

THE MANGA GUIDE™ TO

COMICS
INSIDE!

MICROPROCESSORS

MICHIO SHIBUYA
TAKASHI TONAGI
OFFICE SAWA

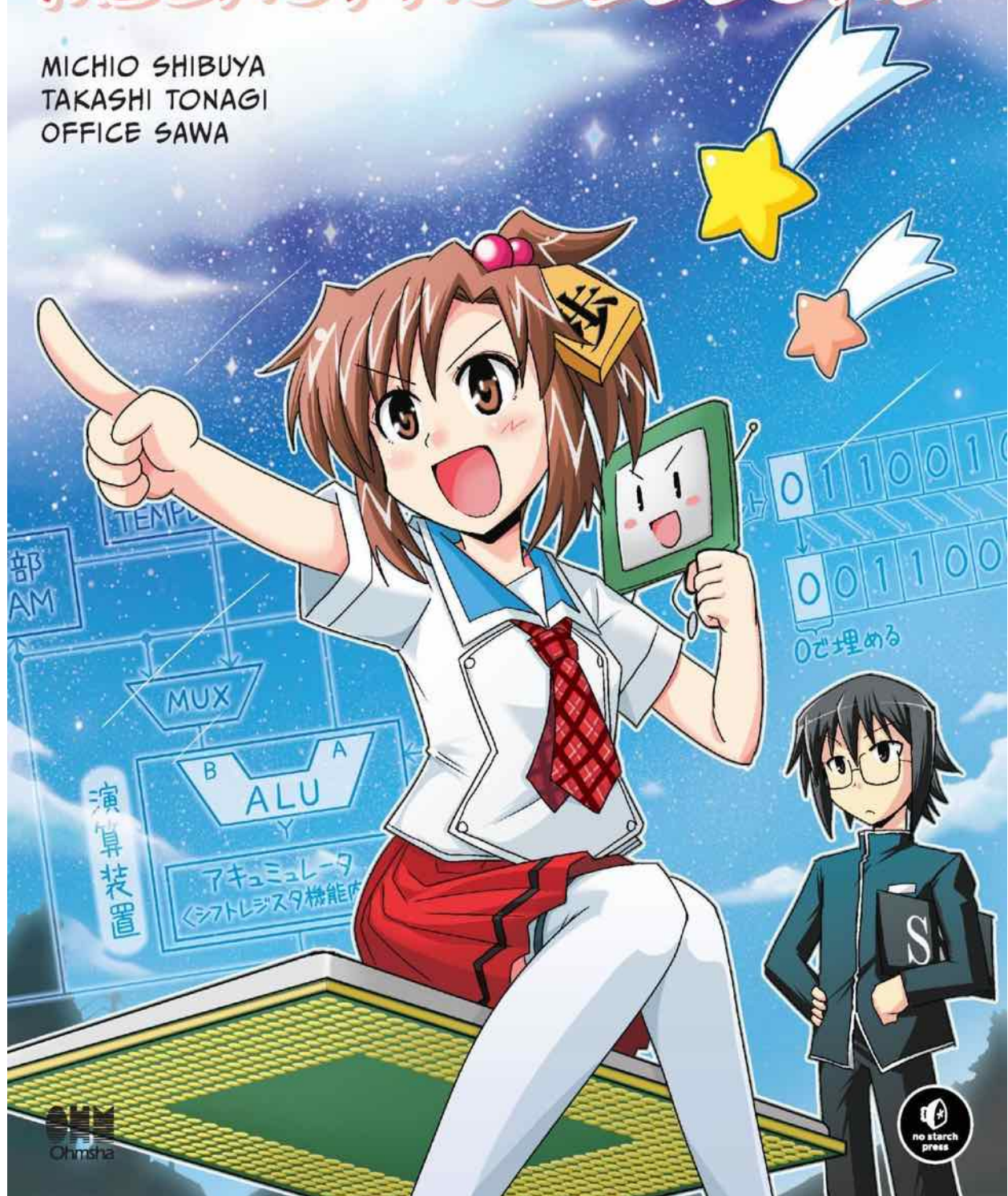


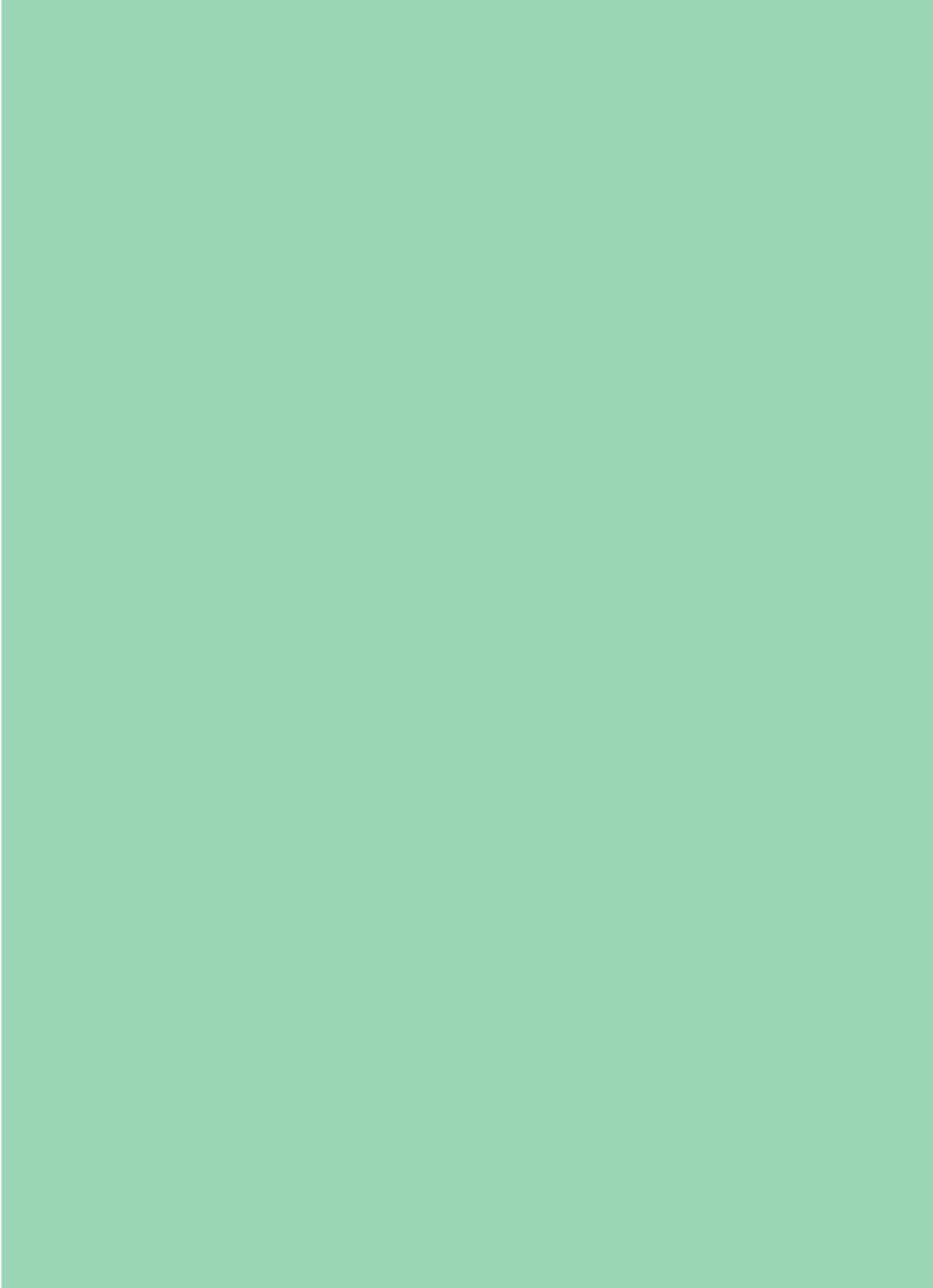
THE MANGA GUIDE™ TO

COMICS
INSIDE!

MICROPROCESSORS

MICHIO SHIBUYA
TAKASHI TONAGI
OFFICE SAWA





PRAISE FOR THE MANGA GUIDE SERIES

"Highly recommended."

—CHOICE MAGAZINE ON *THE MANGA GUIDE TO DATABASES*

"The *Manga Guides* definitely have a place on my bookshelf."

—SMITHSONIAN MAGAZINE

"The art is charming and the humor engaging. A fun and fairly painless lesson on what many consider to be a less-than-thrilling subject."

—SCHOOL LIBRARY JOURNAL ON *THE MANGA GUIDE TO STATISTICS*

"Stimulus for the next generation of scientists."

—SCIENTIFIC COMPUTING ON *THE MANGA GUIDE TO MOLECULAR BIOLOGY*

"The series is consistently good. A great way to introduce kids to the wonder and vastness of the cosmos."

—DISCOVERY.COM

"Absolutely amazing for teaching complex ideas and theories . . . excellent primers for serious study of physics topics."

—PHYSICS TODAY ON *THE MANGA GUIDE TO PHYSICS*

"A great fit of form and subject. Recommended."

—OTAKU USA MAGAZINE ON *THE MANGA GUIDE TO PHYSICS*

"I found the cartoon approach of this book so compelling and its story so endearing that I recommend that every teacher of introductory physics, in both high school and college, consider using it."

—AMERICAN JOURNAL OF PHYSICS ON *THE MANGA GUIDE TO PHYSICS*

"This is really what a good math text should be like. Unlike the majority of books on subjects like statistics, it doesn't just present the material as a dry series of pointless-seeming formulas. It presents statistics as something *fun* and something enlightening."

—GOOD MATH, BAD MATH ON *THE MANGA GUIDE TO STATISTICS*

"A single tortured cry will escape the lips of every thirty-something biochem major who sees *The Manga Guide to Molecular Biology*: 'Why, oh why couldn't this have been written when I was in college?'"

—THE SAN FRANCISCO EXAMINER



"A lot of fun to read. The interactions between the characters are lighthearted, and the whole setting has a sort of quirkiness about it that makes you keep reading just for the joy of it."

—HACKADAY ON *THE MANGA GUIDE TO ELECTRICITY*

"The *Manga Guide to Databases* was the most enjoyable tech book I've ever read."

—RIKKI KITE, LINUX PRO MAGAZINE

"*The Manga Guide to Electricity* makes accessible a very intimidating subject, letting the reader have fun while still delivering the goods."

—GEEKDAD

"If you want to introduce a subject that kids wouldn't normally be very interested in, give it an amusing storyline and wrap it in cartoons."

—MAKE ON *THE MANGA GUIDE TO STATISTICS*

"A clever blend that makes relativity easier to think about—even if you're no Einstein."

—STARDATE, UNIVERSITY OF TEXAS, ON *THE MANGA GUIDE TO RELATIVITY*

"This book does exactly what it is supposed to: offer a fun, interesting way to learn calculus concepts that would otherwise be extremely bland to memorize."

—DAILY TECH ON *THE MANGA GUIDE TO CALCULUS*

"Scientifically solid . . . entertainingly bizarre."

—CHAD ORZEL, SCIENCEBLOGS, ON *THE MANGA GUIDE TO RELATIVITY*

"Makes it possible for a 10-year-old to develop a decent working knowledge of a subject that sends most college students running for the hills."

—SKEPTICBLOG ON *THE MANGA GUIDE TO MOLECULAR BIOLOGY*

"*The Manga Guide to the Universe* does an excellent job of addressing some of the biggest science questions out there, exploring both the history of cosmology and the main riddles that still challenge physicists today."

—ABOUT.COM

"*The Manga Guide to Calculus* is an entertaining comic with colorful characters and a fun strategy to teach its readers calculus."

—DR. DOBB'S

THE MANGA GUIDE™ TO MICROPROCESSORS



THE MANGA GUIDE™ TO
MICROPROCESSORS

MICHIO SHIBUYA,
TAKASHI TONAGI, AND
OFFICE SAWA



THE MANGA GUIDE TO MICROPROCESSORS.

Copyright © 2017 by Michio Shibuya, Takashi Tonagi, and Office sawa.

The Manga Guide to Microprocessors is a translation of the Japanese original, *Manga de wakaru CPU*, published by Ohmsha, Ltd. of Tokyo, Japan, © 2014 by Michio Shibuya, Takashi Tonagi, and Office sawa.

This English edition is co-published by No Starch Press, Inc. and Ohmsha, Ltd.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-817-9

ISBN-13: 978-1-59327-817-5

Publisher: William Pollock

Production Editor: Serena Yang

Author: Michio Shibuya

Illustrator: Takashi Tonagi

Producer: Office sawa

Developmental Editors: Jan Cash and Tyler Ortman

Translators: Fredrik Lindh and Akino Lindh

Technical Reviewer: Dan Romanchik

Copyeditor: Paula L. Fleming

Compositors: Max Burger and Serena Yang

Proofreader: Shannon Waite

Indexer: BIM Creatives, LLC

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900; info@nostarch.com; <http://www.nostarch.com/>

Library of Congress Cataloging-in-Publication Data

A catalog record of this book is available from the Library of Congress.

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

All characters in this publication are fictitious, and any resemblance to real persons, living or dead, is purely coincidental.

CONTENTS

PREFACE	xi
1	
WHAT DOES THE CPU DO?	1
Computers Can Process Any Type of Information	11
The CPU Is the Core of Each Computer	14
The Five Components of a Modern Computer	16
ALUs: The CPU's Core	22
CPUs Process Operations and Make Decisions	25
What Is Information Anyway?	30
The Difference Between Analog and Digital Information	31
2	
DIGITAL OPERATIONS	35
The Computer's World Is Binary	36
The Reciprocal States of 1 and 0	37
Decimal vs. Binary Number Systems	38
Expressing Numbers in Binary	40
Fixed-Point and Floating-Point Fractions	42
Addition and Subtraction in Binary	44
What Are Logic Operations?	48
Integrated Circuits Contain Logic Gates	48
The Three Basic Logic Gates: AND, OR, and NOT	51
Truth Tables and Venn Diagrams	53
A Summary of the AND, OR, and NOT Gates	55
Other Basic Gates: NAND, NOR, and XOR	57
A Summary of the NAND, NOR, and XOR Gates	58
De Morgan's Laws	60
Circuits That Perform Arithmetic	62
The Addition Circuit	62
The Half Adder	64
The Full Adder and Ripple Carry Adder	66
The Carry Look-Ahead Adder	68
Circuits That Remember	70
Circuits with Memory Are a Necessity!	70
Flip-Flop: The Basics of Memory Circuits	74
The RS Flip-Flop	76
The D Flip-Flop and the Clock	78
The T Flip-Flop and Counters	81
Modern Circuit Design: CAD and FPGA	85

3	
CPU ARCHITECTURE	87
All About Memory and the CPU	88
Memory Has Assigned Addresses	89
Data Passes Through the Bus	92
Bus Width and Bits	94
R/W Signals and I/O Signals	98
Instructions Are Made of Operands and Opcodes	101
Accumulators and Other Registers Are Used in Operations	103
CPU Instruction Processing	106
Classic CPU Architecture	106
The Instruction Cycle	107
The Instruction We Process Changes Depending on the Program Counter	112
All Kinds of Memory Devices	115
A Comparison Between HDD and Memory	116
RAM Space, ROM Space, and I/O Space	119
What Are Interrupts?	122
Interrupts Are Useful	122
The Stack and the Stack Pointer	126
Interrupt Priority	128
Memory Classifications	132
I/O Ports and the GPU	132
Clock Frequency and Degrees of Accuracy	133
Clock Generators	134
Timer Interrupts	135
Reset Signals	136
CPU Performance Is Measured in FLOPS	137
4	
OPERATIONS	139
Types of Operations	140
There Are Many Types of Instructions	142
Instructions for Arithmetic and Logic Operations	144
What Are Bit Shifts?	145
The Sign Bit Lets Us Express Negative Binary Numbers	147
Logical Shifts and Arithmetic Shifts	149
Circular Shifts (Rotating Shifts)	152
Data Transfer Operations	153
Input/Output Instructions	154
Branch Instructions	155
Condition Evaluation and Status Flags	158
Putting Branches and Condition Evaluation Together	161
Operand Types	162
How Many Operands Do We Have?	162
Operands Take Many Forms	165
Immediate Value Processing	166
Address References	167

What Are Addressing Modes?	168
Addressing Mode Overview.	172
The Structure of Operations in the ALU	176
A Look Inside the ALU.	176
Basic Circuit Architecture of the 74S181	178
Serial Transmission and Parallel Transmission	185
An Overview of Some Basic Registers	186
An Overview of Some Basic Status Flags.	187
The SLEEP Instruction	188
5	
PROGRAMS	189
Assembly and High-Level Languages	190
What Are Assembly Languages?	192
The Characteristics of Assembly Languages and High-Level Languages.	194
The Difference Between Programs and Source Code	199
Program Basics.	200
What Can You Make Using Conditions and Jumps?	200
What Should We Make the Computer Do?	204
Where are Programs Stored?	208
What Happens Before a Program Is Executed?	208
6	
MICROCONTROLLERS	211
What Are Microcontrollers?	212
Microcontrollers Are in All Kinds of Products	213
The Function of a Microcontroller.	214
Architecture of a Microcontroller	219
What Are DSPs?	222
DSPs and Multiplier-Accumulate Operations.	224
Microcontrollers in Industrial Machines	224
EPILOGUE	227
AFTERWORD	239
INDEX	241

PREFACE

Ever since the 1950s, when computers saw their debut in markets all over the world, interest in information technology (IT) has seen a steady rise. The core that supports this technology is a semiconductor known as the CPU, or central processing unit. Since the start of the 21st century, advancements in circuit design theory and manufacturing technology have led to rapid progress in both processing speed and chip size, allowing us to embed them in most of the electronics we use on a daily basis. In addition to personal computers, smartphones, and tablets, you'll even find CPUs in things like air conditioners, refrigerators, washing machines, and other major appliances, just to name a few.

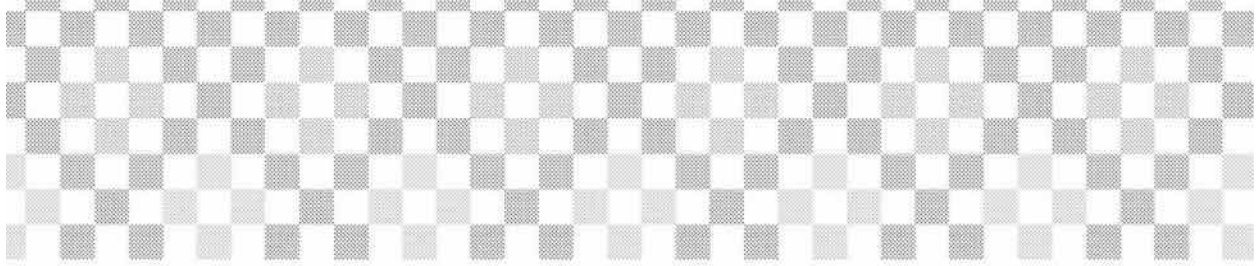
It's worth noting that the CPUs found in modern PCs are extremely powerful, and many of their applications are outside the scope of this book. We also will not delve into computer architecture which has had a research boom in recent years. Instead, I think that the best way to give insight into what CPUs are—and, by extension, how programs work—is to go back and analyze how the first CPUs worked and examine the concepts and principles by which they were designed.

Let me share an allegory with you. It's been quite some time since we first started taking automobiles for granted in our daily lives, but despite their ubiquity, very few people today can explain how an engine works or how the energy generated by the engine gets translated into forward momentum for the car. In the 1950s, you had to answer engine design questions on your driver's license examination, but no such questions remain in today's tests. Essentially, this means that to learn things about the internals of a car engine today, you really have to be an incredibly curious person.

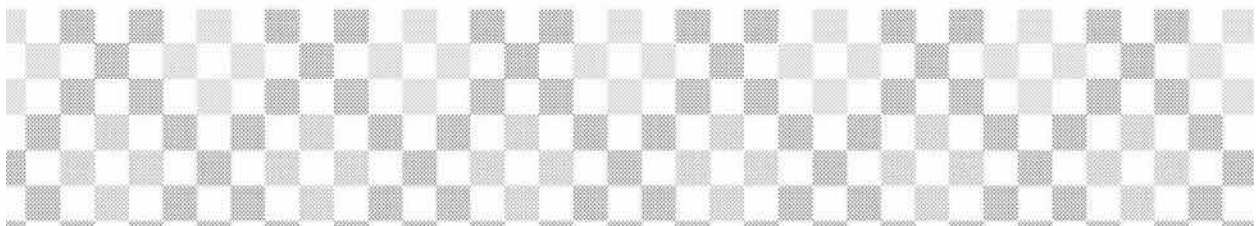
In that vein, my wish is that this book will not only act as a platform to teach readers a range of different topics but also to sate their curiosity on the subject by having them learn some of the deeper principles of the CPUs that have so nonchalantly permeated our daily lives.

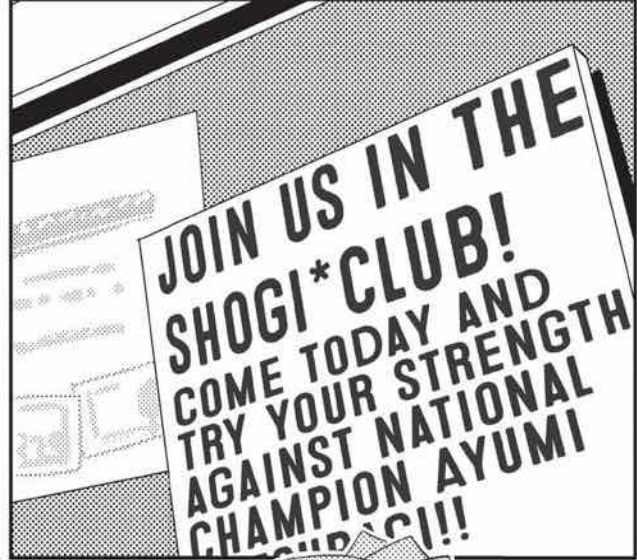
In regards to the publication of this book, I would like to thank Sawako Sawada of Office sawa, who thought up the fun story, and Takashi Tonagi for the illustration work.

