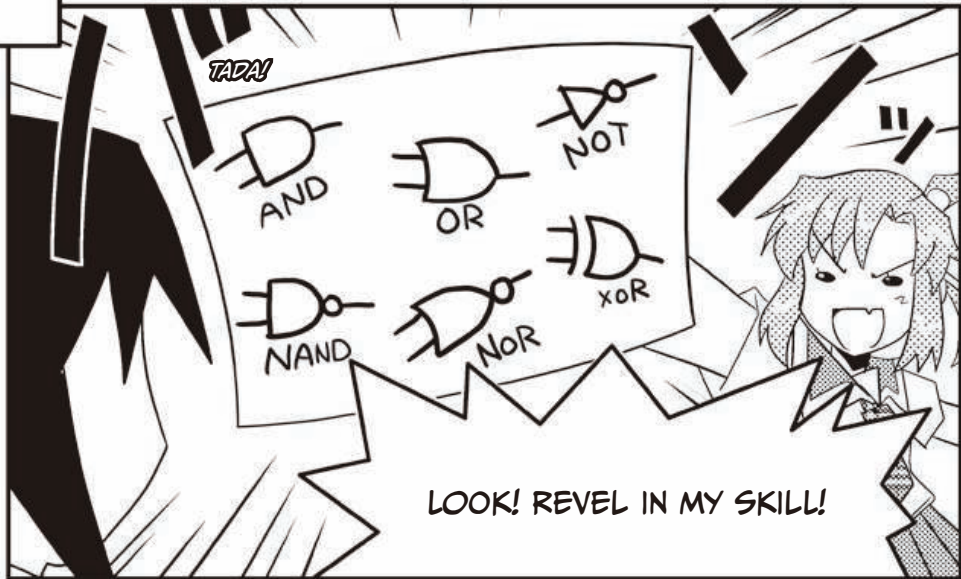
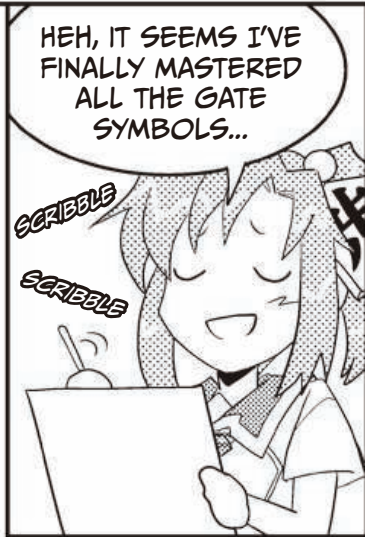
Yeah, the expressions might be different, but their functions are the same. Since logic gates (digital gates) only work with 1s and 0s, everything stays logically the same even if you flip everything. We're just leveraging that particular feature of the math.

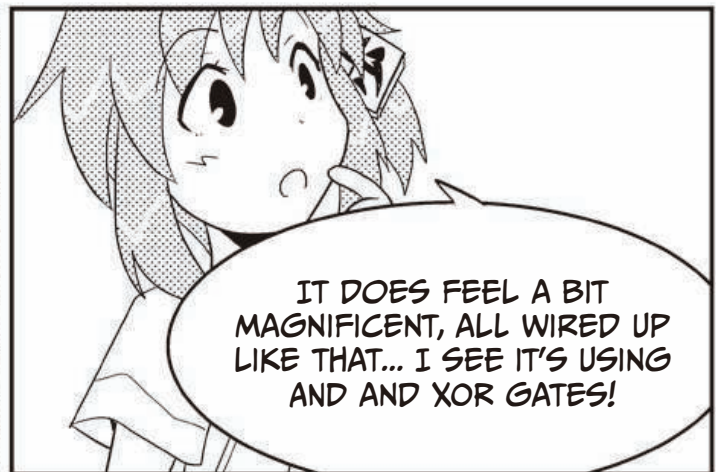
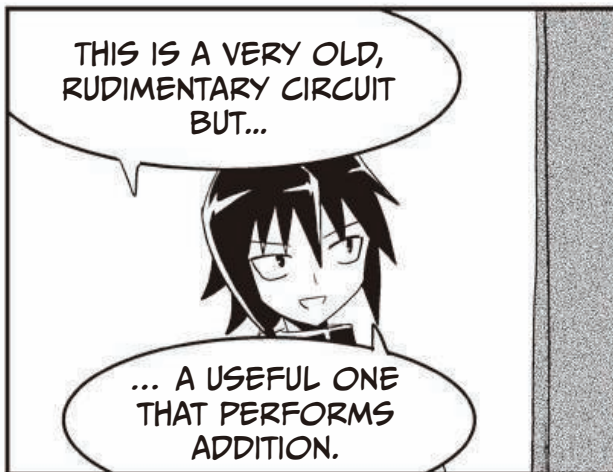
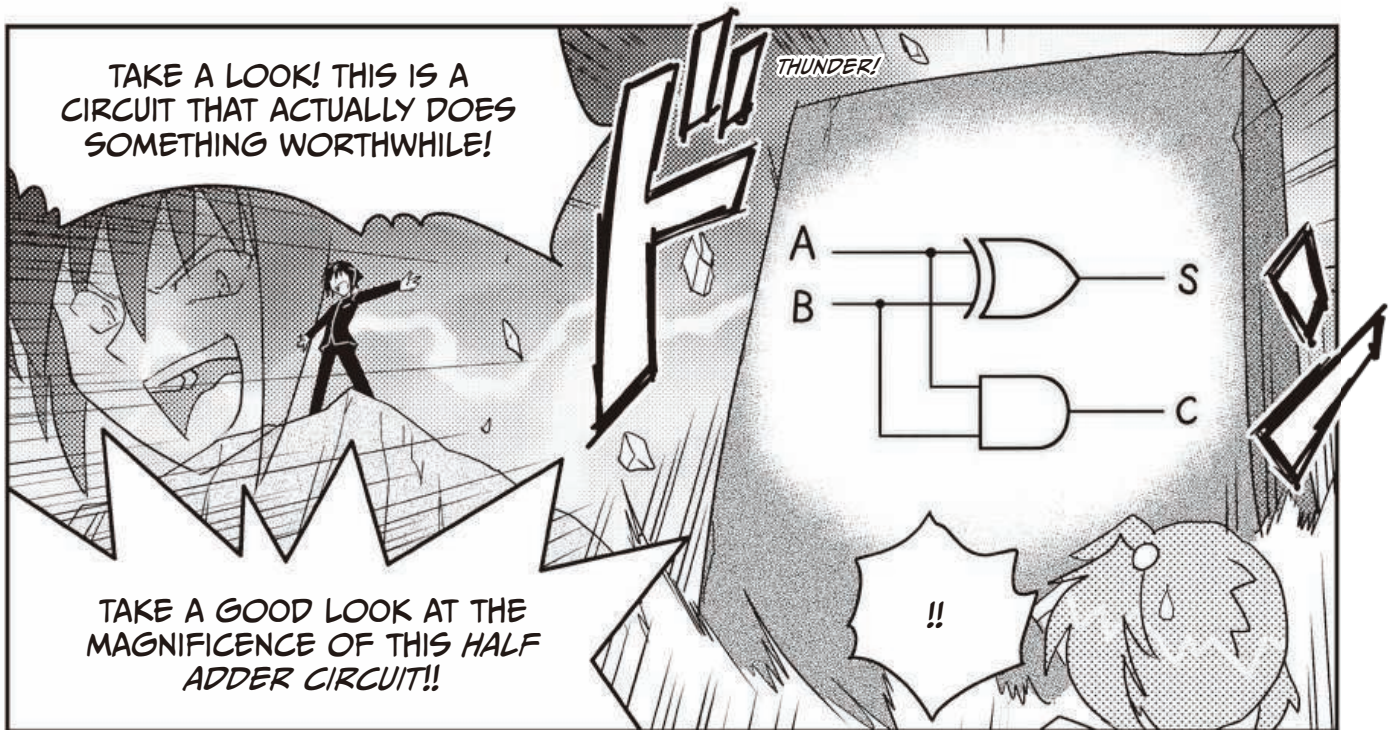


I see... Then you won't mind if I just rewrite all of them then? This is a law I like!

Circuits That Perform Arithmetic

THE ADDITION CIRCUIT





THE HALF ADDER



Let me explain what the *half adder* I showed you is all about. Even though I suspect you won't need that much explaining at this point. First off, do you remember single-bit addition?

$$0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10$$

If we bundle all of these together, it kind of starts to look like a truth table, doesn't it? Let's treat the two bits as inputs A and B, and let's standardize our output to two digits, so an output of 1 looks like 01.

A	B	OUTPUT
0	0	00
0	1	01
1	0	01
1	1	10

(THE DIGIT IS CARRIED.)

↑
THE LOWER DIGIT



Well then, do you notice anything? Pay special attention to the gray area.



Wh—what? Could it be...? The lower digit output... it looks just like an XOR gate's truth table (see page 59)! XOR produces an output of 1 only if the inputs are different, right?



That's correct. This time, look only at the upper output digit.

A	B	OUTPUT
0	+ 0	= 00
0	+ 1	= 01
1	+ 0	= 01
1	+ 1	= 10 (THE DIGIT IS CARRIED.)

↑
THE UPPER DIGIT

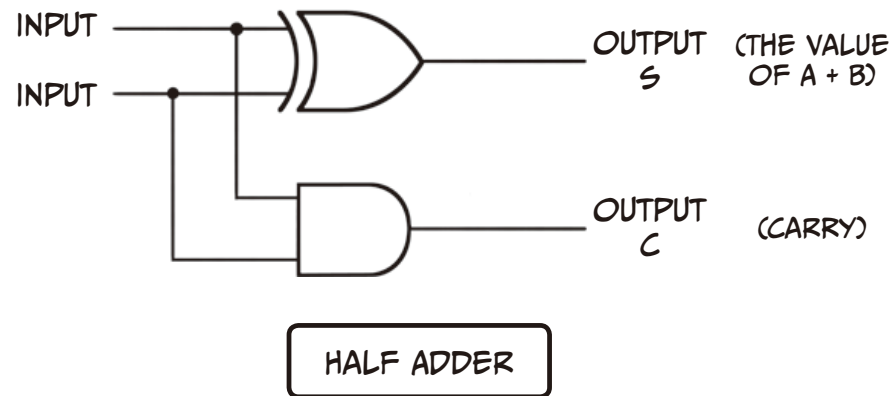


Hmm, that looks just like the truth table for an AND gate (see page 55)! An AND gate's output is 1 only when both inputs are 1. . . . That must mean . . .

That by combining an XOR and AND gate, we can get two outputs (one for the upper digit, one for the lower digit) and perform single-bit addition!

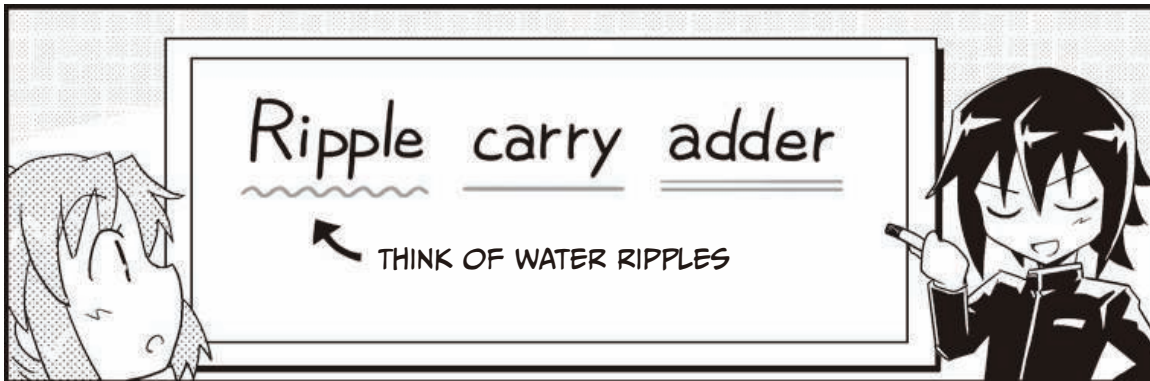


As soon as you get that part, it seems really easy, right? The lower digit comes from output S, and the upper digit comes from output C. In this case, S stands for *sum*, and C for *carry*.



This is how we can get two outputs from two inputs with the same half adder circuit. And this is also how we can add two bits together!

FULL ADDER AND RIPPLE CARRY ADDER



After learning how the half adder works, it seems really simple! Hmm, but, there's still something that bothers me about it.

In that circuit, there's an output for the carry, but there's no input for the carry from the previous digit. That means you can only ever add two single digits, right? That doesn't seem very useful. In fact, only being able to add two single digits seems pretty useless!



Heh, an acute observation for sure. It is true that the half adder cannot deal with carries from previous digits and can therefore only ever add two single bits. That's why half adders are just that, "half an adder." It's no use putting it down for something it can't help.

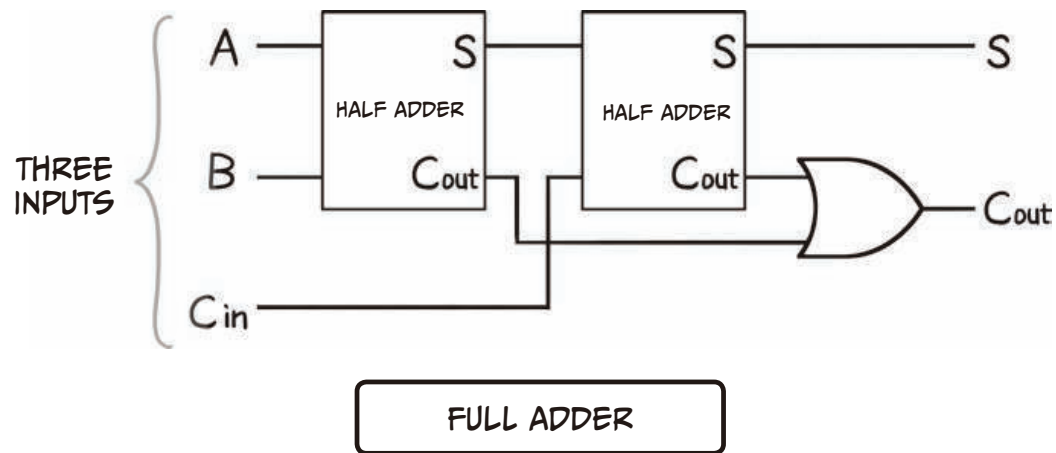


I'm not dissing anyone! Why am I the bad guy all of a sudden?!



Don't underestimate the half adder though! Actually, using two half adders, you can make a *full adder*. In addition to having the inputs A and B, you can use an additional input for the carry in this circuit.

Take a look at this next schematic. We call this circuit with three inputs and two outputs a full adder. We'll put each half adder into its own box to make the diagram a bit easier to understand.

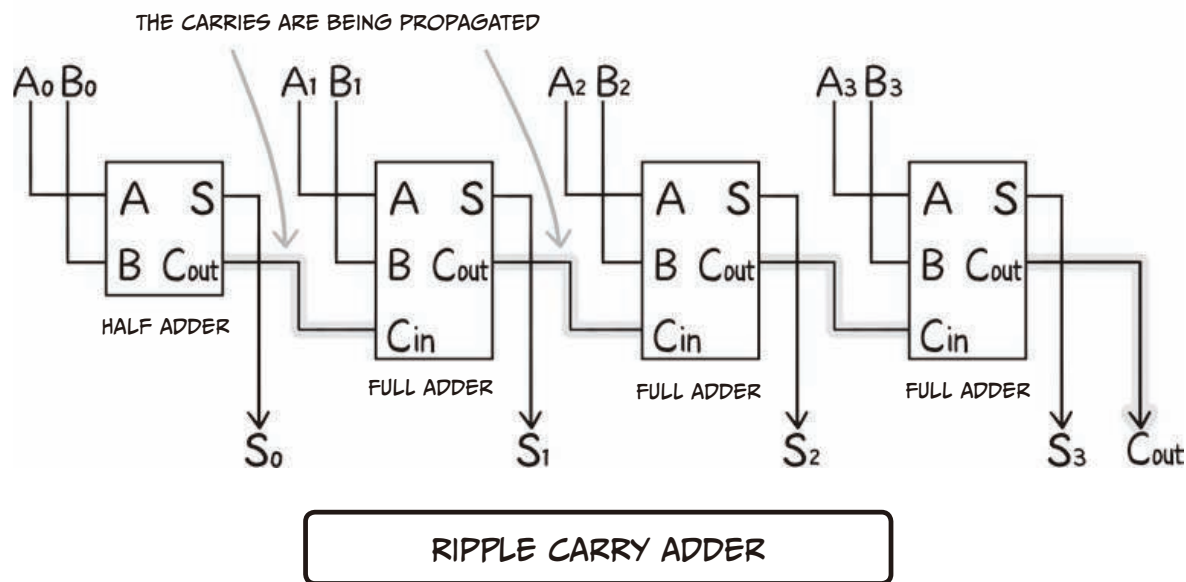


You were right, it's using two half adders! Two halves really make a whole. I guess C_{in} is the carry input and C_{out} is the carry output then.