MICHIO SHIBUYA
NOVEMBER 2014



WHAT DOES THE CPU DO?





* A JAPANESE BOARD GAME SIMILAR TO CHESS











ER, YOU WANT ME, A REAL SHOGI PLAYER, TO PLAY YOUR VIDEO GAME?



HEH... IT'S NOT JUST A COMPUTER GAME.

I'M A BUSY PERSON, YOU KNOW...



THIS LAPTOP, THE SHOOTING STAR, IS RUNNING A PROGRAM OF MY OWN DESIGN.

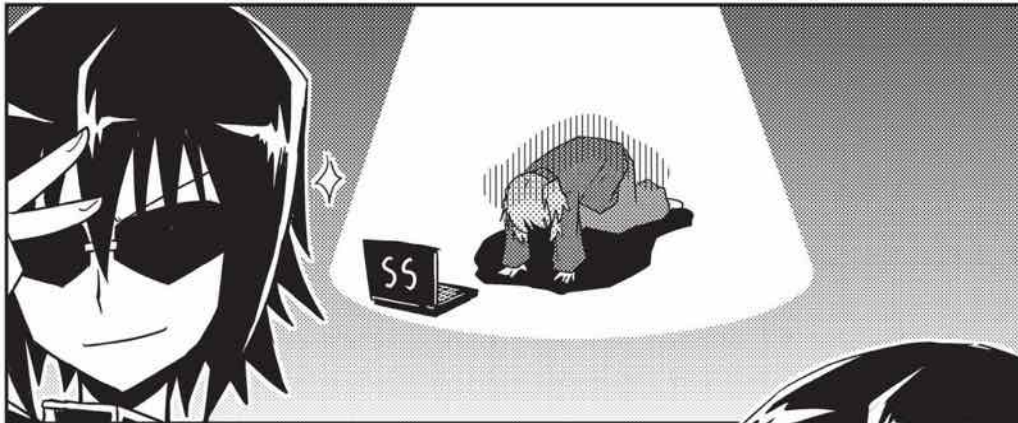
AND... IT'S STRONGER THAN YOU!



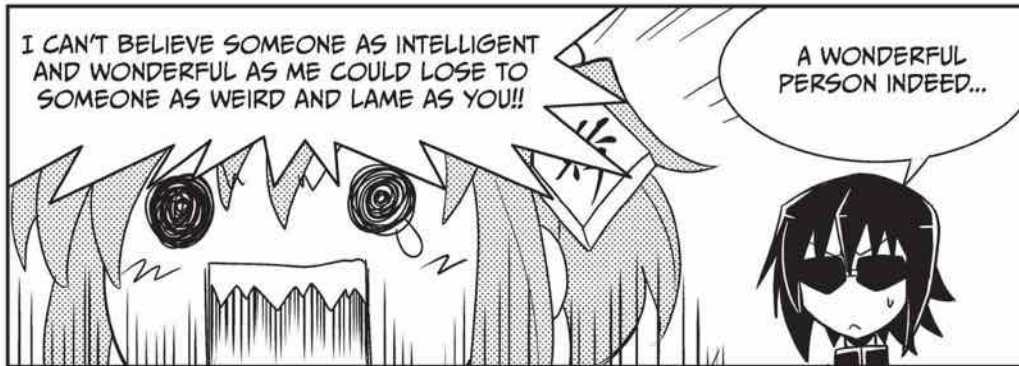
WELL... I DON'T REALLY GET WHAT YOU'RE SAYING, BUT IT'S OBVIOUS YOU'RE LOOKING DOWN ON ME.

I JUST HAVE TO WIN, RIGHT? I HOPE YOU'RE READY TO BE DESTROYED...

I LOST...?
HOW IS THAT POSSIBLE??



*BE SWALLOWED IN
ITS DARK DEPTHS AND
TASTE UTTER DEFEAT!
FUHAHAHAHAHAHAHAHAHA!*



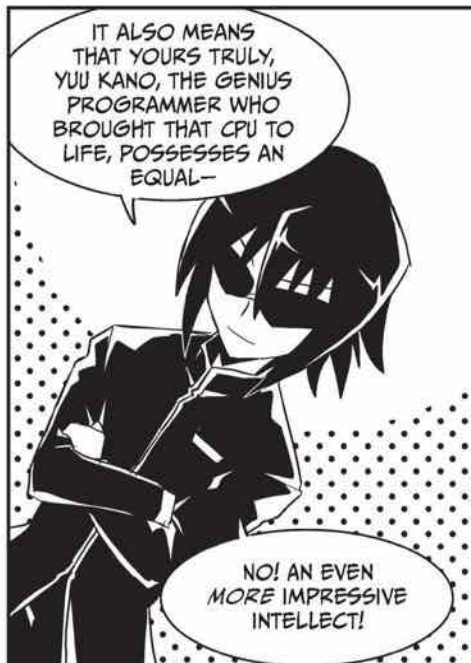


AYUMI KATSURAGI!
KNOW WHEN YOU
ARE BEATEN!

IT'S TRUE THAT IT
WASN'T ME WHO
BEAT YOU.

BUT THAT ALSO MEANS
YOUR INTELLECT WAS
NO MATCH FOR THE CPU,
THE SHOOTING STAR'S
INTELLECT!

AND THAT'S A FACT!



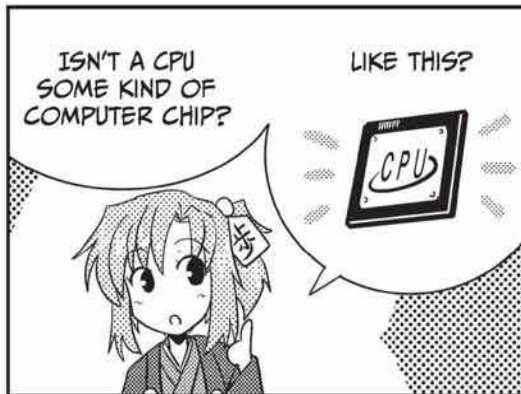
IT ALSO MEANS
THAT YOURS TRULY,
YUU KANO, THE GENIUS
PROGRAMMER WHO
BROUGHT THAT CPU TO
LIFE, POSSESSES AN
EQUAL—

NO! AN EVEN
MORE IMPRESSIVE
INTELLECT!



HUH?

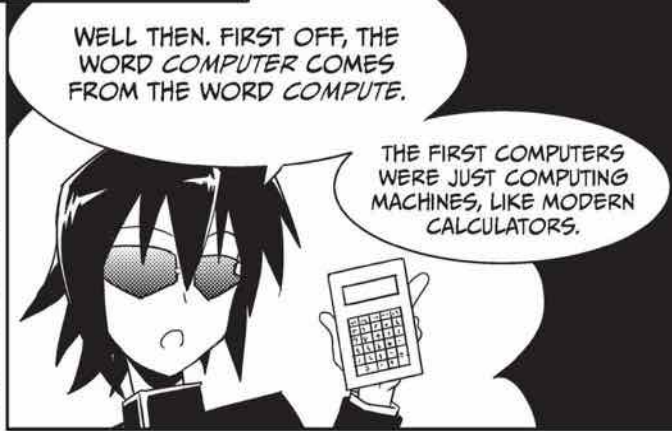
CPU?



COMPUTERS CAN PROCESS ANY TYPE OF INFORMATION

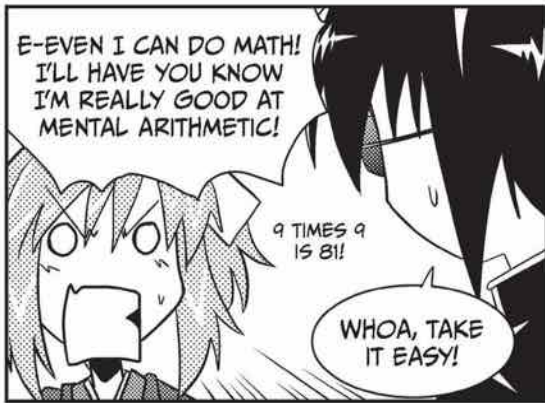


LET'S TAKE OUR TIME AND START FROM THE BEGINNING.



WELL THEN. FIRST OFF, THE WORD COMPUTER COMES FROM THE WORD COMPUTE.

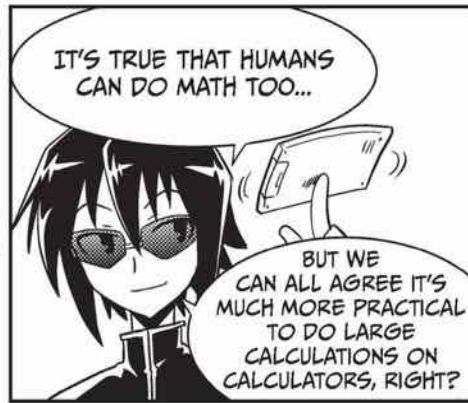
THE FIRST COMPUTERS WERE JUST COMPUTING MACHINES, LIKE MODERN CALCULATORS.



E-EVEN I CAN DO MATH! I'LL HAVE YOU KNOW I'M REALLY GOOD AT MENTAL ARITHMETIC!

9 TIMES 9 IS 81!

WHOA, TAKE IT EASY!



IT'S TRUE THAT HUMANS CAN DO MATH TOO...

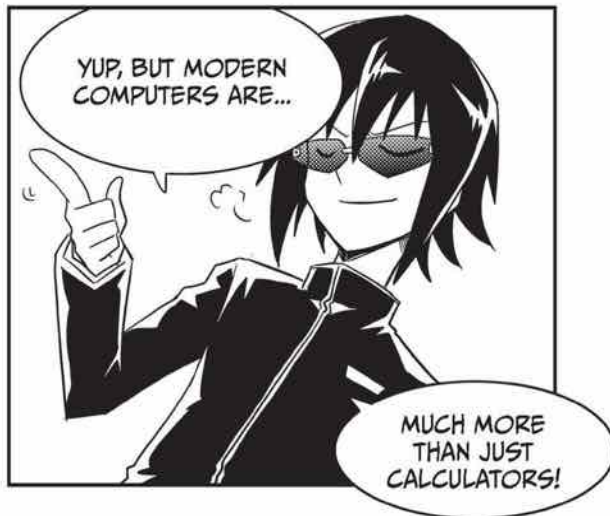
BUT WE CAN ALL AGREE IT'S MUCH MORE PRACTICAL TO DO LARGE CALCULATIONS ON CALCULATORS, RIGHT?



NGH... YEAH... SO COMPUTERS...

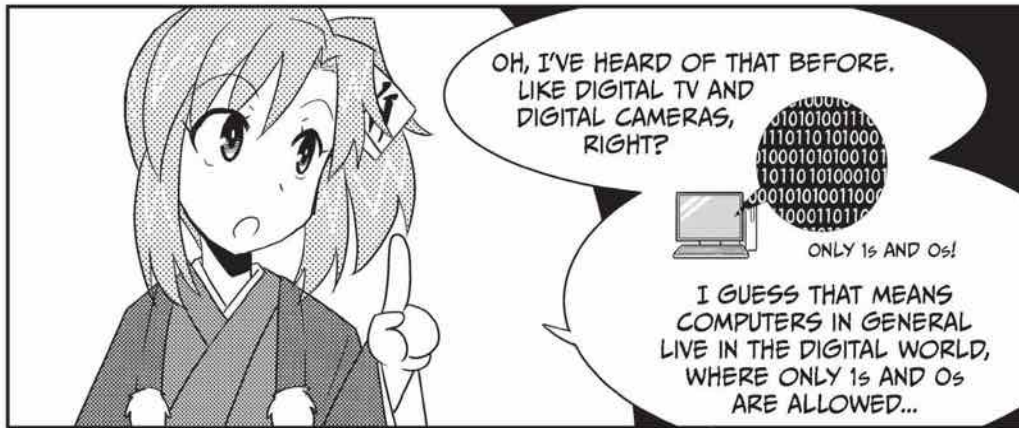
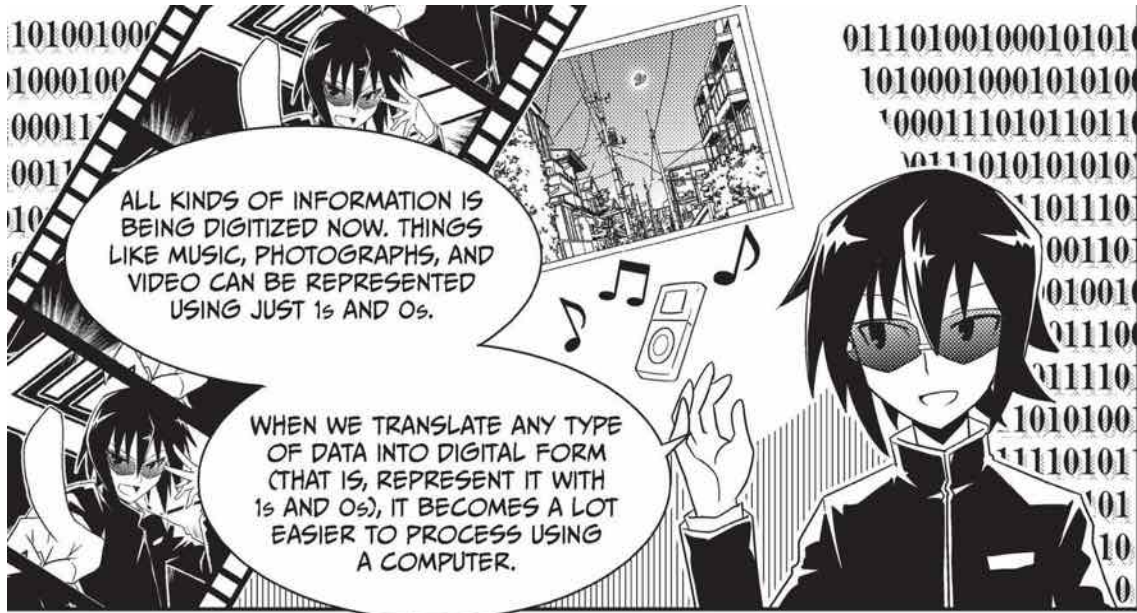
HMM...

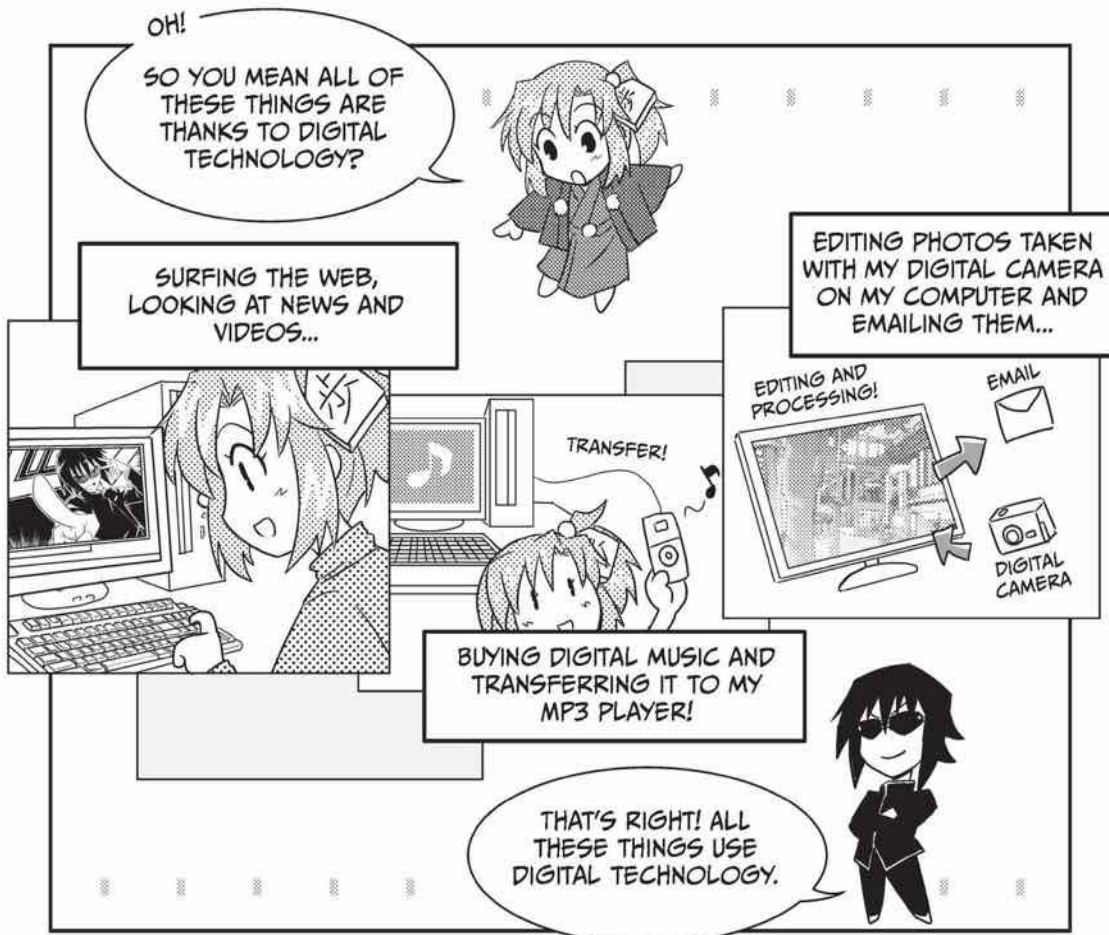
ARE REALLY GOOD AT DOING CALCULATIONS REALLY FAST.



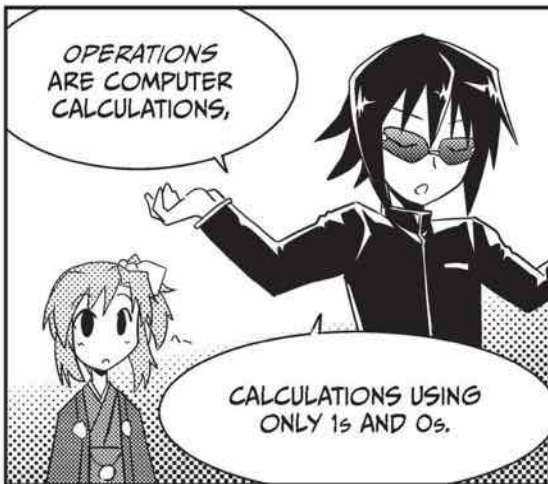
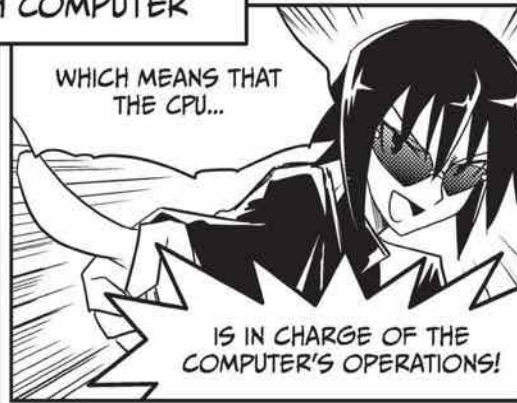
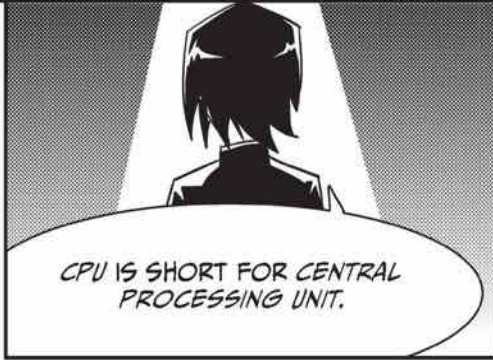
YUP, BUT MODERN COMPUTERS ARE...

MUCH MORE THAN JUST CALCULATORS!

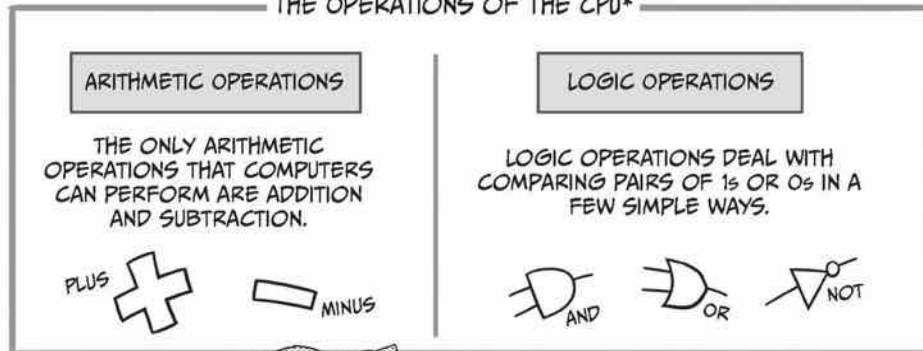




THE CPU IS THE CORE OF EACH COMPUTER



THE OPERATIONS OF THE CPU*



* IN ADDITION TO THESE OPERATIONAL UNITS, MODERN CPUs ALSO CONTAIN FPUs (FLOATING POINT UNITS) THAT CAN HANDLE MULTIPLICATION AND DIVISION. BUT THIS BOOK JUST STICKS TO THE BASICS.





THE FIVE COMPONENTS OF A MODERN COMPUTER

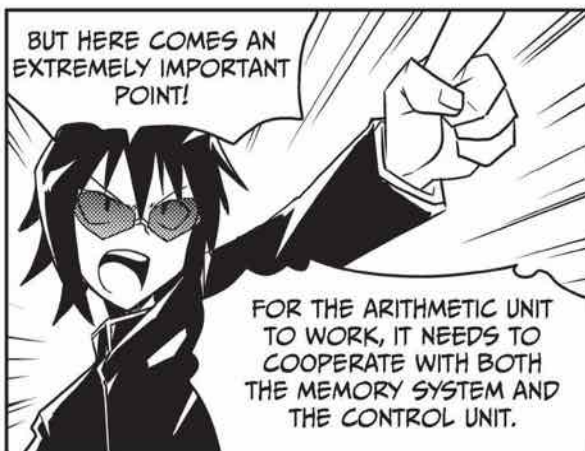
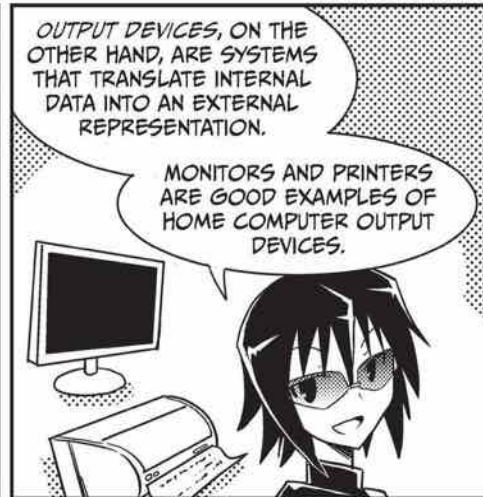
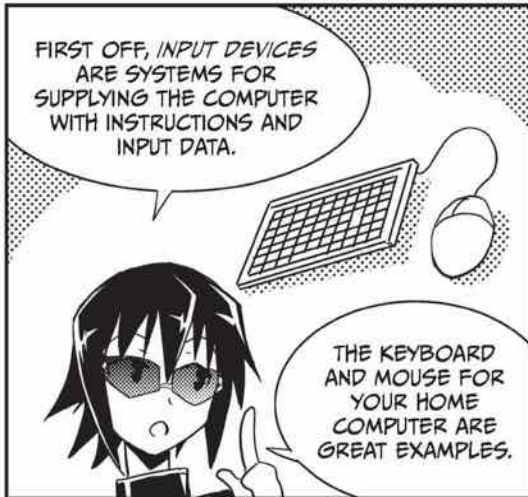
HERE'S A QUICK OVERVIEW OF THE MOST IMPORTANT PARTS OF A COMPUTER.

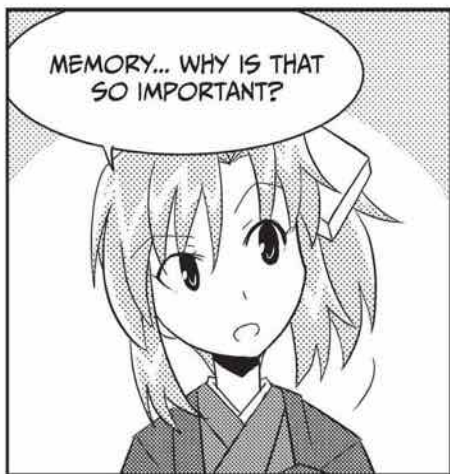
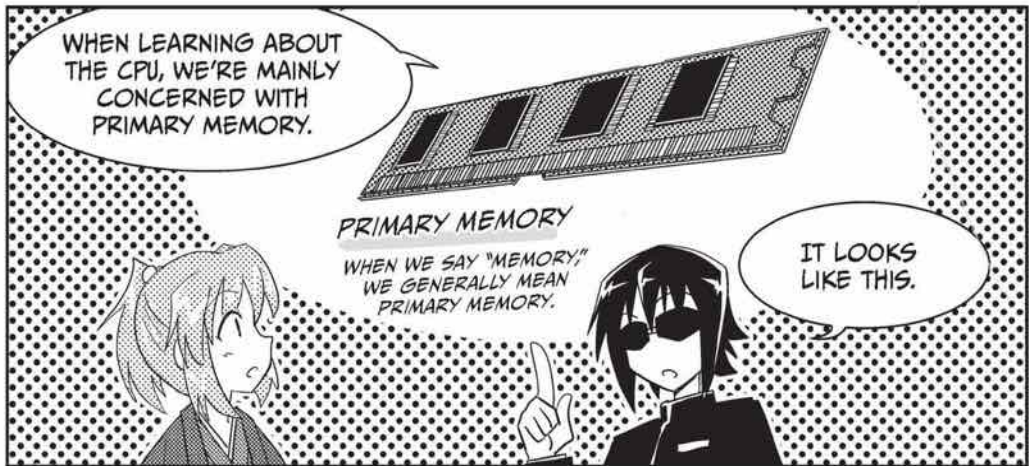
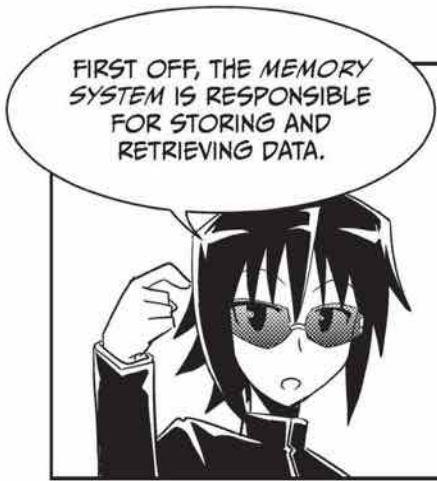
LET'S CALL THESE THE COMPUTER'S FIVE PRIMARY SYSTEMS. AS YOU CAN SEE, THE CPU CONTAINS BOTH THE CONTROL UNIT AND THE ARITHMETIC UNIT.

THE COMPUTER'S FIVE PRIMARY SYSTEMS

DATA IS EXCHANGED THROUGH THE INPUT/OUTPUT SYSTEM.







YES, BOTH THE DATA THAT'S OPERATED ON AND THE COMPUTER PROGRAM ARE STORED IN MEMORY. OPERATIONS USE THESE AS INPUT.

DEPENDING ON THE OPERATION, DATA MAY BE RETRIEVED FROM MEMORY FOR INPUT, OR THE RESULT OF THE OPERATION MAY BE RETURNED BACK INTO MEMORY FOR STORAGE.*

CONTROL UNIT

ARITHMETIC UNIT

CPU

INPUT DATA

PROGRAM (INSTRUCTIONS)

OUTPUT DATA

PRIMARY MEMORY

* THE CPU MAY USE EITHER REGISTERS OR CACHE MEMORY.

RETRIEVING AND RETURNING... THE CPU REALLY EXCHANGES INFORMATION!

BY THE WAY, I'VE HEARD THAT WORD PROGRAM BEFORE, BUT WHAT IS IT?

TO PUT IT SIMPLY...

PROGRAMS ARE INSTRUCTIONS THAT PEOPLE GIVE THE COMPUTER.

INSTRUCTIONS ABOUT WHAT DATA TO USE, AND WHICH OPERATIONS TO RUN AND IN WHAT ORDER.

PLEASE DO IT LIKE THIS.

HUMAN

PROGRAM (INSTRUCTIONS)

COMPUTER

OKAY!

I SEE. PROGRAMS ARE DIRECTIONS THAT TELL THE COMPUTER WHAT TO DO.

ALL THOSE INSTRUCTIONS ARE WRITTEN IN PROGRAMS.

CONTROLS EVERYTHING

CONTROL UNIT

INPUT MEMORY ARITHMETIC UNIT OUTPUT

INSTRUCTION

THERE'S ONE SYSTEM WE HAVEN'T TALKED ABOUT YET!

THE CONTROL UNIT TELLS THE OTHER FOUR SYSTEMS WHAT TO DO! IT GIVES OUT INSTRUCTIONS AND CONTROLS THE COMPUTER.

THAT SEEMS PRETTY ARROGANT! SO IT'S LIKE SOME KIND OF OVERSEER THAT GIVES OUT ORDERS LEFT AND RIGHT?

YEAH, THAT'S RIGHT. ALSO, LIKE I SAID BEFORE, PROGRAMS ARE STORED IN MEMORY.

THE CONTROL UNIT READS THE PROGRAM INSTRUCTIONS FROM MEMORY AND INTERPRETS THEM.

PROGRAM READ FROM MEMORY

INSTRUCTION

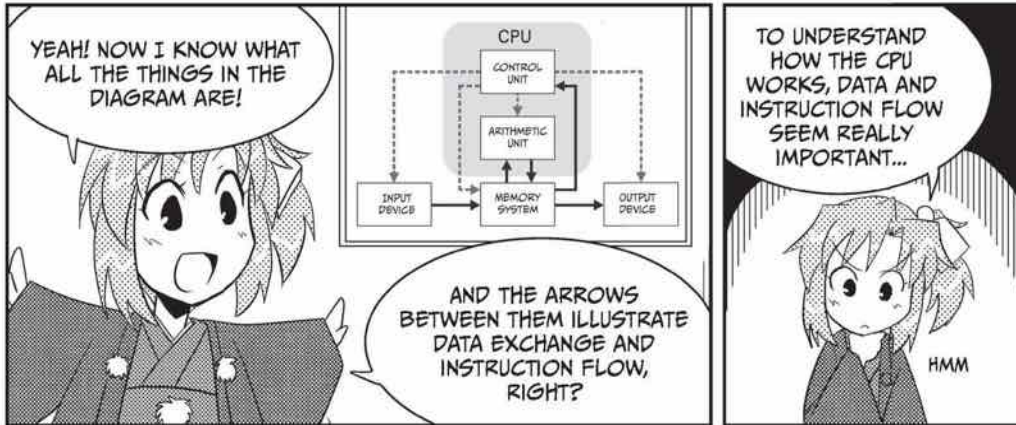
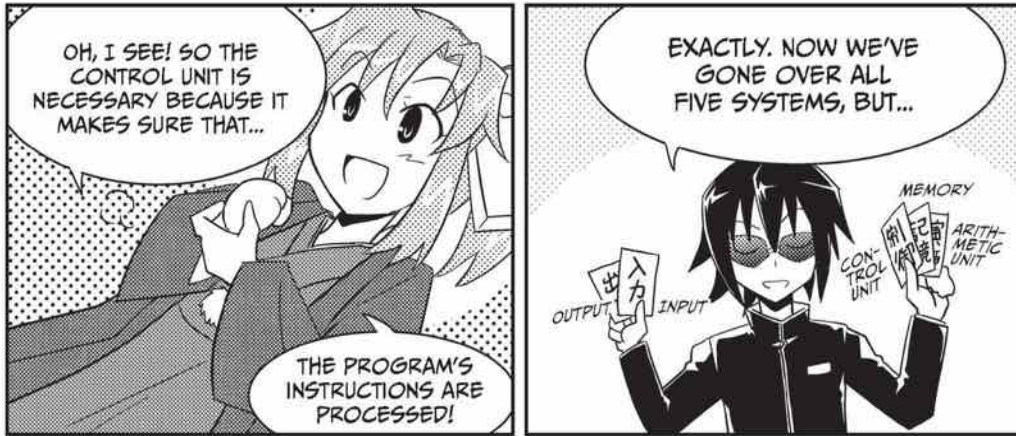
CONTROL UNIT

GET DATA FROM OVER THERE!

THEN ADD THESE TWO!

AND THEN SAVE THAT SUM OVER THERE!

IT GIVES ORDERS TO ALL THE OTHER SYSTEMS, TELLING THEM HOW TO PROCESS THE PROGRAM'S INSTRUCTIONS.



ALUs: THE CPU'S CORE

YOU'RE CATCHING ON PRETTY QUICKLY, IT SEEMS.

SO LET'S TALK A BIT ABOUT ALUs.

ALUs? NOT CPUs? WHAT'S THE DIFFERENCE?

WELL, ALUs ARE WHAT PERFORM OPERATIONS INSIDE THE CPU.

ALUs ARE THE ARITHMETIC UNIT'S PRINCIPAL COMPONENTS.

CONTROL UNIT

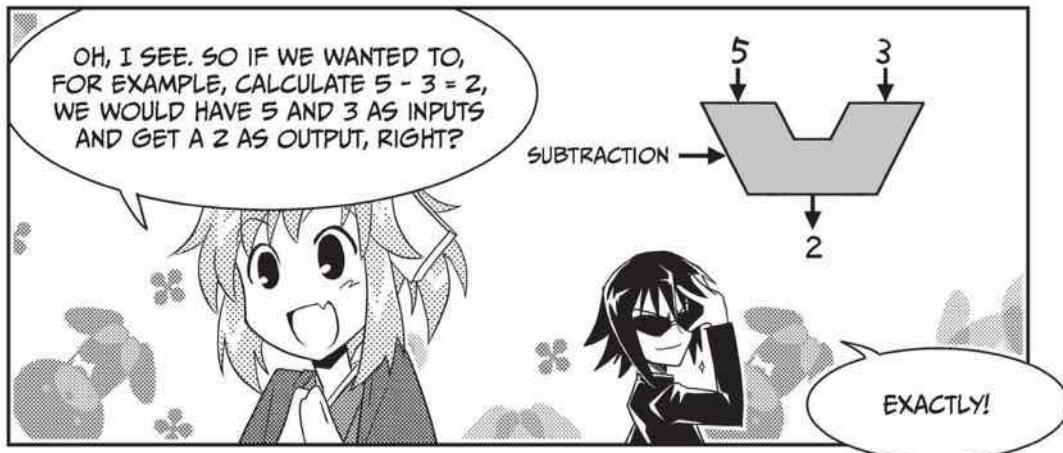
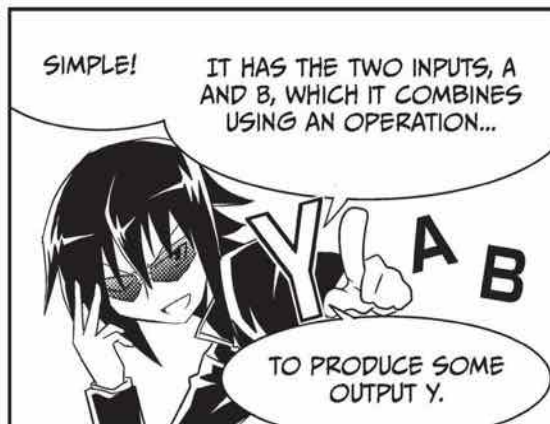
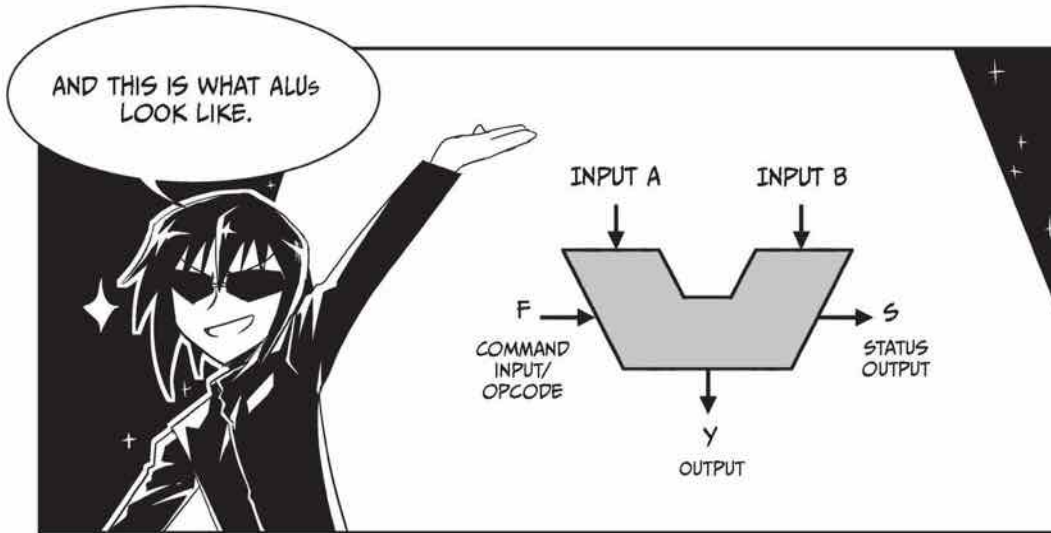
CPU

ARITHMETIC UNIT (ALU)

OH! THAT SEEMS LIKE IT'S SUPER IMPORTANT!

YES, ALU IS SHORT FOR ARITHMETIC LOGIC UNIT.

IT PERFORMS THE ARITHMETIC AND LOGIC OPERATIONS WE TALKED ABOUT BEFORE.



SO THE COMMAND INPUT F IS WHAT WE WANT THE ALU TO DO...

LIKE ADDITION OR SUBTRACTION, FOR EXAMPLE.

AND THE STATUS OUTPUT S TELLS US HOW THE OPERATION WENT.

THIS COULD BE INFORMATION ON WHETHER OUR OUTPUT VALUE WAS POSITIVE OR NEGATIVE, FOR EXAMPLE.

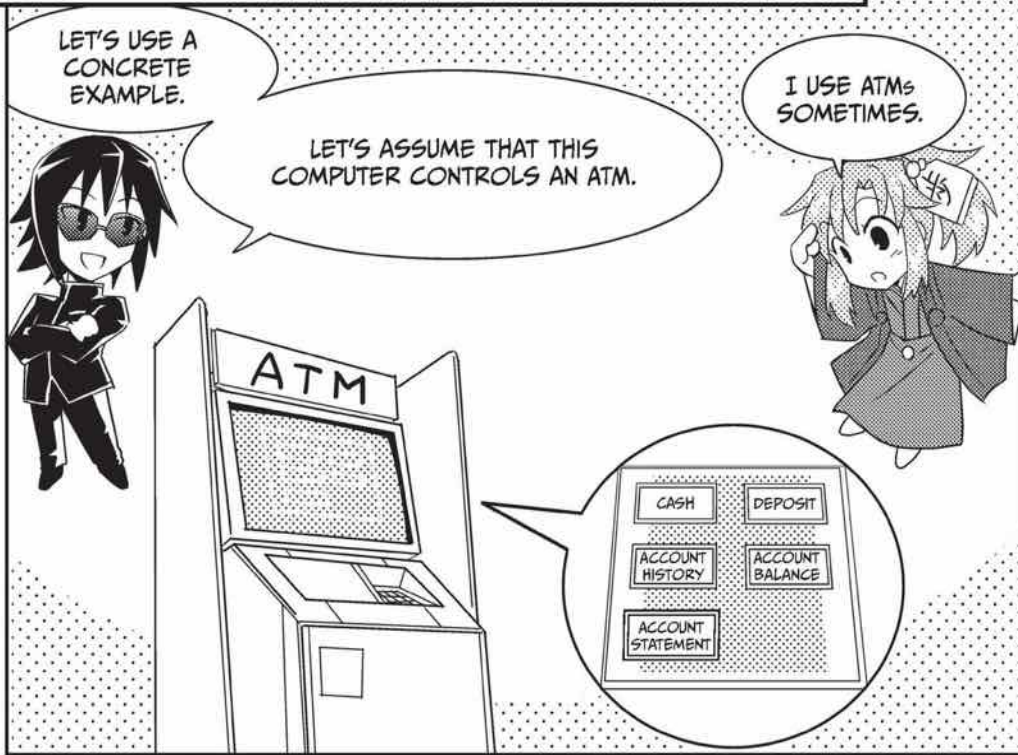
SO IN THE CASE OF $5 - 3 = 2$, SINCE THE RESULT OF THE CALCULATION IS 2, THE STATUS OUTPUT SHOULD SAY IT'S A POSITIVE VALUE, RIGHT?