That's right. And by connecting several of these full adders together, we can add any number of bits! We call a circuit like this a *ripple carry adder*.

In this example, we're using four adders, so we can add four digits. We've also put the full adders into their own boxes. During subtraction, we would deal with the inverse carry (borrow).



Uh-huh. So each adder's output carry goes into the next adder's input carry. This is how the carry flows so that we're able to do the calculation properly.

THE RIPPLE CARRY ADDER AND CARRY LOOK-AHEAD ADDER

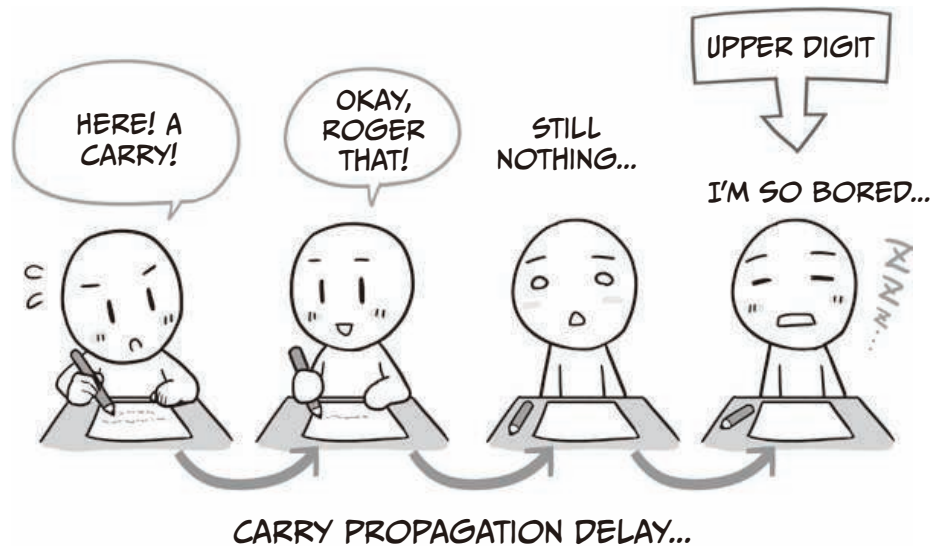


But even then... that ripple carry adder kind of makes me feel a sense of fellowship with how it moves the carry after each step in the calculation. It's really similar to how we humans do calculations on pen and paper, moving the carry from each lower place value to the next higher place value.



Yeah. But that's actually a big problem—it takes a lot of time to keep moving the carry from each calculation to the next.

In *ripple carry adders*, the more digits there are, the slower the calculation speed will be because of the larger *propagation delay*.



AN ARTISTIC IMPRESSION OF A RIPPLE CARRY ADDER

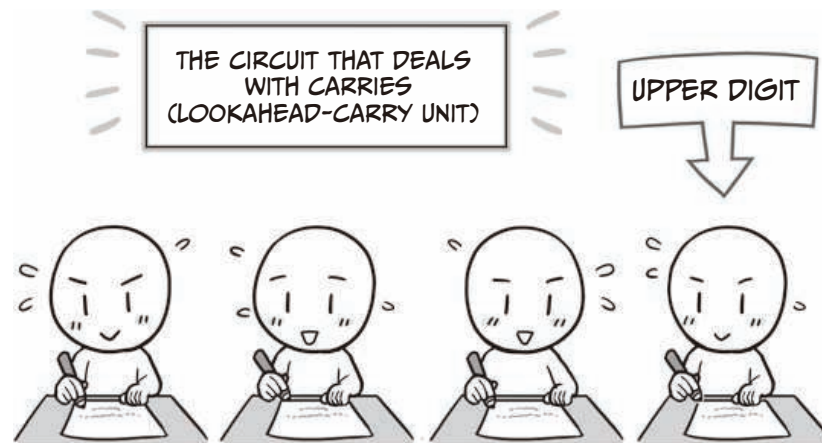


Yeah, that seems a bit slow... Addition and subtraction are pretty common too, so I suppose they're not something you want to be doing slowly. Hmm. So what do we do about it?!



Heh heh heh. To fix this problem, someone came up with what is known as a *carry look-ahead adder*.

It basically delegates the carry calculations to a completely different circuit that serves its results to each digit's adder. Using this method, the upper digits can do their calculations right away, without having to wait!



THEY DON'T HAVE TO WAIT FOR THE CARRY!

AN ARTISTIC IMPRESSION OF A CARRY LOOK-AHEAD ADDER



Eeh, is that even possible? So there's some other dedicated circuit that decides whether or not there's a carry?



Yeah. It determines whether there is a carry in either direction during addition and subtraction. The downside is that the circuit is a lot bigger, but calculation times are drastically reduced.



Hmm. So it's reducing calculation times with all kinds of smart tricks then. When we first talked about making a circuit for addition, I was imagining something pretty small, but the final product is quite impressive.

Circuits That Remember

CIRCUITS WITH MEMORY ARE A NECESSITY!

NOW, LET'S GET INTO TODAY'S LAST TOPIC.

LET'S TALK ABOUT CIRCUITS WITH MEMORY.

O-KAY... THIS MEMORY HAS TO BE THE SAME MEMORY WE TALKED ABOUT LAST TIME, RIGHT?

BACK THEN, YOU SHOWED ME THESE THINGS... (SEE PAGE 18.)

MEMORY!

DATA AND PROGRAM INSTRUCTIONS, AMONG OTHER THINGS USED IN OPERATIONS

HM, YEAH. IT'S TRUE THAT WHEN WE SAY "MEMORY," WE USUALLY MEAN PRIMARY MEMORY LIKE THIS.

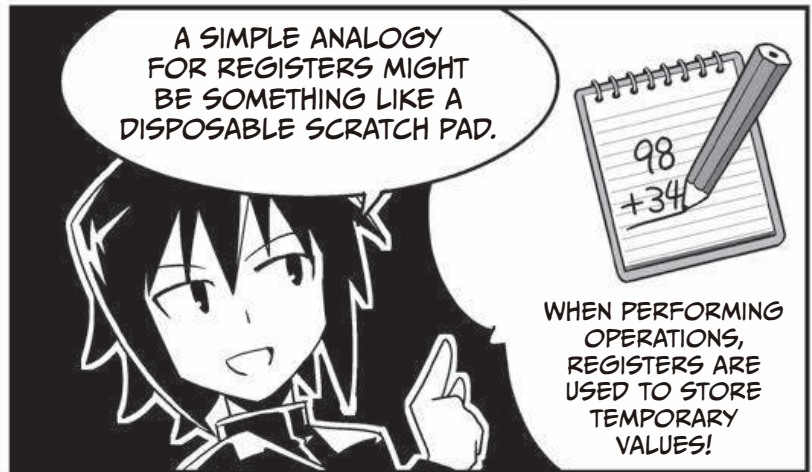
BUT, THERE'S ACTUALLY MEMORY STORAGE INSIDE THE CPU AS WELL.