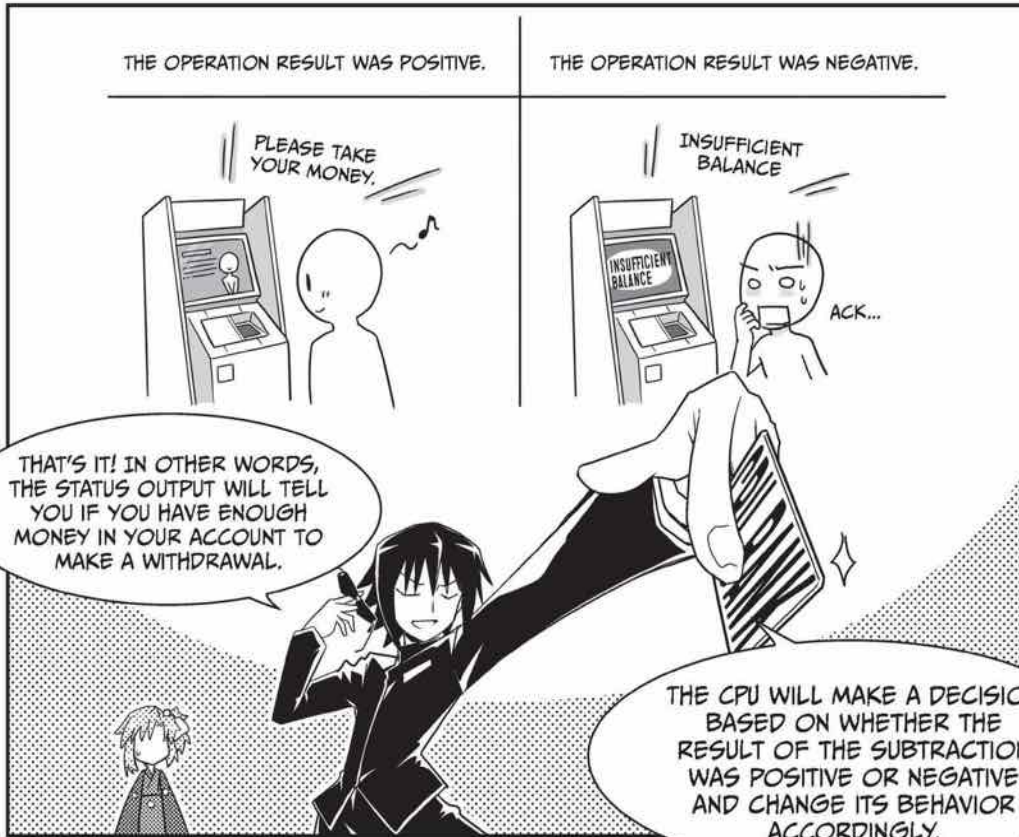
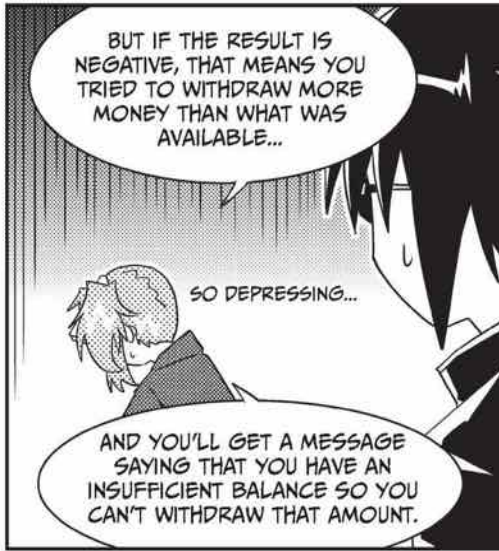
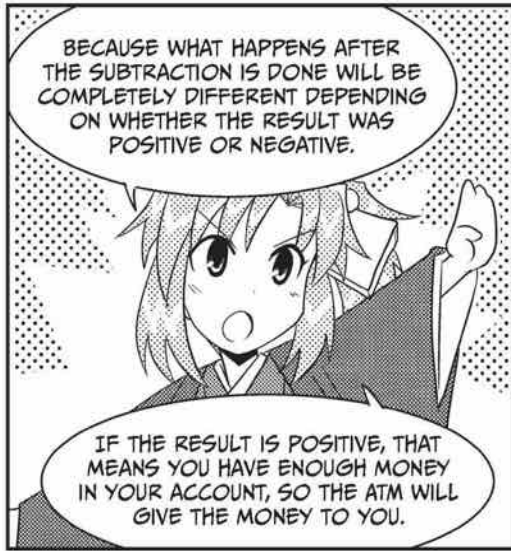
BUT WHY DOES ANYONE NEED TO KNOW WHETHER THE OUTPUT WAS POSITIVE OR NEGATIVE?

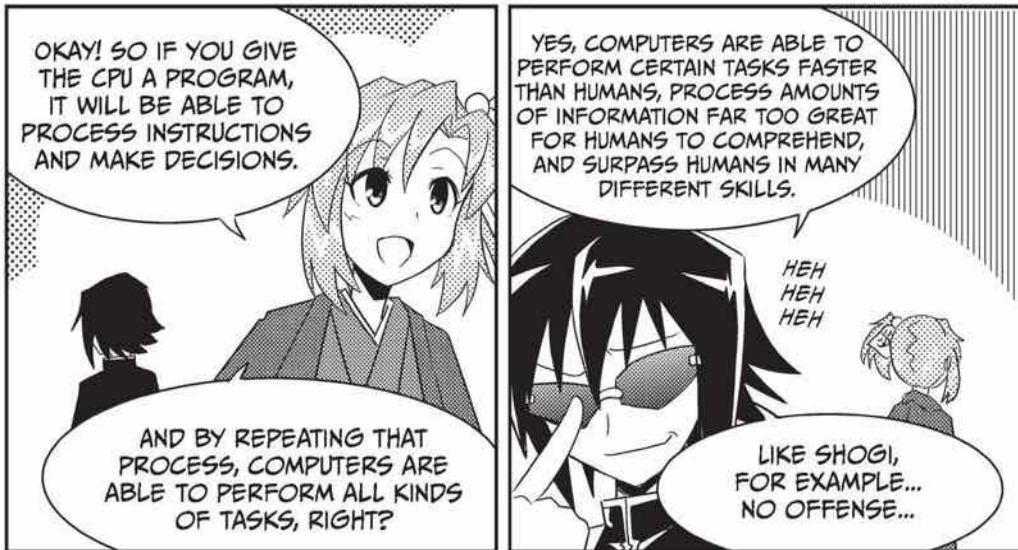
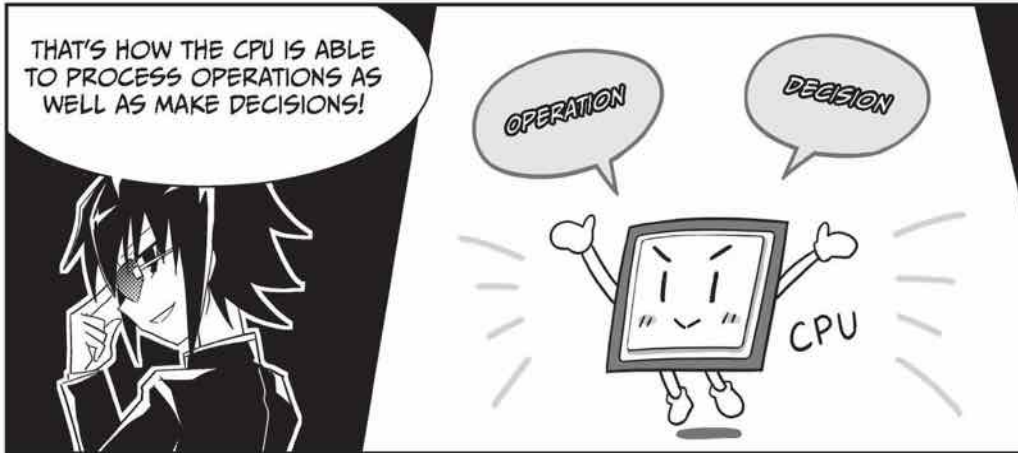
THAT'S A GOOD QUESTION. AS A MATTER OF FACT, THE STATUS OUTPUT CAN BE USED TO MAKE DECISIONS BASED ON WHETHER THE OUTPUT SATISFIES SOME GIVEN CONDITION.

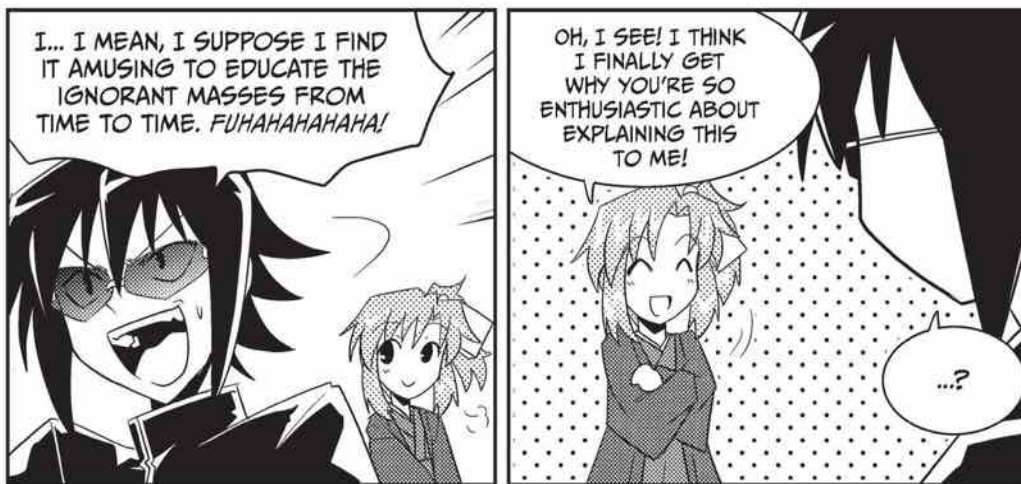
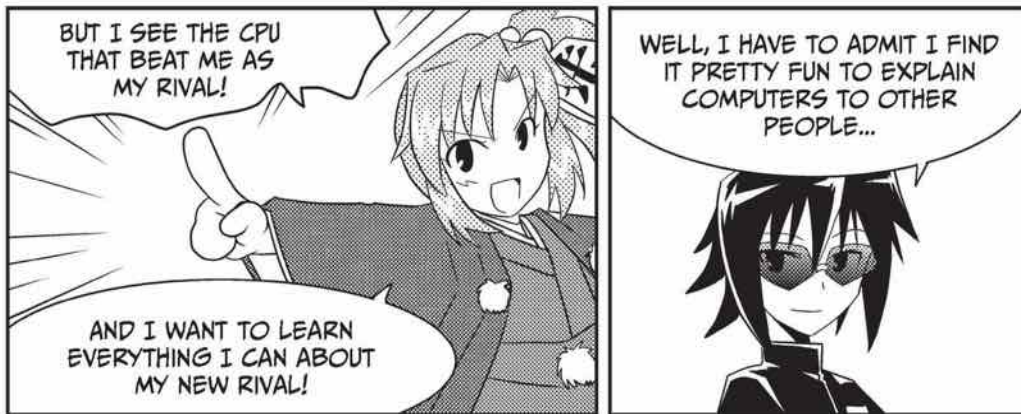
DECISIONS? CONDITIONS?

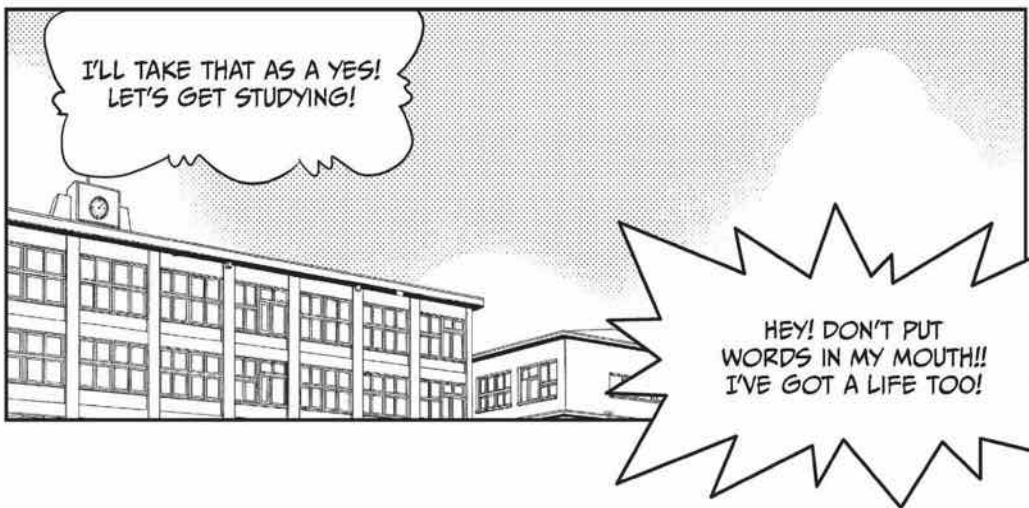
CPU'S PROCESS OPERATIONS AND MAKE DECISIONS









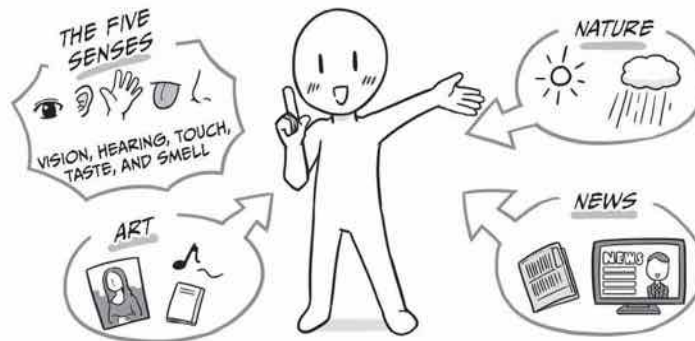


WHAT IS INFORMATION ANYWAY?

Information technology (IT) became an everyday phrase back in the 20th century. The term is frequently heard when people talk about the internet and other computer technology, but it's worth noting that this term predates the use of computers.

First off, what does the word *information* actually mean? To put it simply, information is everything in our environment that can be registered with any of our five senses.

EVERYTHING THAT I CAN PERCEIVE IS INFORMATION!

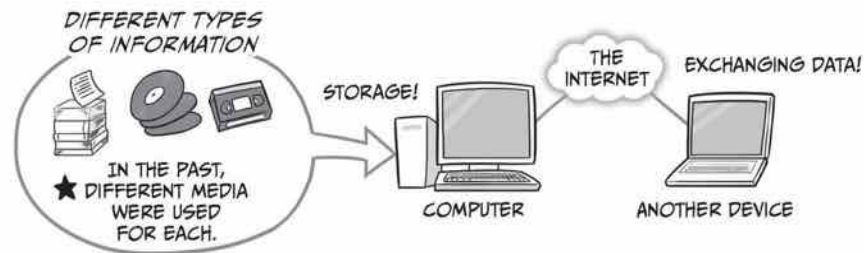


Everything that occurs in nature or in paintings, photographs, music, novels, news, radio, TV broadcasts, and so on is an example of information. Most of these things have been around for a lot longer than our society has had access to electricity. As information spreads throughout society, it affects our lives.

Every day, people and organizations value useful information while trying to filter out everything else. Information that is not important is called *noise*, and important information is called *signal*. Finding ways to maximize the *signal-to-noise ratio*—that is, the amount of signal in an output compared to the amount of noise—without accidentally losing necessary information is important.

One type of information that historically has been important both to people and organizations is information about food—what's safe or healthy to eat, how to find or grow it, and how far away it is or how much it costs to buy. Related information, such as climate and weather forecasts, is also vital. Obviously, information like this was valued long before the rise of the internet. For example, merchants like Bunzaemon Kinokuniya from Japan's Edo period specialized in products such as citrus and salmon and thrived because they valued this type of information. Indeed, the value of information has been respected for as long as people have needed to eat.

However, the digital age has affected many aspects of life. How has it affected our access to information? Well, thanks to the digitization of data, we are now able to process diverse data like text, audio, images, and video using the same methods. It can all be transmitted the same way (over the internet, for example) and stored in the same media (on hard drives, for example).



Computers that are connected to the same network can exchange digitized information. By using computers to match and analyze large sets of data instead of analyzing each instance or type of data individually, people can discover otherwise hidden trends or implications of the information.

Like the storage of data, information transmission has made incredible advances, thanks to important discoveries in electronics and electrical engineering. Commercial applications of this technology in devices such as telephones, radio, and television have played a role in accelerating this development. Today, almost all of Japan enjoys digital television, which uses digital transmission and compression technologies. CPUs play a central part in these applications by performing countless operations and coordinating the transfer of information.

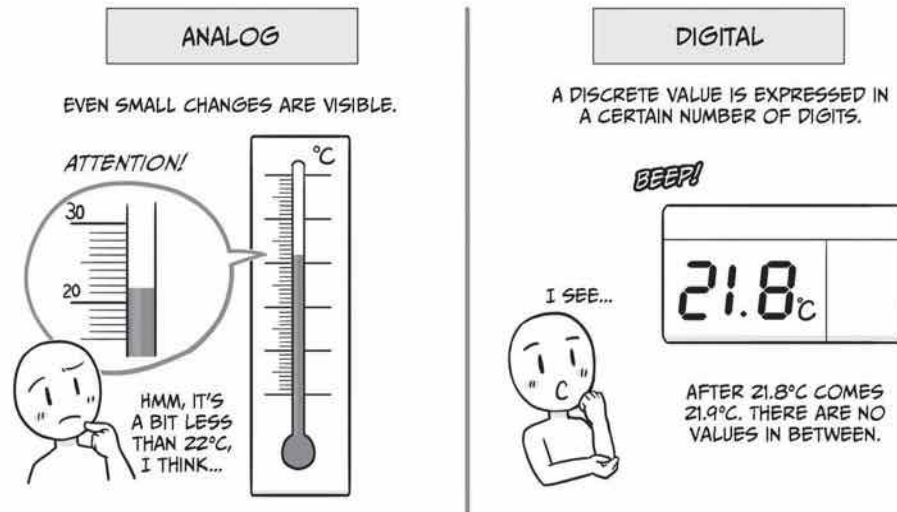
THE DIFFERENCE BETWEEN ANALOG AND DIGITAL INFORMATION

We have been talking about digitizing data into 1s and 0s so that information can be processed by a CPU. But before they are digitized, text, audio, video, and so on exist as analog data.

What is the difference between these two types of data? An example that illustrates the difference is thermometers. Analog thermometers contain a liquid that expands as it heats up, such as mercury or alcohol, in a graduated capillary tube that is marked with lines indicating the temperature. To determine the temperature, we look at the level of the liquid in the tube and compare it to the markings on the tube. We say that the analog thermometer has a *continuous* output because the temperature reading can fall anywhere between the marks on the tube.

Digital thermometers use a sensor to convert temperature into voltage* and then estimate the corresponding temperature. Because the temperature is represented numerically, the temperature changes in steps (that is, the values "jump"). For instance, if the initial temperature reading is 21.8 degrees Celsius and then the temperature increases, the next possible reading is 21.9 degrees Celsius. Because 0.1 is the smallest quantity that can be shown by this thermometer, changes in temperature can only be represented in steps of 0.1 and the value could never be between 21.8 and 21.9 degrees. Thus, digital output is said to be *discrete*.

* Voltage is a way of measuring electric currents and is expressed in volts.



The word *digital* comes from the act of counting off numbers using our fingers—or digits. This tends to lead people to believe that digital computers can only work with data comprised of integers (whole numbers), which is not necessarily true.

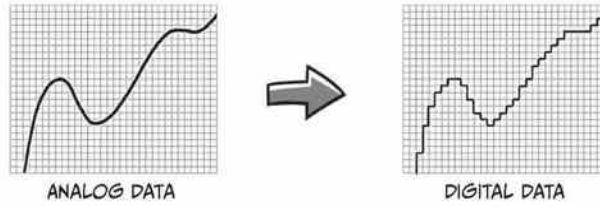
In the digital world, everything is expressed in 1s and 0s. Indeed, they are not even what the CPU works with. Note that these are not actually numbers in this context. Instead, a 1 and a 0 are merely symbols. The CPU consists of transistors that transmit or inhibit electrical signals and consequently output either low or high voltages. It is these voltages that we represent as 1 or 0. A high voltage is represented with a 1, since the transistor's state is "on," and a low voltage, or an "off" transistor, is represented with a 0. In text, you could illustrate this by using the symbols ● and ○. The 1s and 0s are called *primitives*, meaning they are basic data types. Computers can work with decimal numbers as long as the value has a finite number of digits. Values such as these are also digital. The important thing to remember is that for any digital number, you can never add or remove a quantity smaller than the smallest possible value expressible.

Let's compare some analog data and its digitized version to better understand how they are alike and how they differ by looking at the figure on the next page. The first pair of images shows audio data, and the second pair shows image data.

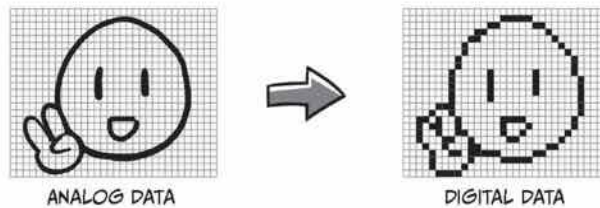
As you can see, every time we translate analog data into digital data, some information is lost. But as you've undoubtedly experienced, most modern digitization processes are so good that humans can't tell the difference between the original and the digital copy, even when they are presented side by side.

To store and transmit digital data of a quality such that our senses can't detect any loss of information, we use special compression techniques. These techniques always involve trade-offs among how much space is used, how much information is lost during compression, and how much processing time is needed to compress and decompress the data.

AUDIO WAVEFORM



GRAPHIC OR VIDEO

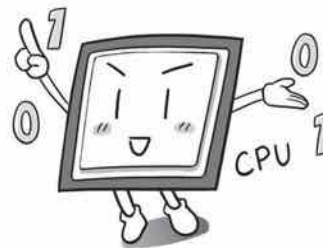


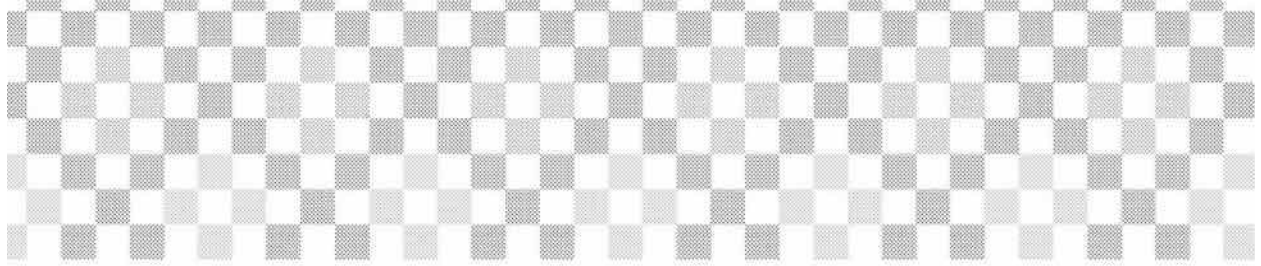
When color information is translated into digital form, it is split into three base component colors, most often red, green, and blue (known as RGB). These colors are combined to create a composite color on a screen. Each component color can be represented by a number, with larger numbers indicating there's more of that color.

When heavily compressing audio or video data, we often use *lossy* techniques that change and simplify the data in such a way that we usually do not notice a difference. While this approach saves a lot of space, as the name implies, reconstructing the original data perfectly is impossible since vital information is missing. Other techniques—most notably all text compression techniques—use *lossless* compression, which guarantees that the original data can be completely reconstructed.

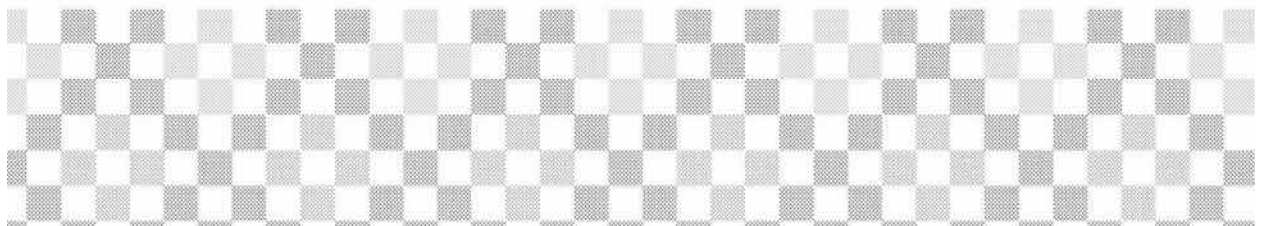
In any case, with the appropriate arithmetic and logic operations, as long as the data is digital, a CPU can use any compression technique on any form of information. Although digitizing data can involve the loss of some information, a major advantage of digital data over analog data is that it allows us to control noise when transmitting the data.

AS LONG AS THE INFORMATION IS
MADE UP OF 1s AND 0s, I'LL KEEP
APPLYING OPERATIONS!

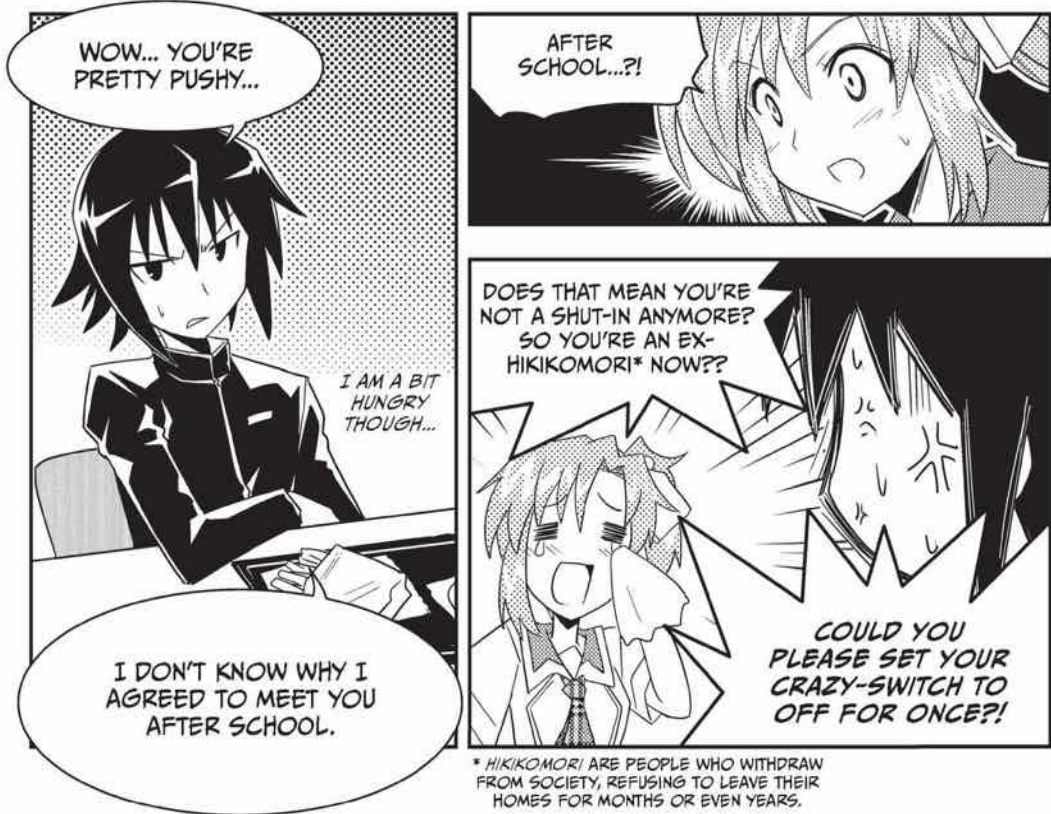




DIGITAL OPERATIONS



The Computer's World Is Binary



THE RECIPROCAL STATES OF 1 AND 0

OKAY, LET ME START OFF WITH A QUESTION!

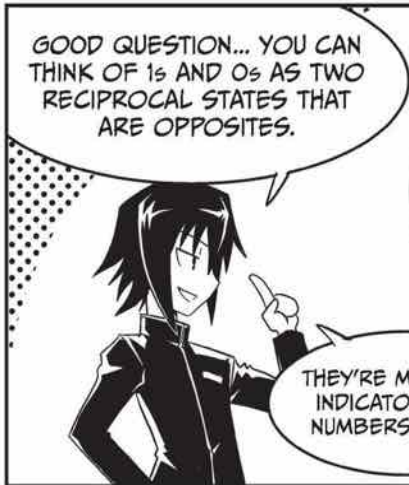


LAST TIME, YOU SAID, "COMPUTERS LIVE IN A WORLD OF 1s AND 0s," BUT THAT WAS ALL PRETTY ABSTRACT.

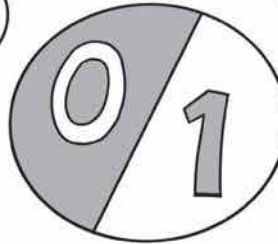


WHAT DO YOU MEAN BY 1s AND 0s ANYWAY?

GOOD QUESTION... YOU CAN THINK OF 1s AND 0s AS TWO RECIPROCAL STATES THAT ARE OPPOSITES.



THEY'RE MORE LIKE INDICATORS THAN NUMBERS, REALLY.

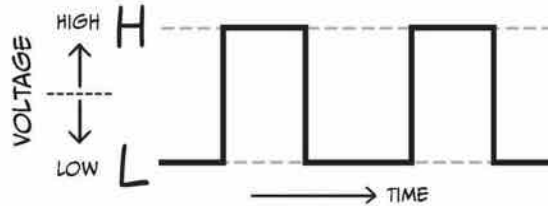


TWO RECIPROCAL STATES...



YOU MEAN LIKE LIGHT AND DARK, LIFE AND DEATH, OR ON AND OFF?

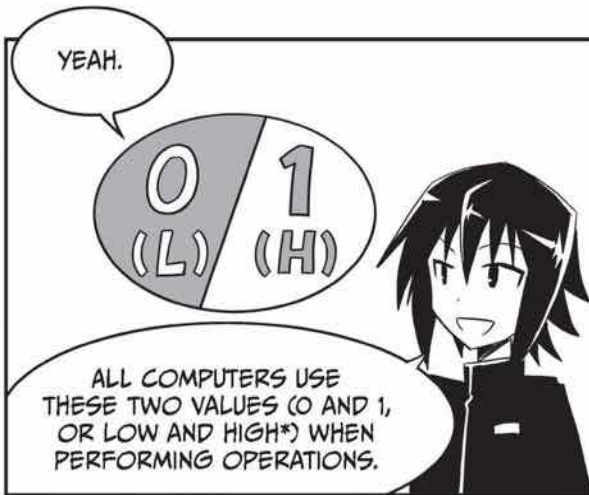
PRECISELY!



VOLTAGE CHANGES WITH TIME

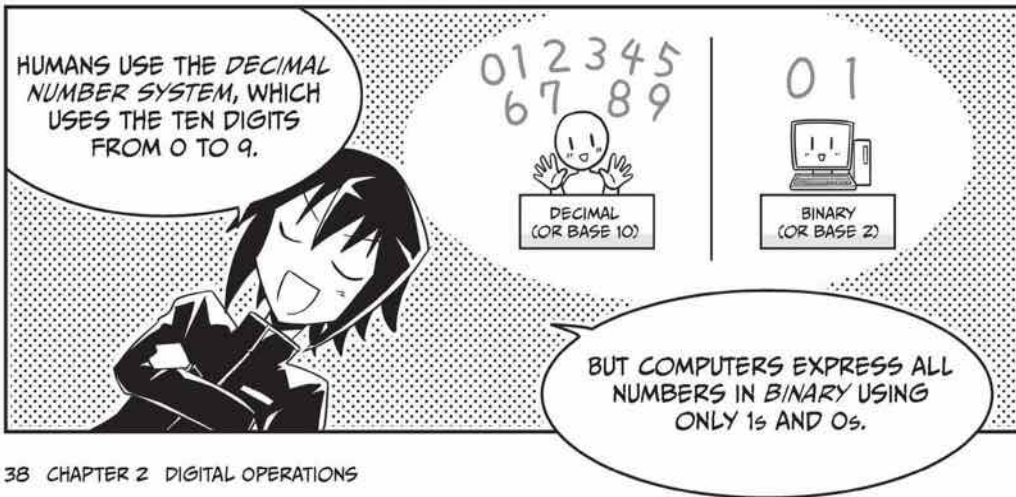
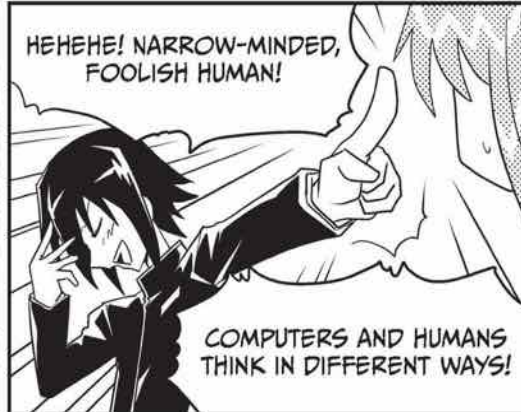
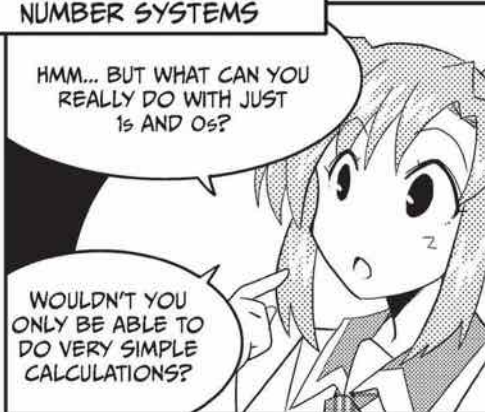
TO PUT IT ANOTHER WAY, THE VOLTAGES IN COMPUTER CIRCUITS GENERALLY FALL INTO TWO BANDS. HIGH VOLTAGES ARE CLOSE TO THE SUPPLY VOLTAGE, AND LOW VOLTAGES ARE CLOSE TO GROUND.*

* GROUND IS THE REFERENCE POINT FOR VOLTAGE AND IS EQUIVALENT TO ZERO VOLTS.



* IN THIS BOOK, WE'LL TREAT LOW AS 0 AND HIGH AS 1, BUT IT'S POSSIBLE TO DO IT THE OTHER WAY AROUND AS WELL. IT'S UP TO THE SYSTEM DESIGNER AS TO WHICH ASSIGNMENT TO USE.

**DECIMAL VS. BINARY
NUMBER SYSTEMS**



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

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
⋮	⋮

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 1 OR A 0) IS ALSO CALLED A BIT IN COMPUTER TERMINOLOGY. THAT'S REALLY IMPORTANT, SO DON'T FORGET IT!

COMPARING DECIMAL AND BINARY

1 bit
1001

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. For example, the number 356 is divided up, and then each digit is multiplied by successive powers of ten to get the final value.

$$\begin{array}{c} \boxed{356} \\ \swarrow \quad \downarrow \quad \searrow \\ \underbrace{3 \times 10^2}_{\text{HUNDREDS}} + \underbrace{5 \times 10^1}_{\text{TENS}} + \underbrace{6 \times 10^0}_{\text{ONES}^*} \end{array}$$

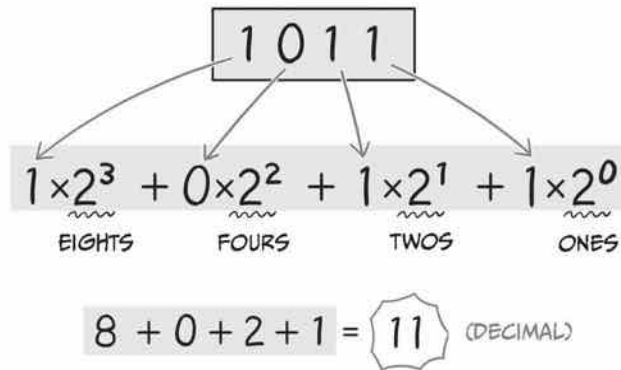


Okay! It's really easy if I think of the digits like different coin denominations: 356 yen is just three 100-yen coins (10^2), five 10-yen coins (10^1), and six 1-yen coins (10^0) added together.



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 the following illustration.

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

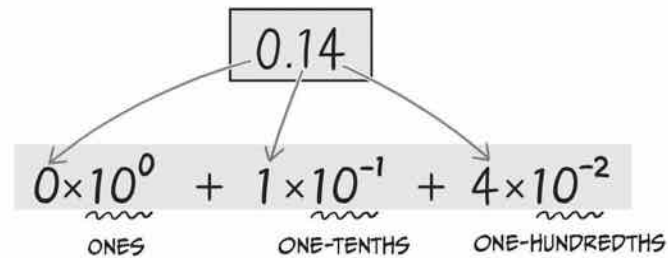


Uh-huh! I don't think anyone uses coins like this, though. But if someone did, I would just take either 1 or 0 of each of the 8-yen, 4-yen, 2-yen, and 1-yen coins, right?

So the binary 1011 translates to $8 + 0 + 2 + 1 = 11$ in decimal. As soon as you understand the basic principle, it's easy!



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



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



So, it's the same reasoning with binary, right? We would use 2^{-1} , 2^{-2} , 2^{-3} and so on as we add more digits after the decimal point. 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.



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 or 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.

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

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



I used the decimal 1.69 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.



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



Yes! And it's also important to understand that the speed with which you can calculate using floating-point numbers is critical to CPU performance. Gaming systems that process real-time, high-fidelity graphics also use floating-point arithmetic extensively. (See "CPU Performance Is Measured in FLOPS" on page 137 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!



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!

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1, \quad 1 + 1 = 10$$



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

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

CARRIED TO THE NEXT PLACE VALUE



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 binary 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 it's in 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
$\begin{array}{r} 15 \\ + (-15) \\ \hline 0 \end{array}$	$\begin{array}{r} 15 \\ + (85) \\ \hline 100 \end{array}$
	<p>↑ IGNORE!</p>

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 the equation B result of 100?



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 as "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 by adding instead. I suppose that might be useful. So what happens if we try this far-fetched solution with binary numbers?



It's not far-fetched—it's awesome! It's logical!! 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, the two numbers are complementary if the result equals 0. To subtract 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.



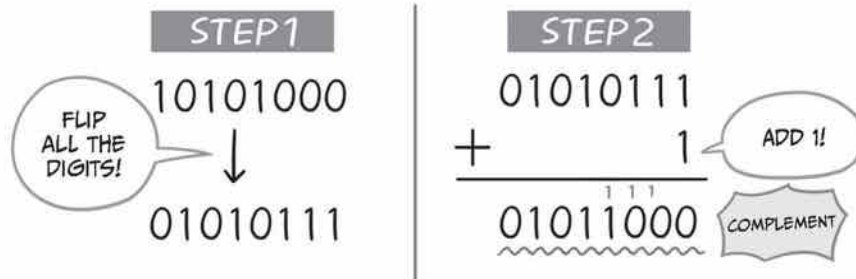
LET'S FIND THE TWO'S COMPLEMENT TO DO SUBTRACTION!

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 these two steps 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, they add 1 to that sum. The order of operations is different, but the end result is the same, right?

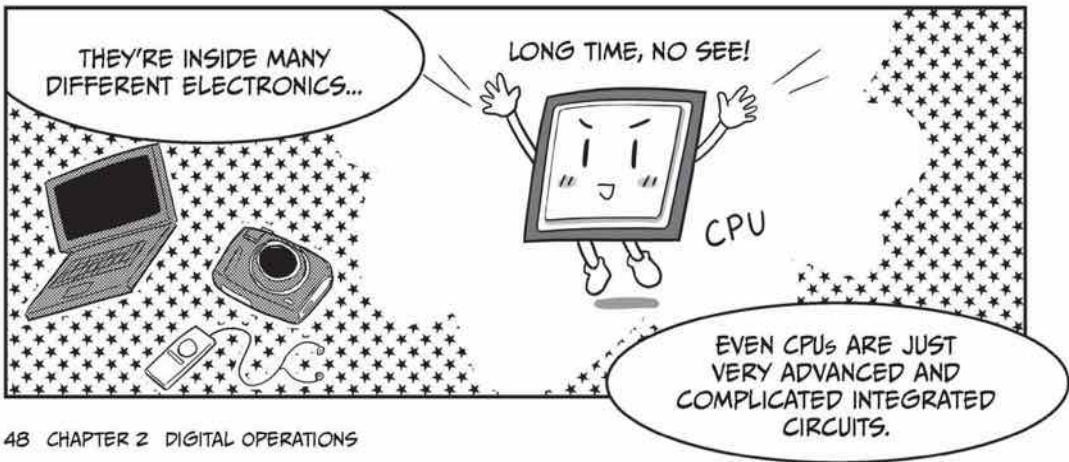
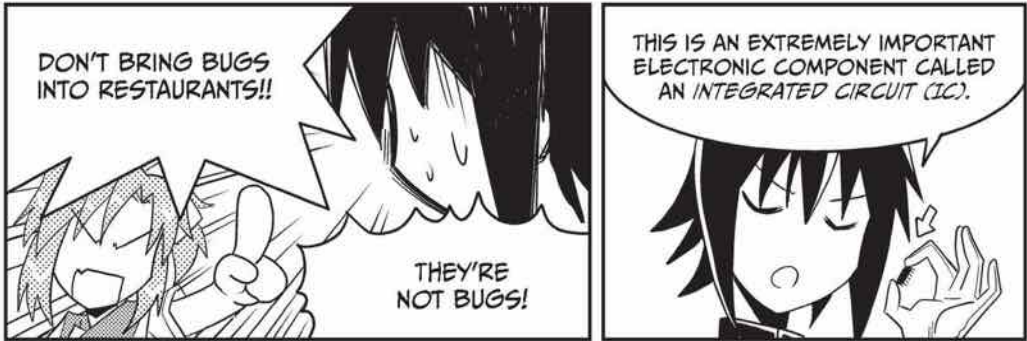
And since computer calculations only deal with 1s and 0s, this method is both really simple and incredibly fast at the same time.

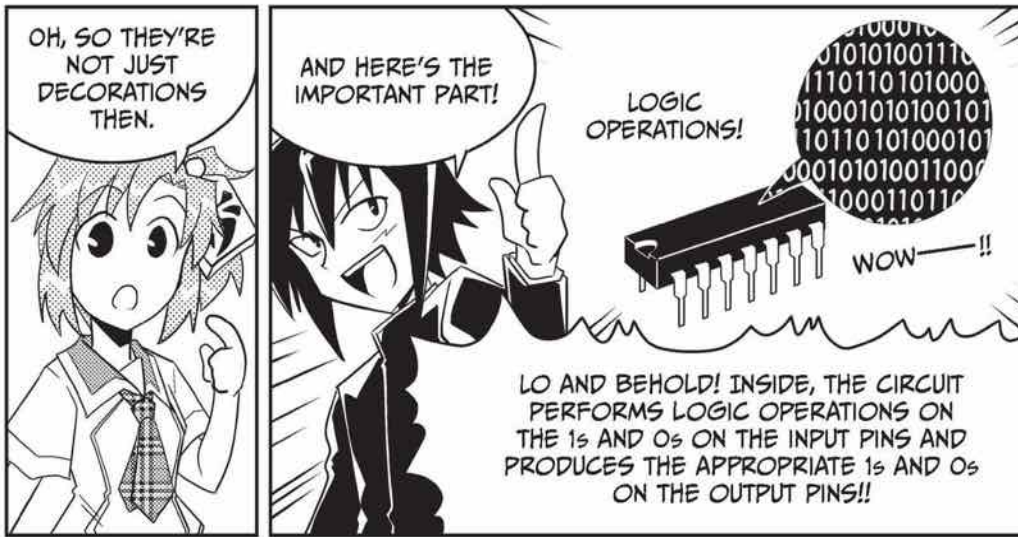


I see. So there are some merits to binary, I suppose!

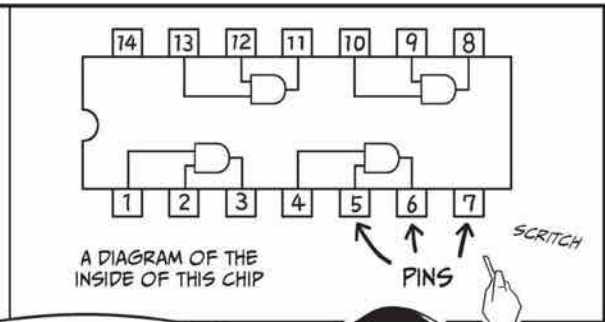


What Are Logic Operations?





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



A DIAGRAM OF THE INSIDE OF THIS CHIP

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.

A close-up diagram of a logic gate symbol. It has two inputs labeled 'INPUT A' and 'INPUT B' on the left and one output labeled 'OUTPUT' on the right. An arrow labeled 'ATTENTION!' points to the symbol. Above the diagram, three pins are shown with arrows pointing to them, labeled 'PINS'.

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...

A character with dark hair and a black jacket is standing with their arms crossed, looking at the diagram.A character with long, light-colored hair is looking thoughtful, with one hand on their chin.