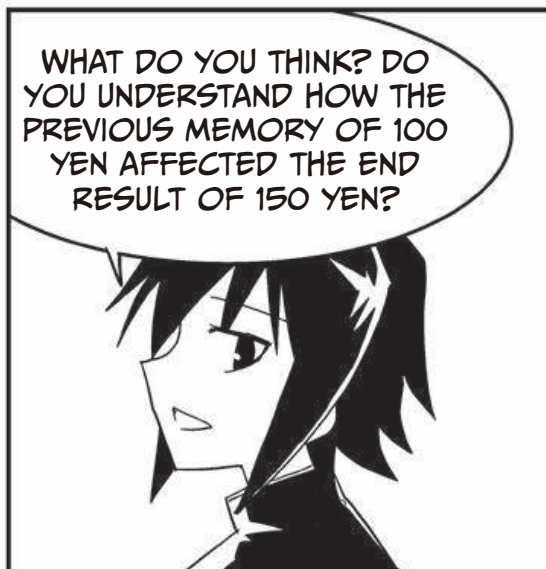
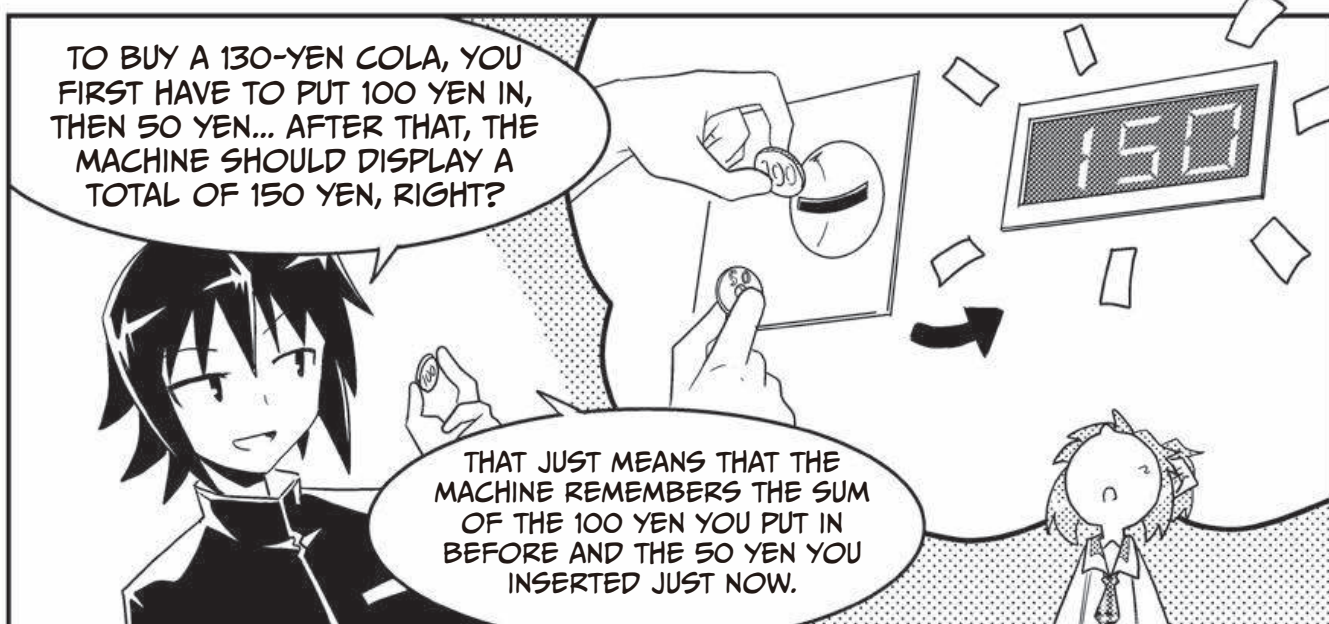
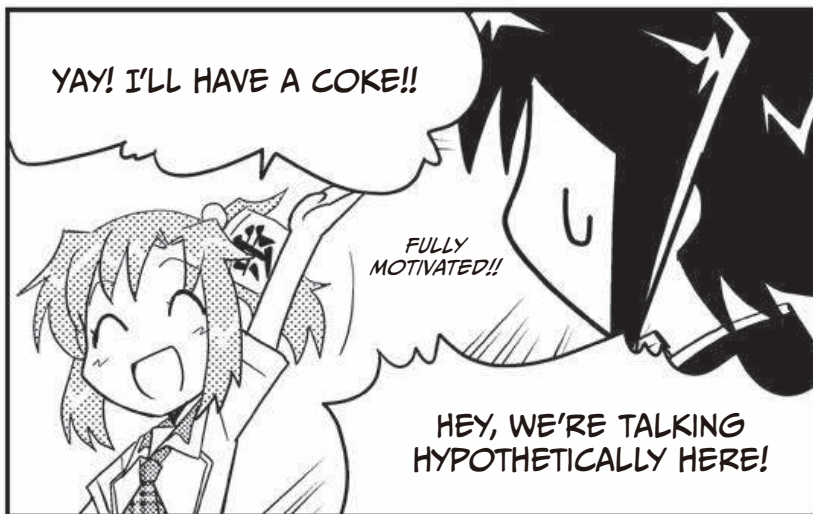
AND THIS STORAGE IS CALLED REGISTERS!!