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

INPUT A — [LOGIC GATE] — OUTPUT Z

INPUT B

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

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

YEAH, THAT'S RIGHT.

THE THREE BASIC LOGIC GATES:
AND, OR, AND NOT

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

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

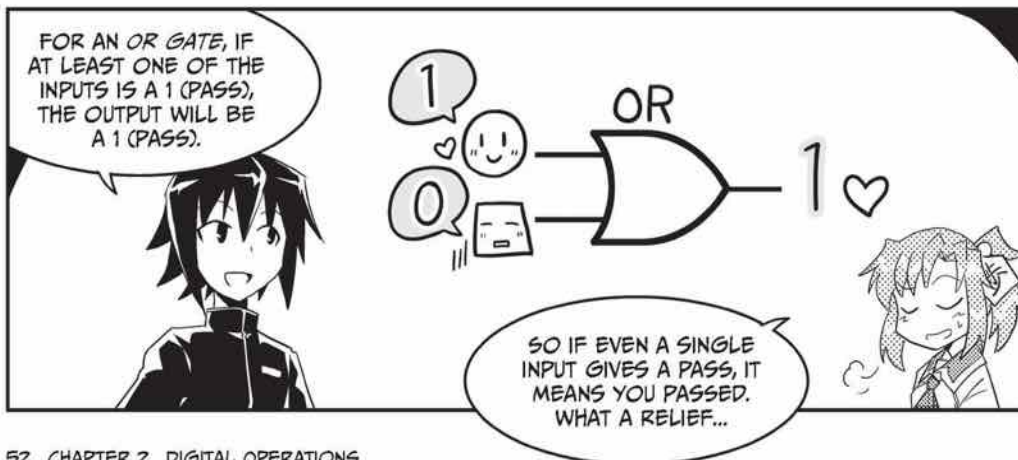
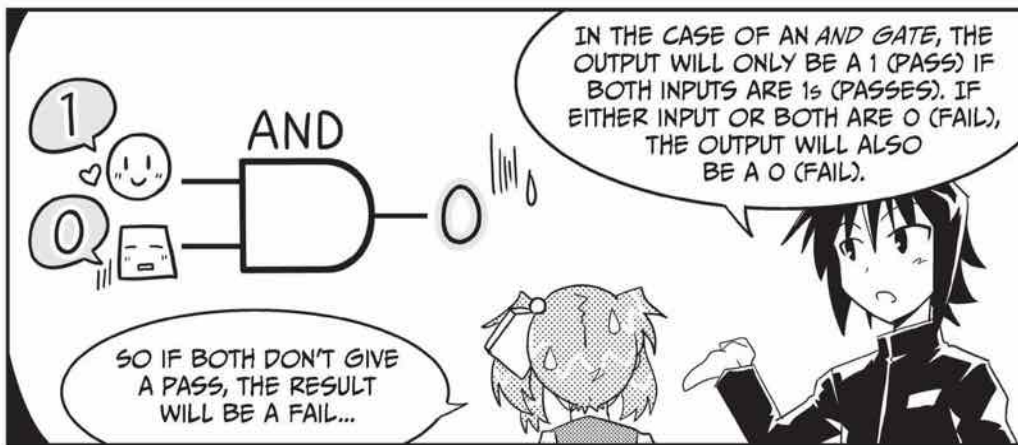
AND	OR	NOT

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!!!

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.

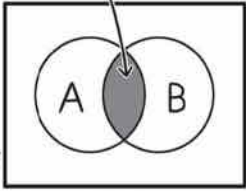
OH, I REMEMBER
THOSE FROM
JUNIOR HIGH.

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

ASSUMING
THE WORLD CONSISTS
ONLY OF REGIONS
WITHOUT COLOR (0)
OR WITH COLOR (1)...

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

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



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

THAT'S RIGHT.
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 they 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 .

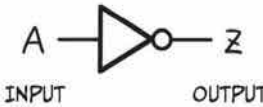
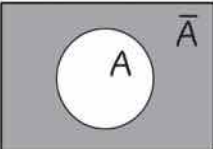
OR GATE (LOGIC UNION GATE)

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															

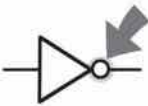
OR gates output 1 when either input or both is 1, and they are sometimes expressed in equation form as $Z = A + B$. The symbols used to represent OR are those for logical unions: $+$ or \cup .



NOT GATE (LOGIC NEGATION GATE)

SYMBOL	TRUTH TABLE	VENN DIAGRAM						
 <p style="text-align: center;">INPUT OUTPUT</p>	<table border="1" style="margin: auto;"> <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 style="text-align: center;">$Z = \bar{A}$</p>
A	Z							
0	1							
1	0							

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



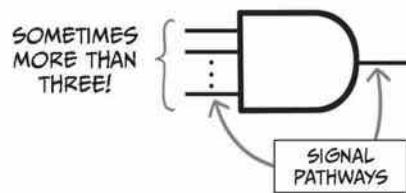
THIS WHITE CIRCLE INDICATES THAT 0 AND 1 SHOULD BE FLIPPED!



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



Good. 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.



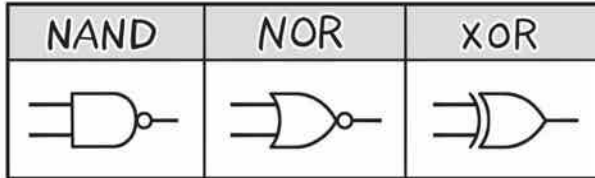
In these cases, we require that all inputs of the AND gate be 1 for the output to be 1. In the case of OR gates, we require that at least one input be 1 for the output to be 1.



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...



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



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



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





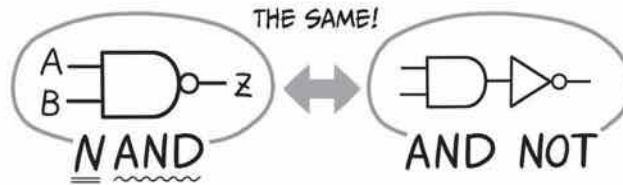
A SUMMARY OF THE NAND, NOR, AND XOR GATES



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

NAND GATE (LOGIC INTERSECTION COMPLEMENT GATE)

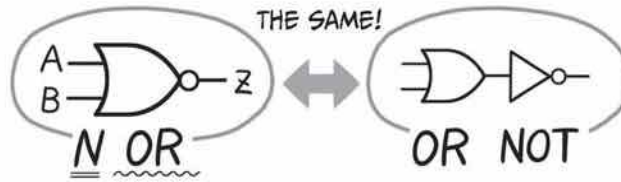
SYMBOL	TRUTH TABLE	VENN DIAGRAM															
<p>INPUTS OUTPUT</p>	<table border="1"><thead><tr><th>A</th><th>B</th><th>Z</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr>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

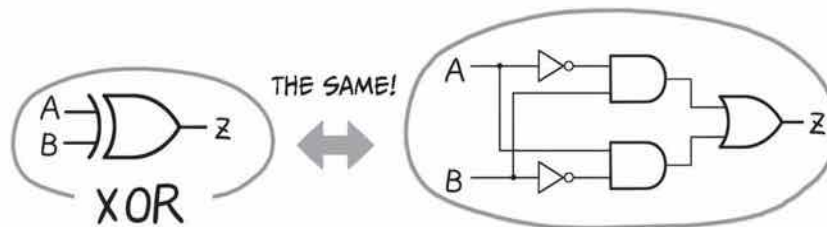
SYMBOL	TRUTH TABLE	VENN DIAGRAM															
<p>INPUTS OUTPUT</p>	<table border="1"><thead><tr><th>A</th><th>B</th><th>Z</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr>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



The XOR gate outputs 1 only when the inputs A and B are different. This gate is sometimes written as the equation $Z = A \oplus B$.

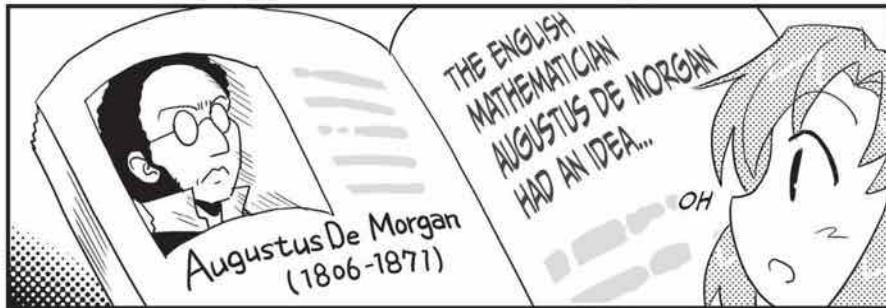
The XOR gate's function is shown in the schematic above, where you see a combination of AND, OR, and NOT gates. The X in XOR stands for *exclusive*.



Oh! You were right. These gates really are just combinations of the three 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: De Morgan's indispensable laws for logic operations. Here it is!

DE MORGAN'S THEOREM

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

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



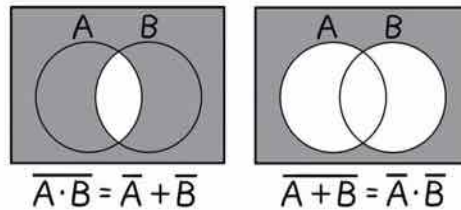
Aah, I might have eaten a little too much today. But fast food can be really good sometimes, don't you think?



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 a NAND gate is the same as using an OR gate on each input's complement, and a NOR gate is the same as using an AND gate on each input's complement. 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). So according to De Morgan's law you can swap AND for OR operators and vice versa by using complements.



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. Here are some conversions using De Morgan's laws.



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



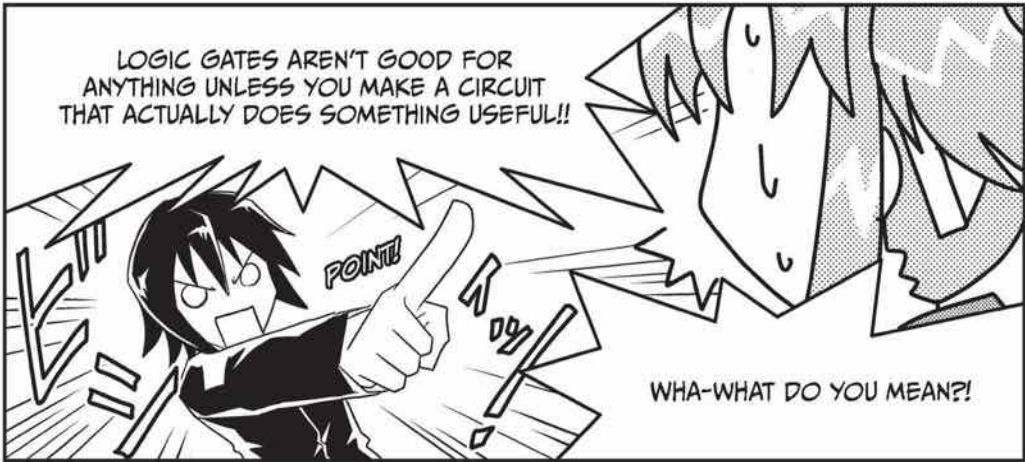
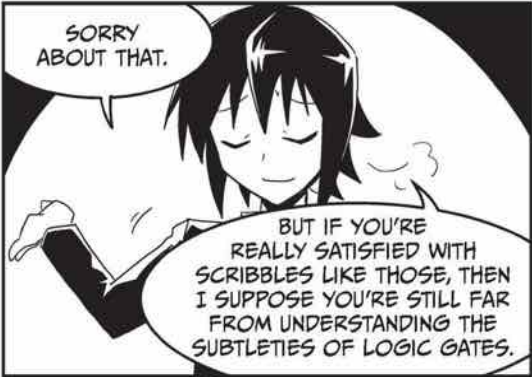
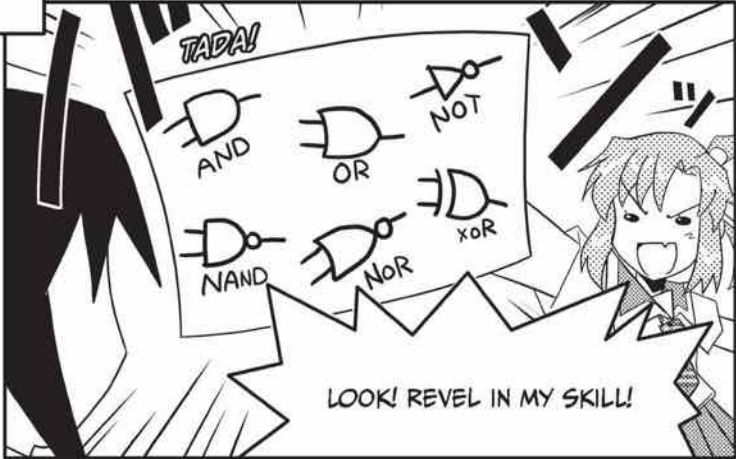
Yeah, the expressions might be different, but their functions are the same. Since logic gates (digital gates) only work with 1s and 0s, everything stays logically the same even if you switch out all the gates. We're just leveraging that particular feature of the math.

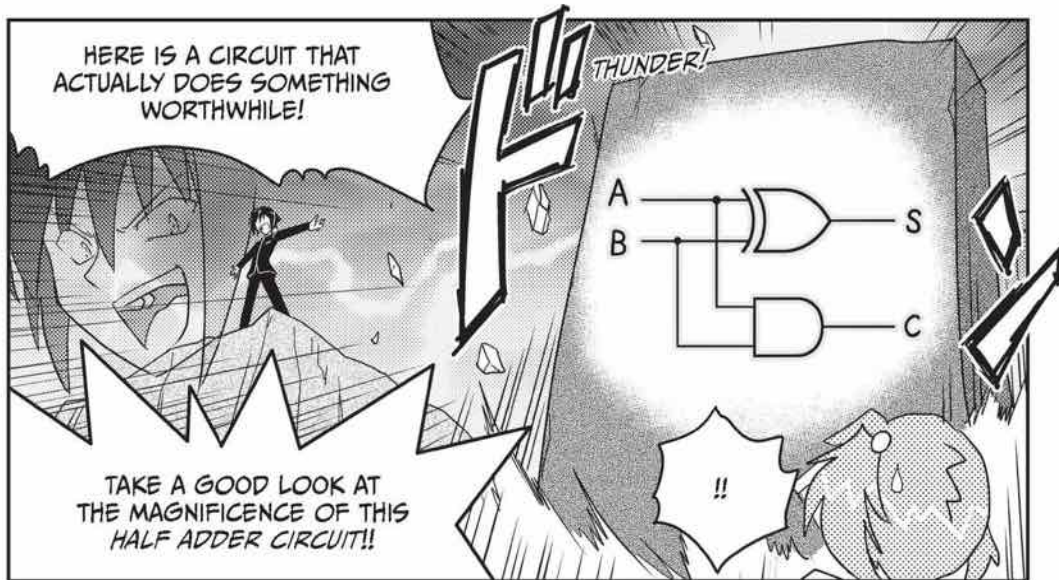


I see. . . . Then you won't mind if I just rewrite all of them? This is a law I like!

Circuits That Perform Arithmetic

THE ADDITION CIRCUIT







THE HALF ADDER



Let me explain what the *half adder* I showed you is all about (though I suspect you won't need that much explanation 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

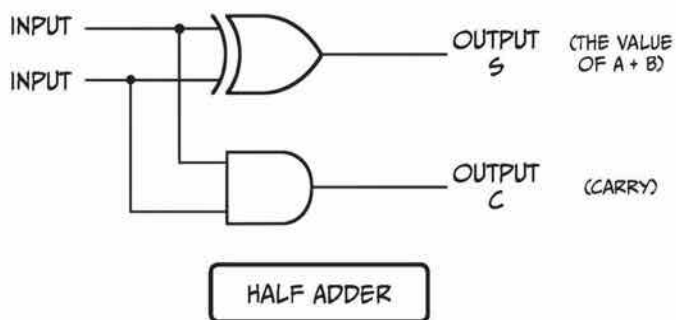


Hmm, that looks just like the truth table for an AND gate (see page 55)! An AND gate's output is 1 only when both inputs are 1. . . .

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



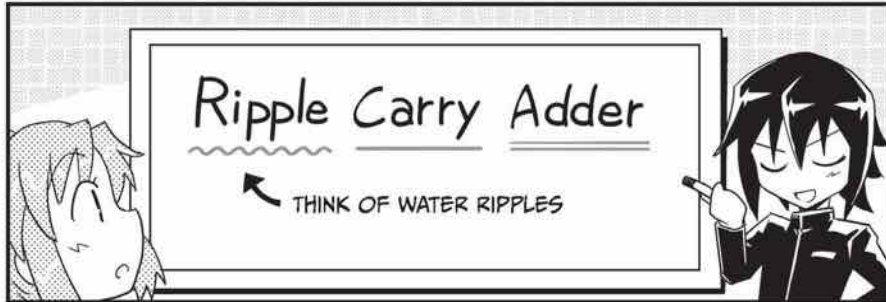
As soon as you get that part, it seems really easy, right? The lower digit comes from output S, and the upper digit comes from output C. In this case, S stands for *sum*, and C for *carry*.



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!



THE 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's 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.

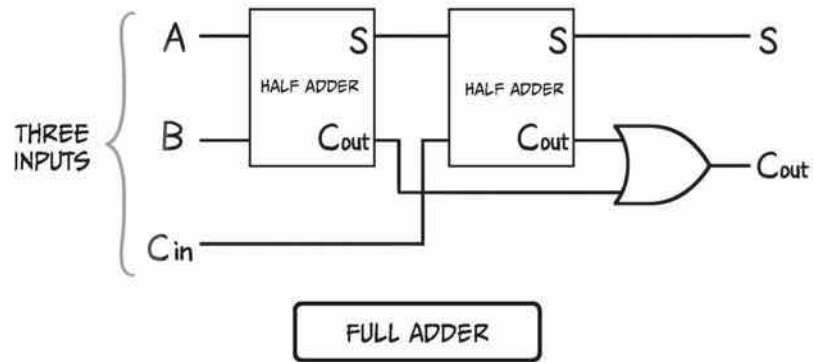


I'm not dissing anyone! Why am I the bad guy all of a sudden?!



Don't underestimate the half adder though! By 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.

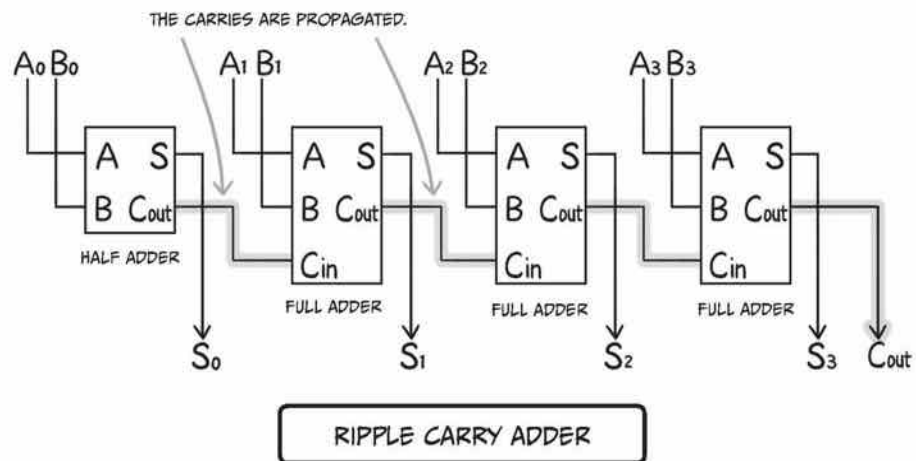


You were right—it's using two half adders! Two halves really make a whole. I guess C_{in} is the carry input and C_{out} is the carry output then.



That's right. And by connecting one half adder and several full adders, 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 individual adders into their own boxes. During subtraction, we would deal with the inverse carry instead (borrow).



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



THE CARRY LOOK-AHEAD ADDER

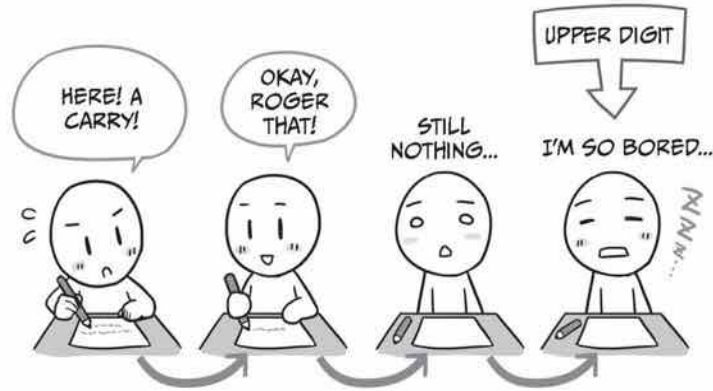


But even then . . . that ripple carry adder kind of makes me feel like there's something familiar with how it moves the carry after each step in the calculation. It's really similar to how we humans do calculations with pen and paper by moving the carry from each lower place value to the next higher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



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



PROPAGATION DELAY IN A RIPPLE CARRY ADDER



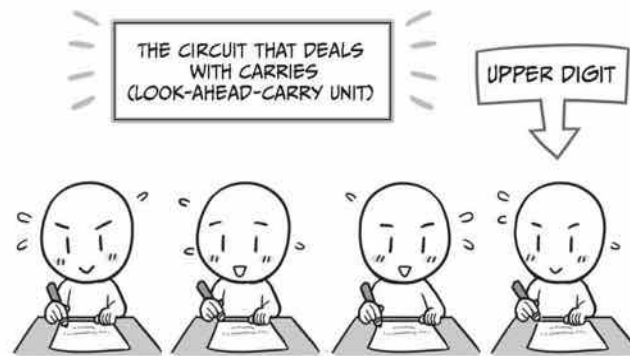
Yeah, that seems a bit slow. . . . Addition and subtraction are pretty common, too, so I suppose they're not something you want to be doing slowly. Hmm. So what do we do about it?!



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



The carry look-ahead adder 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!



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

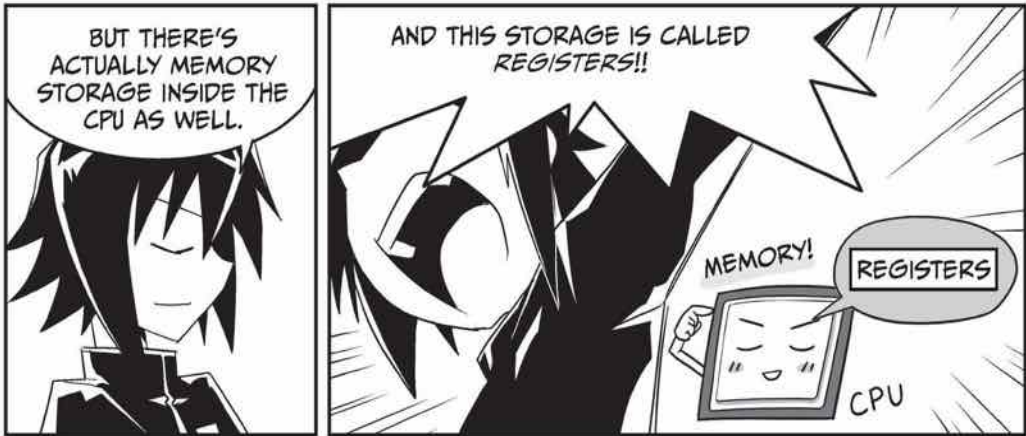
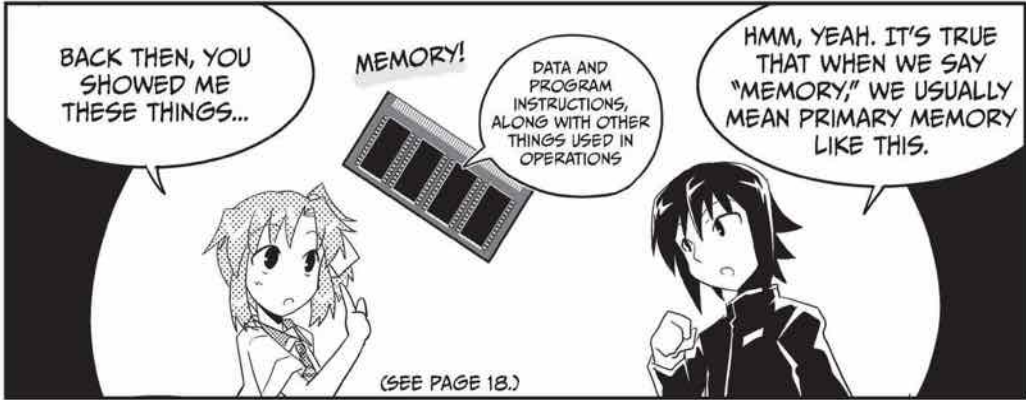
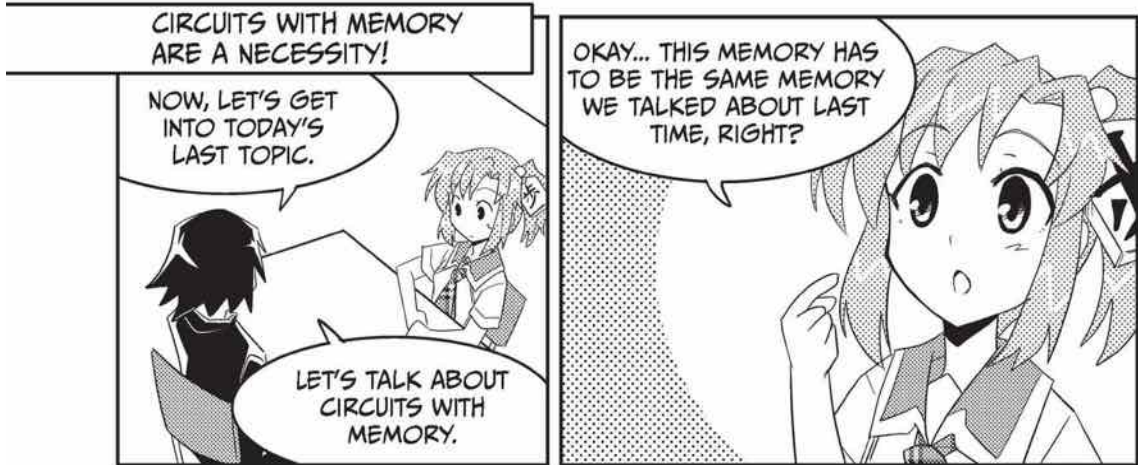


Yeah. It determines whether there is a carry in either direction during addition and subtraction. The downside is that the circuit is a lot bigger, but calculation times are drastically reduced.

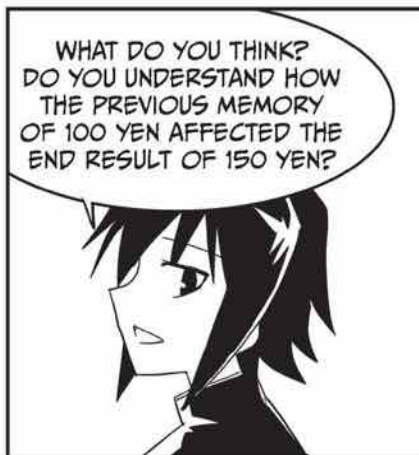
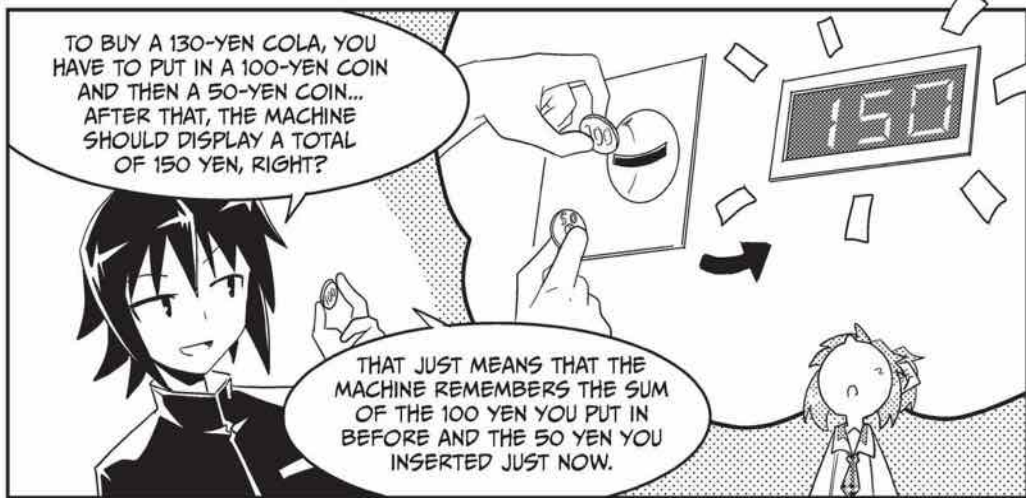


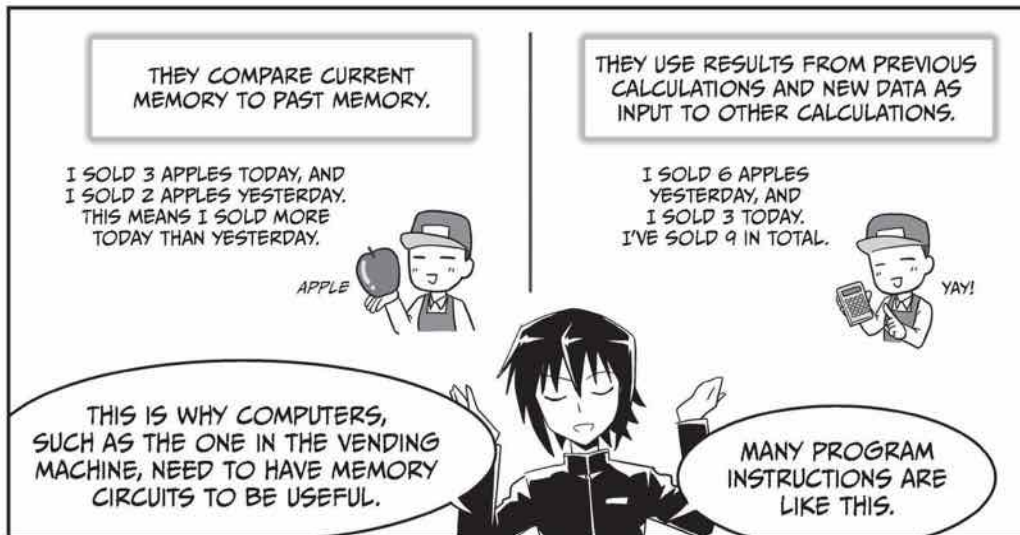
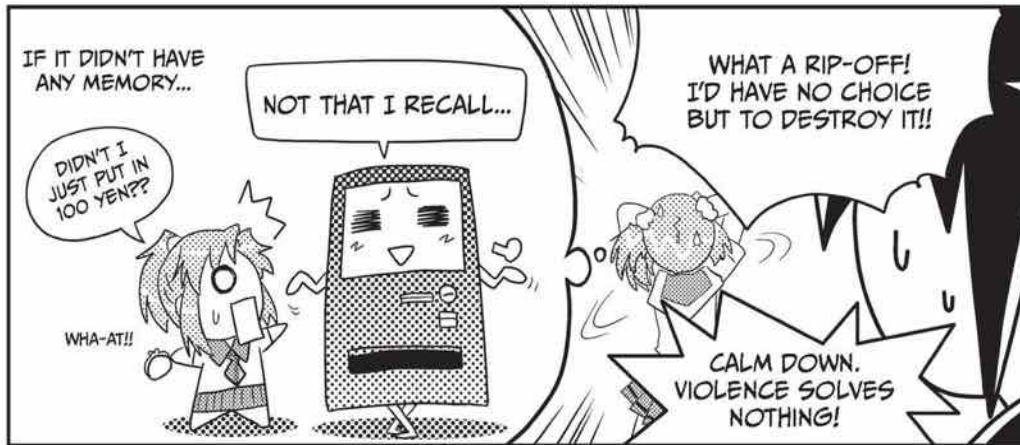
Hmm. So it's reducing calculation times with all kinds of smart tricks then. When we first talked about making a circuit for addition, I was imagining something pretty small, but the final product is quite impressive.

Circuits That Remember











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 simply. Computers can only use 1s and 0s, right? That means that to a computer, *memory* 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 turn on a room's light switch, it would stay on until someone turns it off again, and then it would stay off until someone turns it on again. It would be great if we could create some kind of *trigger condition* to freely swap 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!



Um, 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.



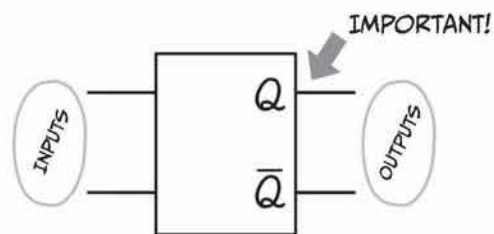
It is selfish, but *flip-flop* circuits are a basic component of any memory circuit.



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



They're super useful!! They grant us the ability to change states. 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.



Okay. 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!



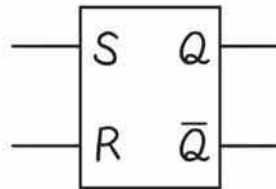
All in good time. First off, there are several types of flip-flops. Both the function and circuit depend on the type. Out of these types, I'll teach you about RS flip-flops, D flip-flops, and T flip-flops.



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 jumping to 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.



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?



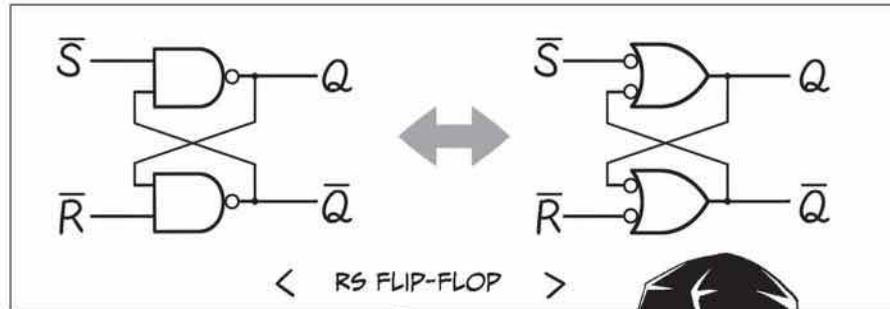
Yeah. It might seem a bit complicated here, but the circuit looks like the figure on the next page. In accordance with De Morgan's laws (see page 60), it can be created using either NAND gates or NOR gates.



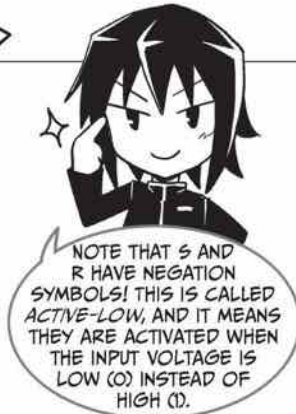
Whoa. It looks a bit weird. . . . There are two NAND gates (or NOR gates), but they're all tangled up in figure eights.



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



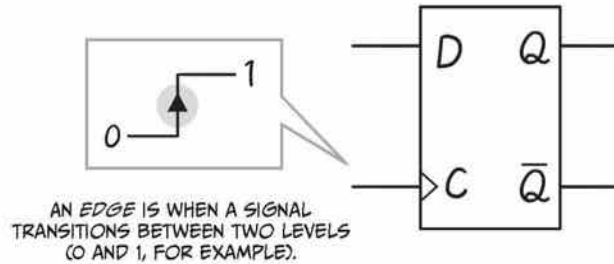
Oh, I see. So just follow the traffic, er, circuit rules, right?



THE D FLIP-FLOP AND THE CLOCK



Let's see. The next one is the *D flip-flop*. 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 standing for *data*. That triangle is the symbol for a rising edge, and the *C* stands for *clock*.

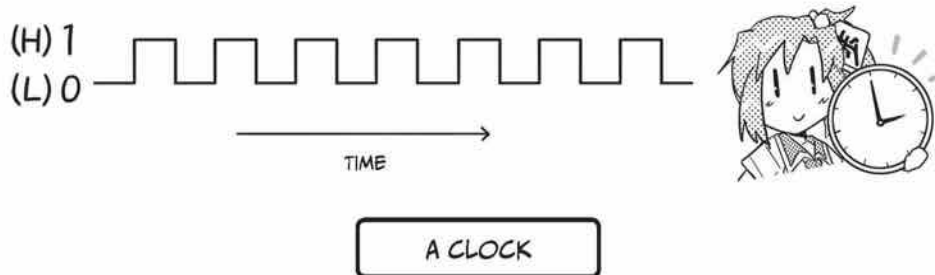


Um . . . Rising edge?? And the clock—is that just a normal clock?



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.

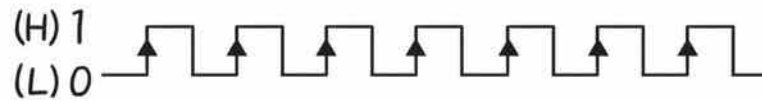




Hmm. It really reminds me of a clock . . . tick-tock, tick-tock. . . . Just like we plan our days with the help of clocks, I guess circuits need them, too.



Yeah. When a circuit needs to take some action, the clock can sometimes act as its cue. 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 as a signal to take action, like at the start and end of class, for example.

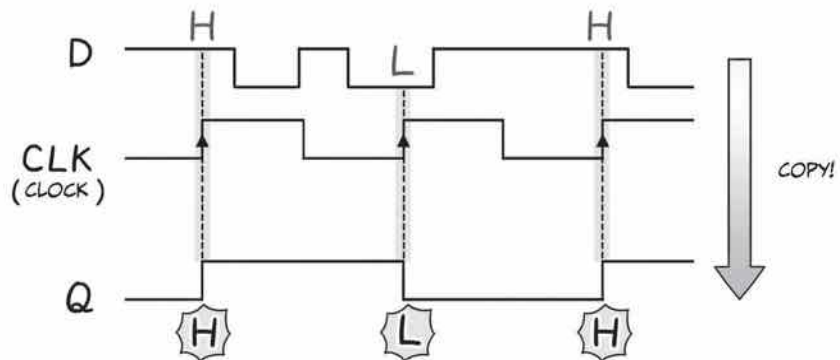


That's just it! That's a pretty good analogy coming from you.



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

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.



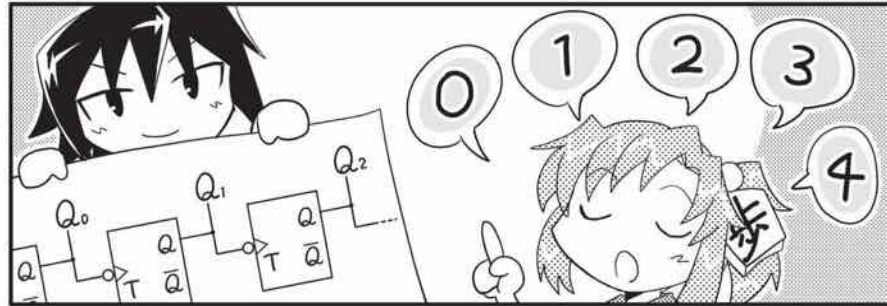
The important lesson here is that the D input can change as much as it wants, but Q won't change until a rising edge arrives!



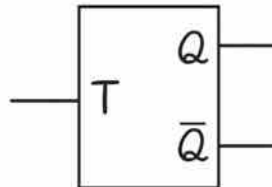
Mmmh. It's a bit complicated, but I think I get it now that I've looked over the timing diagram. In any case, the main characteristic of the D flip-flop seems to be that it acts in sync with the clock's rising edges! Hmm, it seems like clocks are super important both to modern man and circuits.



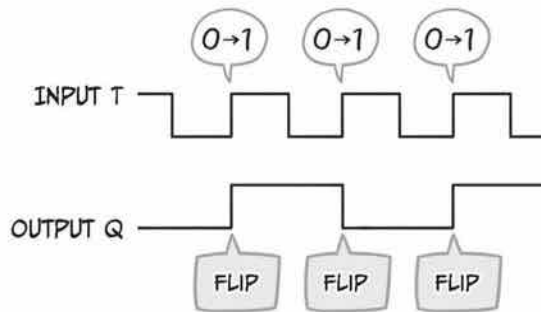
THE T FLIP-FLOP AND COUNTERS



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



Fuhahaha! Like I would ever forget! The *T flip-flop* has only 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.



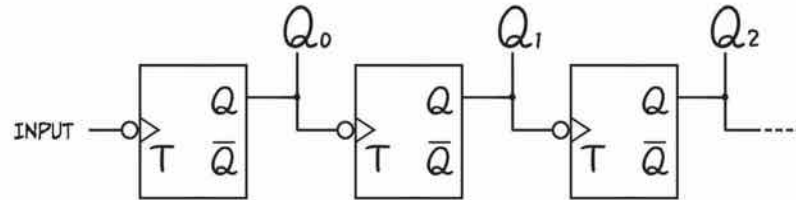
THERE ARE T FLIP-FLOPS THAT ACTIVATE JUST ON FALLING EDGES INSTEAD (1 TO 0).



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, as in the following schematic, you can make a circuit that can count—a *counter circuit*.



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 $2n$ 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.

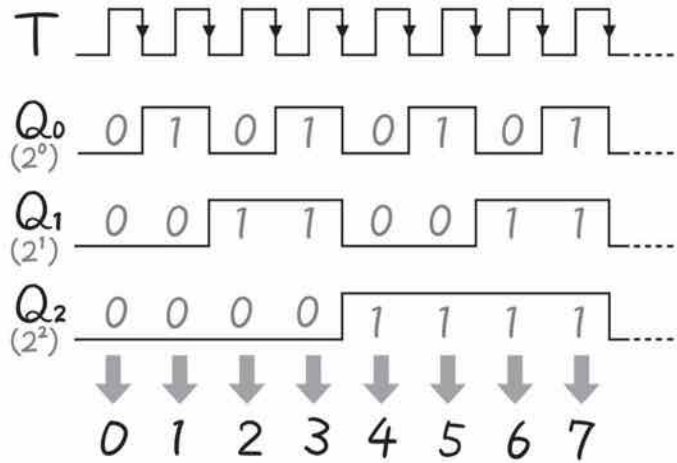


Umm, but why do we say that the circuit can count?



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.

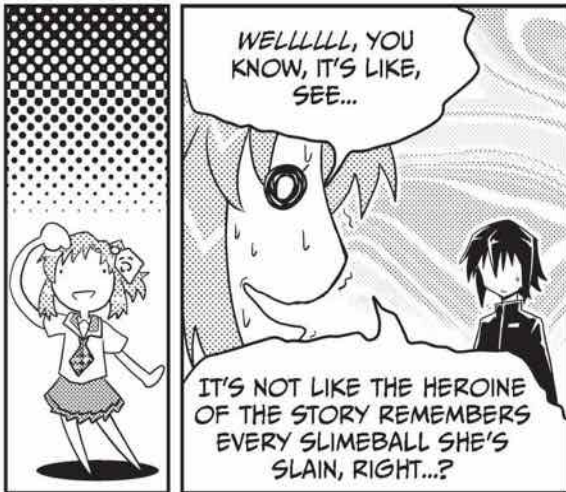
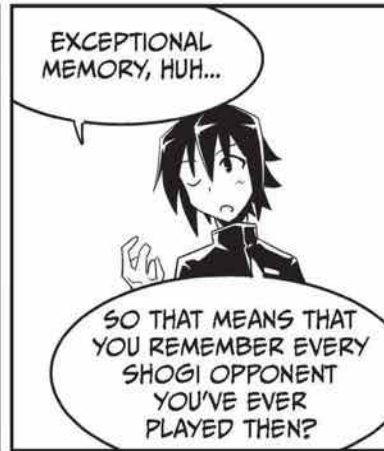


Yeah, well that's it for flip-flops. Just don't forget what I said at the start: flip-flops are the foundation of any memory circuit!