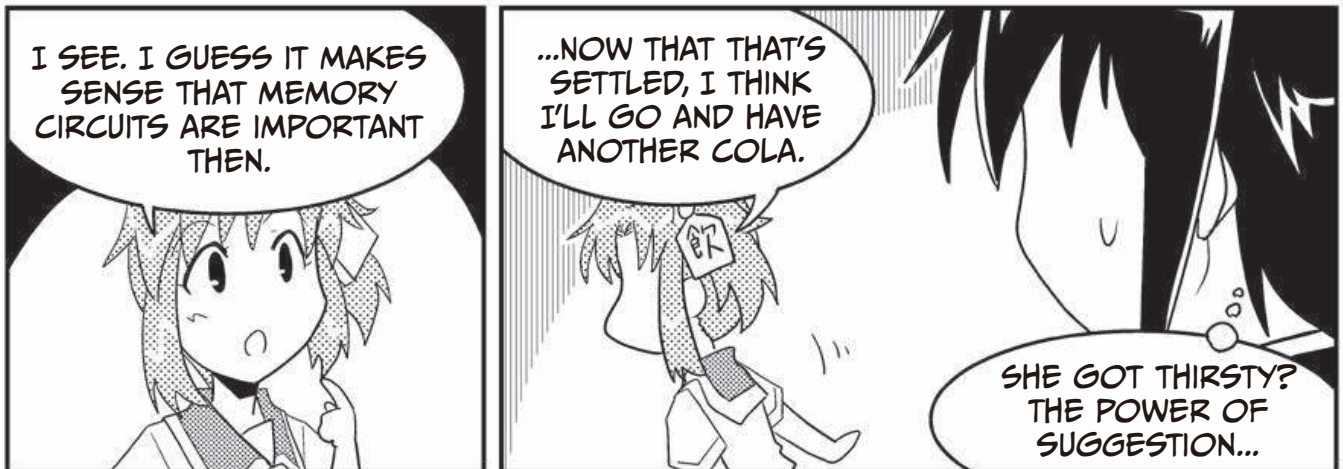
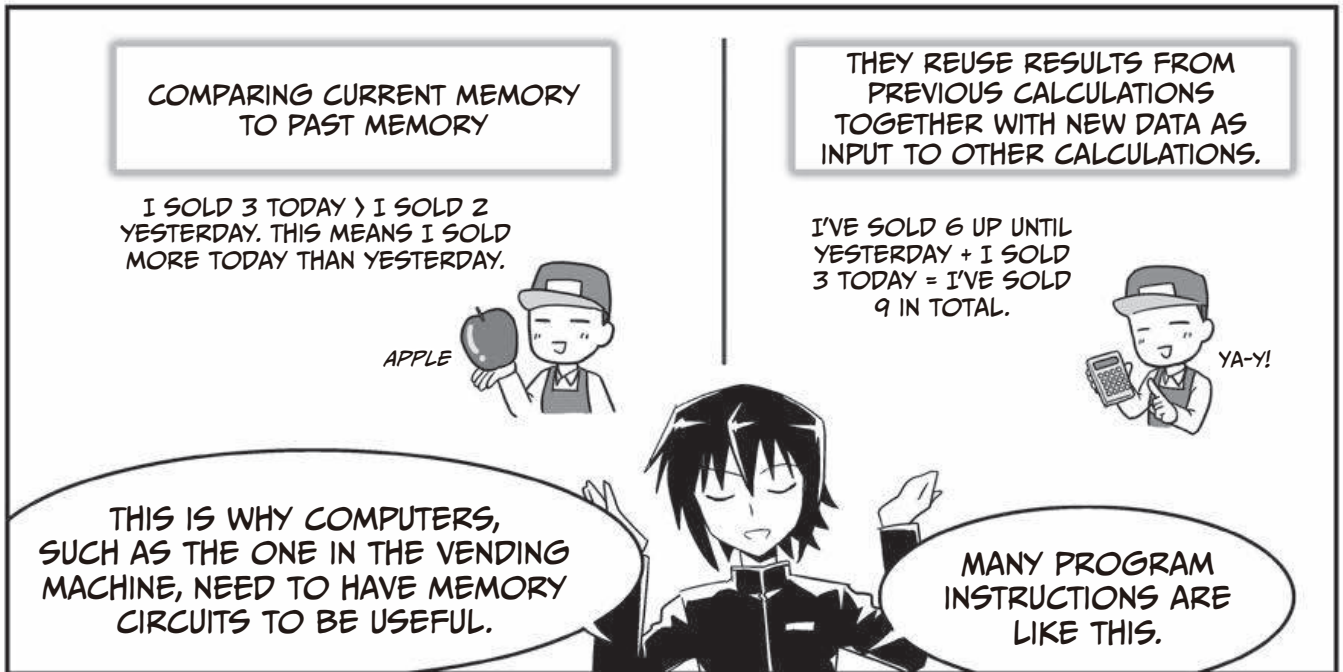
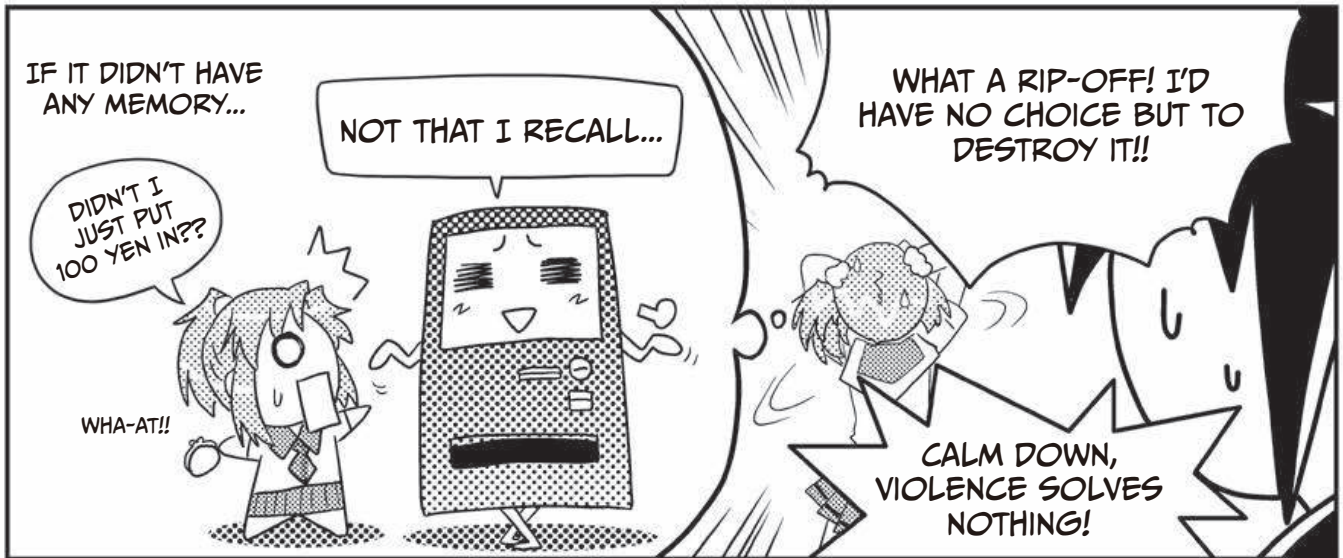
MEMORY!

REGISTERS

CPU







FLIP-FLOP, THE BASICS OF MEMORY CIRCUITS



Ngh. I can't even imagine a circuit that has memory. Even human memory is really complicated you know...



Yeah. You have to think really simple. Computers can only use 1s and 0s right? That means that memory to a computer means somehow storing the states of 1s and 0s.

I've already explained that these 1s and 0s actually correspond to different voltage levels (low and high) (see page 37). This means that to save a 1, we would have to create something that can retain that state over a longer period of time, as in the graph below. We call storing data like this *latching*.



I see. But it's probably not very useful if it just stays in that state forever... What if I want it to go back to 0 later on or I want to overwrite the memory with something else? Wouldn't it make sense to be able to store whatever I want, whenever I want?



Yeah, that's right! For example, if you turned a room's light switch on, it would stay that way until someone turned it off again, and then it would stay off until someone turned it off again. It would be great if we could create some kind of *trigger condition* to freely swap between the 1 and 0 states, just as we do with the light switch.

That is, we would like to be able to store 1s and 0s indefinitely while still being able to flip each bit individually whenever we want. This is exactly what memory circuits do!



Uhm, that sounds a bit selfish, doesn't it? I want to store 1s and 0s, but I also want to be able to flip them at will.



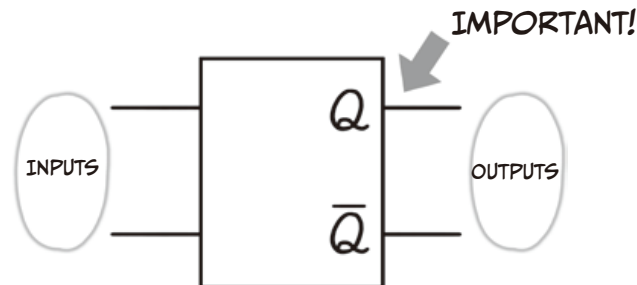
It is selfish, but *flip-flop* circuits grant us the ability to change states! Flip-flops are a basic component of any memory circuit.



Flip-flop...? That's a cute name, but how are they useful?



They're super useful!! First take a look at the picture below. To make it easier to understand, I've put the flip-flop in its own box. Using one of these, we can store one bit of data.



The reason why there are no concrete symbols for the inputs is that they change depending on the type of flip-flop we use.



O-kay. There are inputs. . . . And two outputs Q and \bar{Q}



Yes. Pay special attention to the Q output! This is the output that will stay either 1 or 0. Q will always be the inverse of \bar{Q} . So, if Q is 1, then \bar{Q} will be 0. \bar{Q} can be very useful to have when designing a circuit, but we're going to ignore it for now.



Uh-huh. Then, how does it work? Tell me what's inside that box!

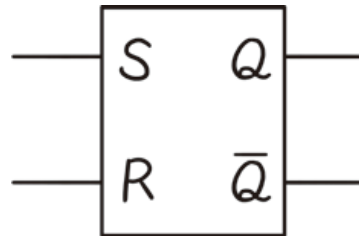


All in good time. First off, there are several types of flip-flops. Both the function and circuit depend on this type. Out of these, I'll teach you about RS flip-flops, D flip-flops, and T flip-flops.

THE RS FLIP-FLOP



Okay, I guess *RS flip-flops* come first? So the box has two input signals, R and S. Rice... Sushi... Rice and sushi?!



THEY'RE ALSO SOMETIMES CALLED RS LATCHES.

YOU CAN ALSO FLIP THE R AND S AND CALL THEM SR FLIP-FLOPS.



Um, no. *R* means *reset* and *S* means *set*. The reset and set inputs are the two main features of this type of circuit.

I might be giving away the main point too quickly here, but setting *S* to 1 will set *Q* to 1 and setting *R* to 1 will reset *Q* to 0. Once *Q* has changed state, removing the input signal won't change it back. It will keep that state until the countersignal (*S* for *R* and vice versa) is sent. As soon as that happens it will, of course, flip the saved state back though.



Hmm, so that means that it remembers which of the two got set to 1 last? If *S* got set to 1 most recently, then the latch remembers 1, and if *R* was the last 1, it remembers 0! Is that it?



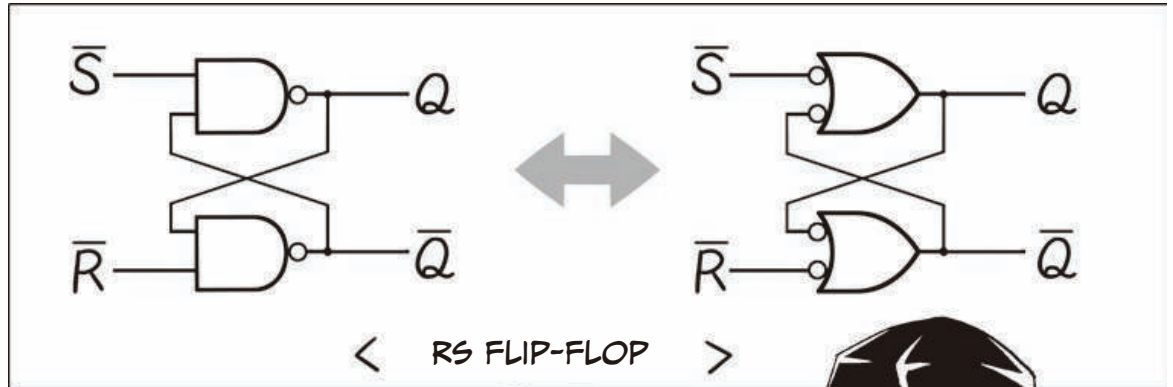
Yeah. It might seem a bit complicated here, but the circuit looks like the figure on the next page. In accordance with De Morgan's laws (see page 60), it can be created using either NAND gates or NOR gates.



Whoa. It looks a bit weird... There are two NAND gates (or NOR gates), but they're all tangled up in figure eights.



Yep! The two circuits are interconnected with the output of one acting as one of the inputs to the other.



INPUTS		OUTPUTS		FUNCTION
\bar{S}	\bar{R}	Q	\bar{Q}	
1	1	DOES NOT CHANGE		RETAINS ITS CURRENT OUTPUT
0	1	1	0	SET
1	0	0	1	RESET
0	0	1	1	NOT ALLOWED



It's thanks to this figure eight that the circuit is able to retain either a 1 or a 0. We call this a latch. You could say that this figure eight is the most important characteristic of a memory circuit!



Hmm, even so, it's pretty complex. If I look back and forth between the schematic and the truth table, I get the feeling I kind of get it, but still...

Let's see, the part of the truth table that says "does not change" means that output Q either stays a 1 or a 0 indefinitely, right? But what does the "not allowed" on the bottom mean? What's not allowed?!



Ah, yeah. That just means that you are not allowed to trigger both set and reset at the same time. Remember that since the circuit is active-low, this means that both outputs can't be 0 at the same time. If you were to set both to 0, this would make both Q and \bar{Q} output 1 until you changed one of them back—but the outputs are always supposed to be either 0 and 1, or 1 and 0. It's not allowed to invalidate the rules we set for this logic circuit.

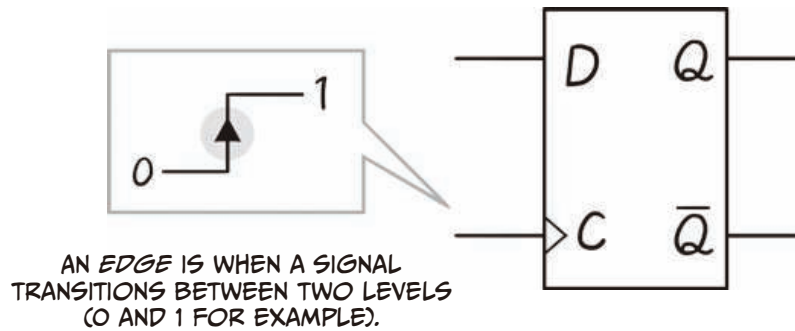


Oh, I see. So just follow the traffic, er, circuit rules, right?

THE D FLIP-FLOP AND THE CLOCK



Let's see. The next one is the *D flip-flop*, I think. The inputs are D and... what's this triangle next to the C?! It looks like that piece of cloth Japanese ghosts wear on their headbands!!



That observation is pretty far removed from the computer world. But I suppose it's a bit cryptic and warrants an explanation. First off, it's easiest to think of the *D* as being for *data*. That triangle is the symbol for a rising edge, and the *C* stands for *clock*.

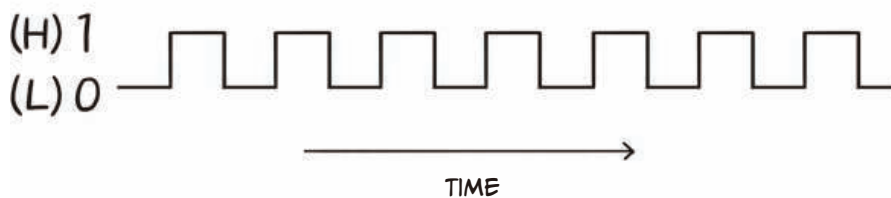


Um... Rising edge?? And the clock—is that just a normal clock?



That's right! Computers need some kind of fixed-interval digital signal to synchronize all the operational states in their circuits. That's what the clock does!

Just like a normal clock measuring time, it flips between high and low voltage (1 and 0) in fixed intervals. It has nothing to do with the circuit's input or output though—it's completely separate.



A CLOCK



Hmm. It really reminds me of a clock... Ticktock, ticktock... Just like we plan our days with the help of clocks, I guess circuits need them, too.



Yeah. When a circuit needs to take some action, the clock can sometimes act as its cue. And inside the clock, what is known as the rising edge acts as that *action signal*. Have a look!



Ohh! Those arrows are at even intervals on the clock graph.



When the clock goes from low to high (0 to 1), we see a rising edge, and when it goes back from high to low (1 to 0), we see a falling edge.

RISING EDGE	FALLING EDGE
WHEN THE CLOCK GOES FROM LOW TO HIGH	WHEN THE CLOCK GOES FROM HIGH TO LOW



Oho, I think I get it. So the rising and falling edges are like ringing bells on the clock, right? When the bell rings, it acts like a signal to take action. Like at the start and end of class, for example.

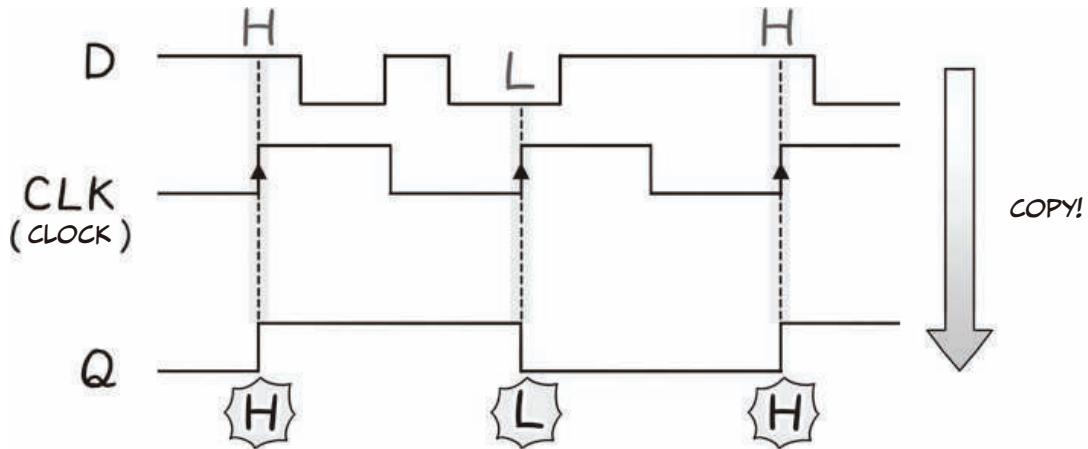


That's just it! That's a pretty good analogy coming from you.



Okay, let's get back to the problem. In a D flip-flop, every time a rising edge passes, the input 1 or 0 at the D input is copied directly to the output Q.

It might be easier to understand by looking at the timing diagram below. A timing diagram is a good way to see how signals change their state over time.

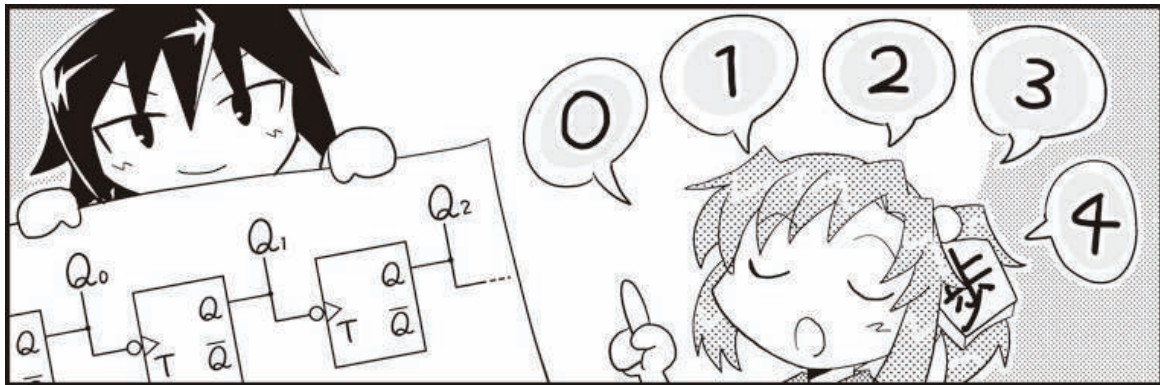


The important lesson here is that the input D can change as much as it wants, but Q won't change until a rising edge arrives!

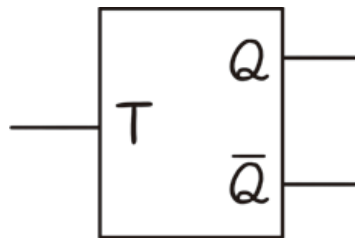


Mmmh. It's a bit complicated, but I think I get it now that I've looked over the timing diagram. In any case, the main characteristic of the D flip-flop seems to be that it acts in sync with the clock's rising edges! Hmm, it seems like clocks are super important both to modern man and circuits.

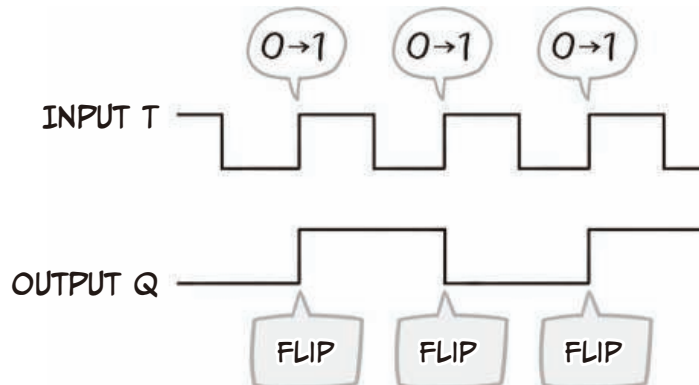
THE T FLIP-FLOP AND COUNTERS



So the last one is the T flip-flop? Wait, it only has one input! Did you forget to draw the rest?



Fuhahaha! Like I would ever forget! The *T flip-flop* only has one input, as you can see, and is pretty simple. Whenever the input *T* changes from 0 to 1, or 1 to 0, the output stored in *Q* flips state. It looks something like this time chart.



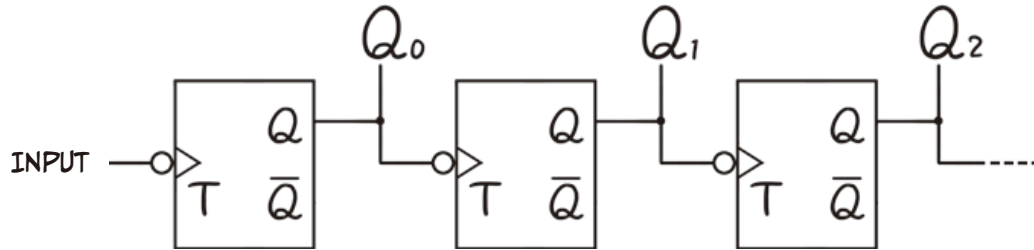
THERE ARE T FLIP-FLOPS THAT ACTIVATE JUST ON FALLING EDGES INSTEAD (1 TO 0).



Oh, this was super easy to understand! It's a memory circuit even though it has only one input.



By the way, flipping between 1 and 0 is called *toggling*. The *T* in T flip-flop actually stands for *toggle*! Also, by connecting several T flip-flops together as in the schematic below, you can make a circuit that can count—a counter circuit.



THIS CIRCUIT SHOWS HOW SEVERAL T FLIP-FLOPS TOGGLED BY THE FALLING EDGE OF AN INPUT SIGNAL CAN ACT AS A COUNTER.

COUNTER CIRCUITS

The first flip-flop will toggle its output state every time the input on the far left changes from high to low. Consequently, the second flip-flop will toggle its output whenever the first flip-flop's output changes from high to low. All following outputs will keep toggling in this pattern. If the input signal is connected to a clock, then each flip-flop in the series will toggle every $2^{(n-1)}$ clock cycles if n is the flip-flop's position in the series. Put another way, the period of each flip-flop's output signal will be 2^n of the original signal's period. Counters that work this way are called *asynchronous counters*, since not all flip-flops are connected to the same clock but, instead, each flip-flop's clock after the first is the output signal of the flip-flop that came before. In contrast, there is a circuit commonly found in CPUs called a *synchronous counter*. As the name here implies, all flip-flops in this type of counter trigger on the signal from the same clock, meaning they all toggle at the same time, in parallel. It's worth mentioning that I've simplified these descriptions to make them easier to understand.

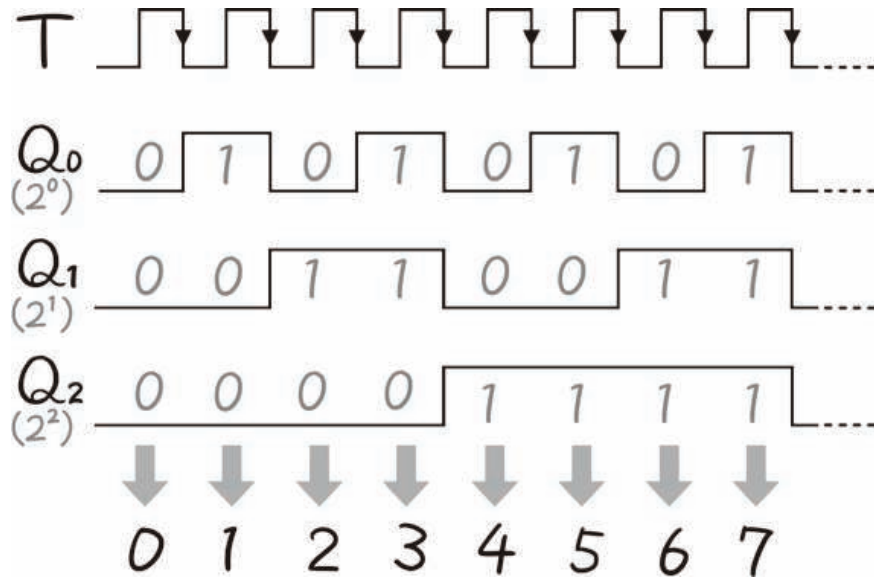


Umm, but why do we say that the circuit can count?



Looking at the time chart, do you see that each output signal has half as many toggles as its input signal? This means that the period of the output signals is twice as long as the period of the input signals. I've put all three of the flip-flops in the schematic above into this time chart so you can see all of their individual outputs next to each other when they are connected.

If you look at each column in this graph individually, you should see that the digits from Q_2 , Q_1 , and Q_0 form binary numbers! Isn't it cool that every time we have a falling edge on the input of the first T flip-flop, this binary number increases by 1? It's counting!



Wow, you're right! Q_2 corresponds to the 2^2 digit, Q_1 to 2^1 , and Q_0 to 2^0 , right?

If you look at Q_2 , Q_1 , and Q_0 in order, the first column forms 000 (the number 0), the second one 001 (1), the third 010 (2), and the fourth 011 (3) in binary. So using this technique, you can actually make the circuit count! That's a really smart design.



Yeah. In this example, we used three flip-flops, so that lets us express 2^3 (8) numbers, meaning we can count from zero to seven.

You can actually make counters from other types of flip-flops, like D flip-flops, for example. Using some other tricks, you can also make circuits that count down if you want.



Oh, that seems like it could be really useful for a lot of things.

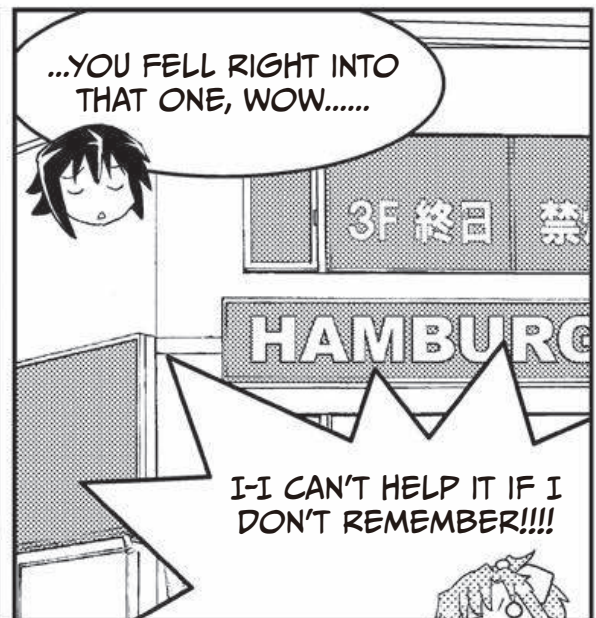
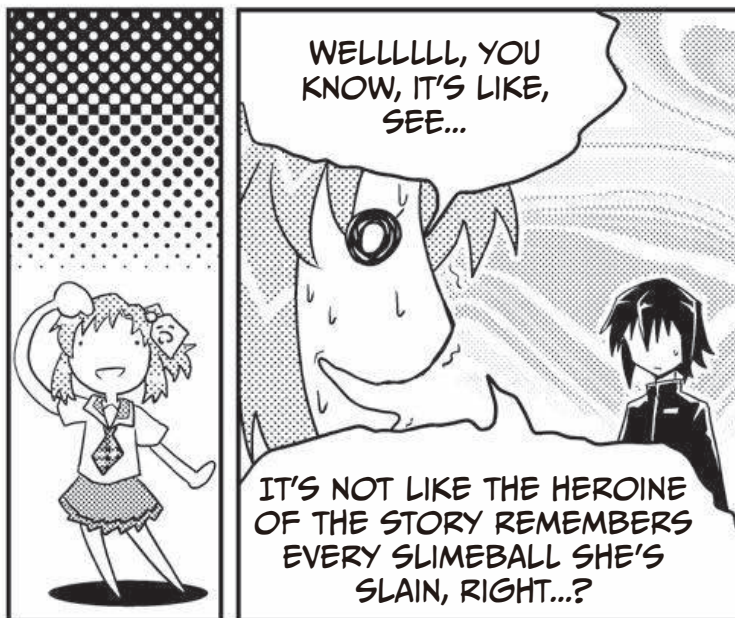
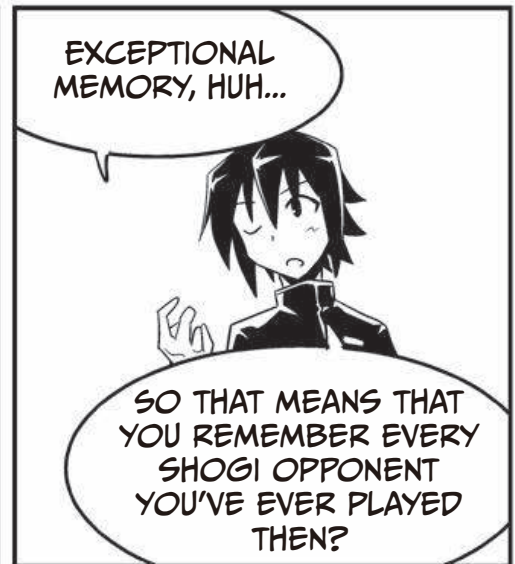
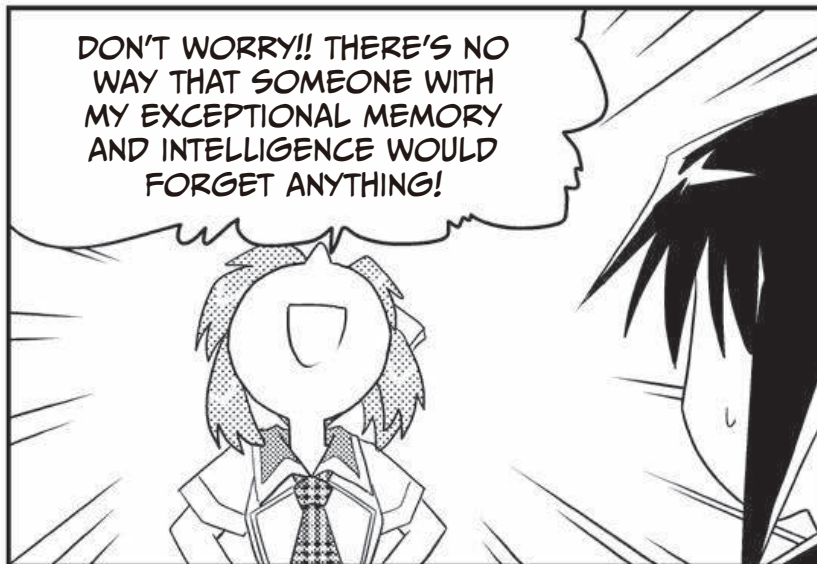


Yeah, well that's it for flip-flops. Just don't forget what I said at the start, that flip-flops are the foundation of any memory circuit!

This means that both primary memory and CPU registers use flip-flops at their core. And flip-flops are also the basis of any counter circuit, just like what we just talked about.



Haha, so they're the base for a lot of different devices, basically. And even though they have a cute name, they're super useful circuits we can't do without!



CIRCUIT DESIGN TODAY (CAD AND FPGA)

Multipurpose integrated circuit design is surprisingly similar to software development these days. It's usually accomplished using a *hardware description language (HDL)* to define the operation of a circuit.

In the past, circuits were drawn using logical circuit symbols, much like the 1s we have shown in this book, but these symbols are now used mostly just for very simple circuits. The development of *computer-aided design (CAD) programs* allows people to design complicated circuits with relative ease.

But, it's important to learn the basics since it can be useful to know these symbols when trying to figure out how data flows through a digital circuit or when trying to understand a particular feature of some schematic.

At the dawn of CPU development, it was common to create reference circuits consisting of many AND, OR, and NOT gates. These were then used when iterating, prototyping, and evaluating new generations of CPUs and other ICs.

By doing this, it was possible to test each function of the advanced circuit individually and even hardwire the circuits together to try to work out problems in the design if some error was detected.

Nowadays, reference circuits like these are rarely used in development. Instead much more flexible field-programmable gate array (FPGA) circuits are preferred.



FPGAS CAN, JUST AS THE NAME SUGGESTS, BE REPROGRAMMED "IN THE FIELD" TO CHANGE THE FUNCTION OF THE IC COMPLETELY. THEY ARE INDISPENSABLE TO CIRCUIT DESIGNERS.

FPGAs consist of a series of logic blocks, which can be wired together in different ways depending on the programming. Some of these blocks contain lookup tables to map the usually available 4–6 bits of input to output in a truth table-like format. The number of lookup tables in an FPGA can range anywhere from a couple of hundred to more than several millions, depending on the FPGA model.

And of course, it's possible to reprogram all of the tables whenever needed. In this way, the same FPGA circuit can be used to perform the functions of many different types of ICs. You can simulate the function of a CPU using an FPGA if you want to, but it's a lot cheaper and easier to mass-produce a dedicated circuit instead. Even so, since the price of FPGAs is dropping and development costs for new ICs are high, if the life span or projected sales of a particular IC are not high enough, it might be more cost-effective to simply use an FPGA.