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!



MODERN CIRCUIT DESIGN: 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 ones we have shown in this book, but these symbols are now used mostly 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. It can be useful to know these symbols if you're trying to figure out how data flows through a digital circuit or when you're 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.

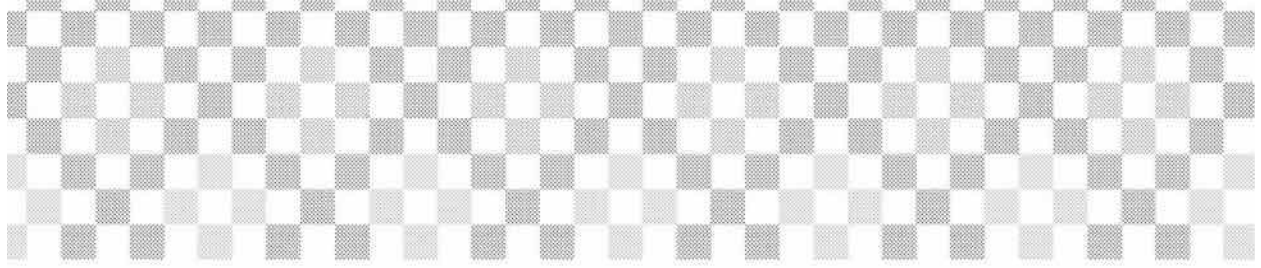
Nowadays, reference circuits like these are rarely used in development. Instead much more flexible *field-programmable gate array (FPGA) circuits* are preferred.



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.

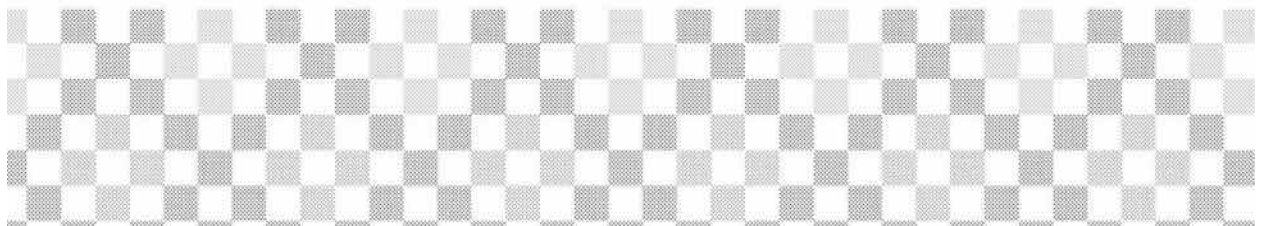
FPGAs consist of a series of logic blocks that can be wired together in different ways depending on the programming. Some of these blocks contain lookup tables to map the 4–6 bits of input to output in a format that's similar to a truth table. The number of lookup tables in an FPGA can range anywhere from a few hundred to more than several million, 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.



3

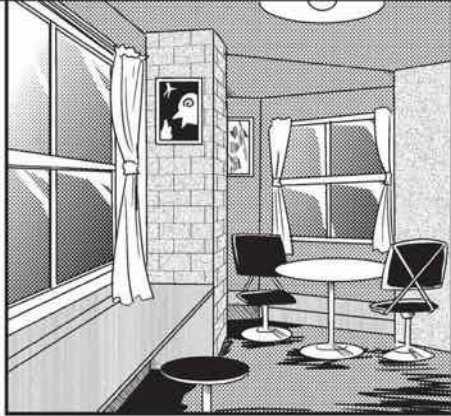
CPU ARCHITECTURE



All About Memory and the CPU



MEMORY HAS ASSIGNED ADDRESSES



WELL THEN...



DO YOU KNOW WHAT ADDRESSES ARE?



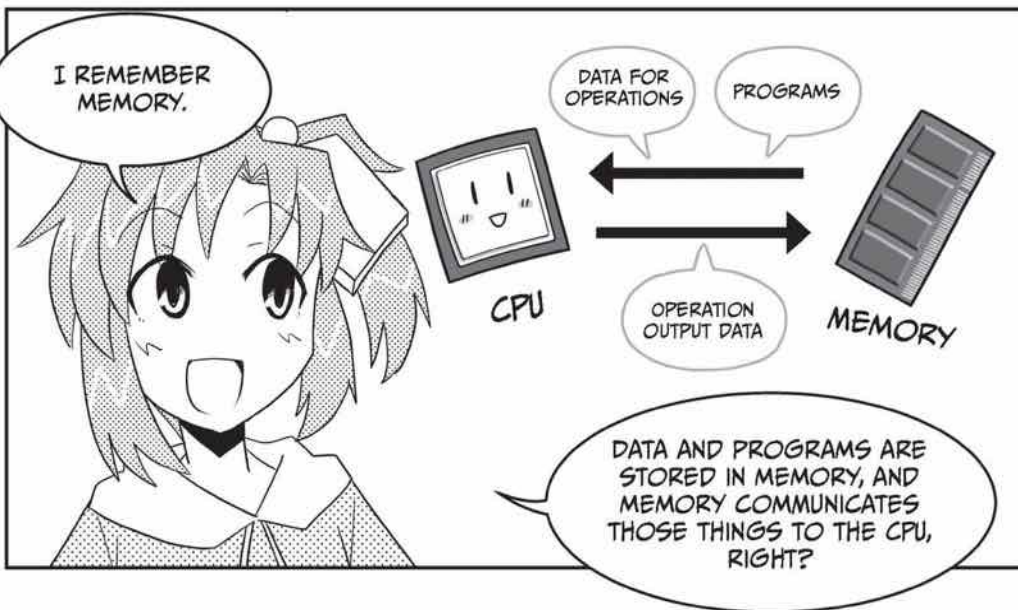
OF COURSE I DO!
LIKE I WOULDN'T KNOW
WHERE MY FRIENDS LIVE!!

UH, I WASN'T
TALKING
ABOUT MAILING
ADDRESSES.



YOU SEE, LOCATIONS
IN MEMORY...

...ARE CALLED
ADDRESSES.



THAT'S RIGHT.

PROGRAMS AND DATA ARE SAVED TO MEMORY IN AN ORGANIZED WAY.

EVERYTHING IS ASSIGNED A UNIQUE ADDRESS DEPENDING ON WHERE IT IS STORED.

《 INSIDE MEMORY 》

ADDRESS	TYPE
0	INSTRUCTION
1	INSTRUCTION
2	INSTRUCTION
3	INSTRUCTION
4	INSTRUCTION
⋮	⋮
30	DATA
31	DATA
32	DATA
33	DATA
⋮	⋮

PROGRAMS

DATA FOR USE IN OPERATIONS

YOU'RE RIGHT!! EVERYTHING HAS A UNIQUE ADDRESS AND IS NEATLY SORTED IN ORDER.

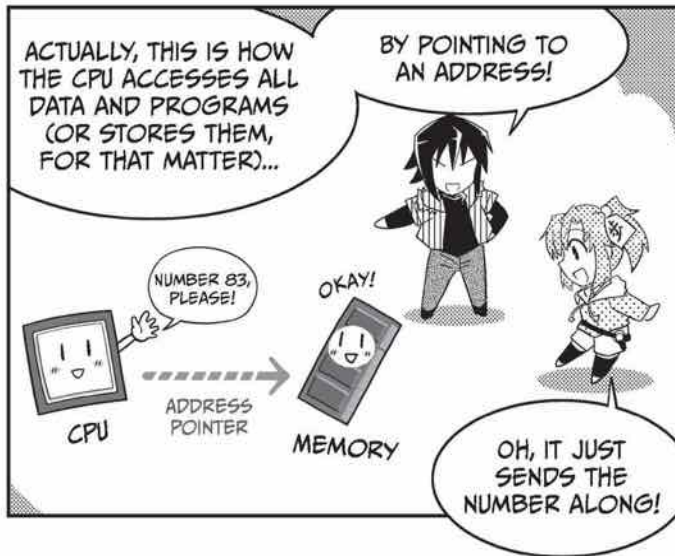
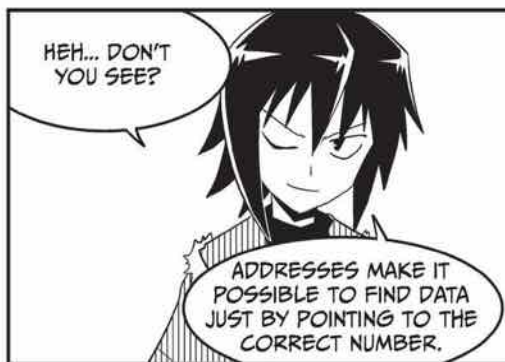
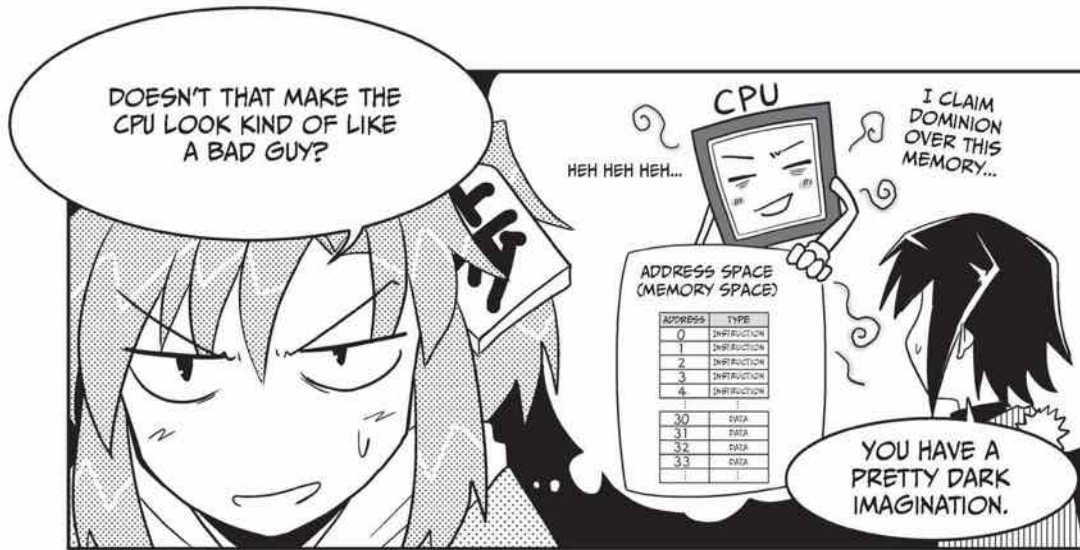
YOU SHOULD ALSO REMEMBER THAT THE CPU... HAS FULL CONTROL OVER THIS ADDRESS SPACE, WHICH CAN ALSO BE CALLED MEMORY SPACE.

ADDRESS	TYPE
0	INSTRUCTION
1	INSTRUCTION
2	INSTRUCTION
3	INSTRUCTION
4	INSTRUCTION
⋮	⋮
30	DATA
31	DATA
32	DATA
33	DATA
⋮	⋮

ADDRESS SPACE (MEMORY SPACE)

UH-HUH. SO THE CPU CONTROLS THE MEMORY SPACE...

AND IF IT CAN READ AND WRITE WHEREVER IT WANTS...



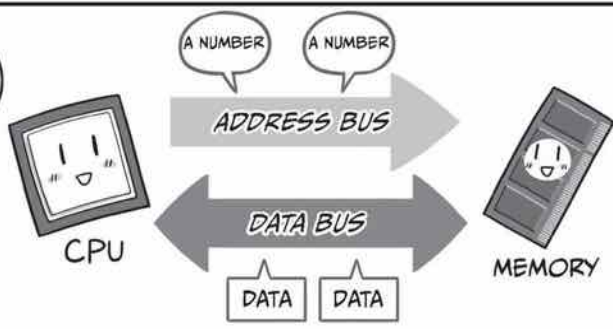
DATA PASSES THROUGH THE BUS

WELL SINCE WE ALREADY TALKED ABOUT ADDRESSES, LET'S ALSO TALK A BIT ABOUT THE BUS.

THE BUS...?


YEAH, IT ACTUALLY COMES FROM THE WORD OMNIBUS.
BUT THE KIND OF BUS WE'RE TALKING ABOUT IS A PATH THAT TRANSMITS DATA INSIDE A COMPUTER.

AND, AS SHOWN IN THIS PICTURE, WE CALL THE BUS THAT TRANSPORTS ADDRESS POINTERS THE ADDRESS BUS.



APPROPRIATELY, WE ALSO CALL THE BUS THAT TRANSPORTS DATA—WAIT FOR IT—THE DATA BUS.

I SEE. SO THEY'RE KIND OF LIKE TWO ROUTES WITH COMPLETELY DIFFERENT PURPOSES.

YEAH, AND THERE ARE ALSO EXTERNAL BUSES AND THE INTERNAL DATA BUS.

THE EXTERNAL BUS CONNECTS THE CPU TO MEMORY AND EXTERNAL DEVICES. THE MOST COMMON ONE IS THE UNIVERSAL SERIAL BUS (USB).

INTERNAL DATA BUS

EXTERNAL BUS

CPU

MEMORY

OH, JUST LIKE THE NAME SAYS.

THE INTERNAL DATA BUS IS A BUS THAT PASSES INTERNALLY THROUGH THE CPU.

AND INSIDE THE INTERNAL BUS, THERE ARE SWITCHES THAT CAN CHANGE THE FLOW OF DATA, JUST LIKE IN THIS PICTURE...

MULTIPLEXER (MUX)

A

B

Y

A PATH FROM A TO Y

A

B

Y

A PATH FROM B TO Y

THESE SWITCHES ARE CALLED MULTIPLEXERS (MUX).

THESE SWITCHES MAKE IT POSSIBLE TO SIMPLIFY THE CPU'S INTERNAL DATA BUSES.

THEY'RE SUPER USEFUL!

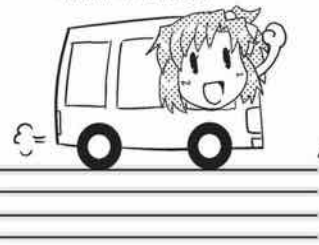
OOOH, THEY'RE REALLY LIKE BUS ROUTES AFTER ALL!

AND THEY'RE DESIGNED TO MAKE EVERYTHING AS EASY AS POSSIBLE. I CAN APPRECIATE THAT!



BUS WIDTH AND BITS

WITH 4 BITS...



0000	0100	1000	1100
0001	0101	1001	1101
0010	0110	1010	1110
0011	0111	1011	1111

$2^4 = 16$ WIDE



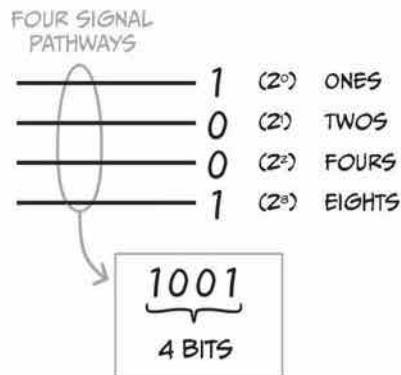
Let's talk a bit more about buses. I said that buses are for transporting data, but to be more exact, they're actually *bundled signal pathways*.



Ah, signal pathways, I remember those from before. They're lines that transmit 1s and 0s, right?



That's right! And, the number of pathways determines how many values those pathways can represent. For example, if you have four pathways, and each can be a 0 or a 1, then those pathways can be used to send or receive a four-digit (four-bit) binary number.



Uh-huh. So that means that the number of pathways equals the number of bits. With four pathways, we can send and receive 16 (2^4) different numbers from 0000 (0 in decimal) to 1111 (15 in decimal).

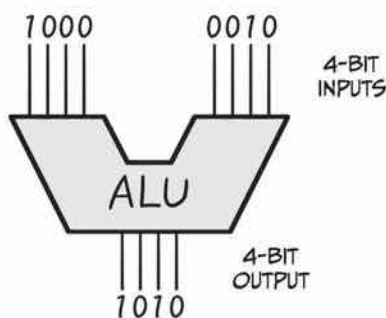


Hah! I think I might have figured out something important! Doesn't that mean that the more signal pathways you have, the better? If you can express larger numbers, that also has to mean you can process more data.

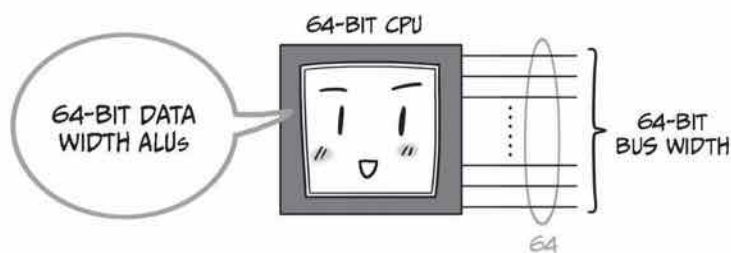


Heh heh heh, good guess. You're correct! We call the number of signal pathways (or bits) the *bus width*. A wider bus width gives the CPU faster processing capabilities.

For example, the ALU in the following diagram can process four-bit operations. This means that the data bus serving this ALU also has to be four bits wide.



We chose 4 bits to make the diagram simpler, but most modern ALUs work with 64 bits. It makes sense to use a bus with a bus width that matches the ALU's data width, which means buses end up being 64 bits wide more often than not. In general, the 64-bit CPUs that you hear so much about have 64-bit ALUs and 64-bit data buses.



It's worthwhile to point out that the data bus doesn't necessarily have to have the same width as the working bit length of the CPU. For example, in 1982, you could find 16-bit CPUs with 16-bit ALUs using an 8-bit wide data bus. This simply meant that you had to send data over the bus two times before the CPU could start doing any work.



Haha! So that just means that the CPU's performance is determined in part by the width of the data bus. And the wider the data bus, the better the CPU's performance!



You should try to remember these things about data bus width.

By looking at the width of the external bus between the CPU and memory, you can see how many bits can be sent between these two systems at once. By looking at the width of the internal data bus, you can see how many bits can be processed at the same time in a single operation.



Okay! So data bus width is super important. Gotcha!



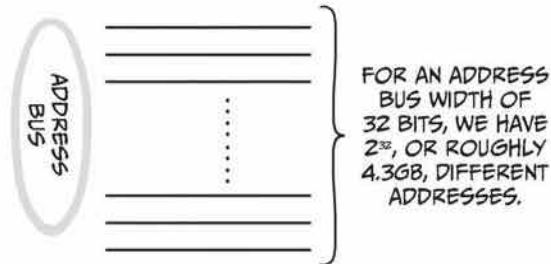
That's it for now about data buses. Let's talk a bit about the address bus. By looking at the address bus width, you can see how large your system memory is. We call this the *address space size*.



The address space size ... ? Does that mean how many addresses there are? Then if the address bus has a width of 4 bits, wouldn't that give us 2^4 , so a total of 16 different addresses?



Yeah. On the other hand, if we have an address bus width of 32 bits, that would give us 2^{32} , or 4,294,967,296, different addresses. We can say that the size of our address space for 32 bits is roughly 4.3 gigabytes.





And the size of the address bus directly relates to memory capacity (see "Memory Capacity and Bus Width" below).



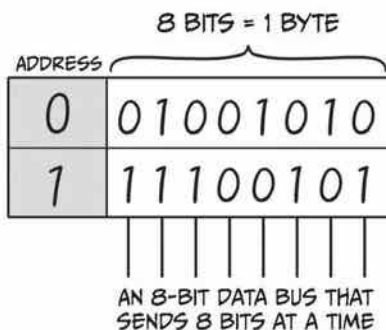
Hmm, so address bus width is also really important. Does that mean it should be as large as possible?



Is that greed or naive honesty I detect? Well I suppose it's true that the bigger the buses, the better.

MEMORY CAPACITY AND BUS WIDTH

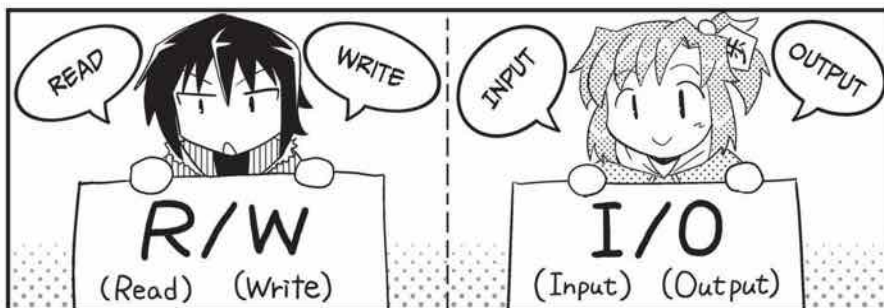
Let's think a bit about the relationship between memory capacity and bus width by looking at a simple example. As shown in the diagram, one byte corresponds to one address. One byte is eight bits. A *byte* is a unit commonly used to describe the size of data.



If instead our address space (address bus width) was 12 bits, then we would have 2^{12} , or 4,096 addresses. Each address fills 1 byte, so that means our memory capacity is 4,096 bytes, or roughly 4 kilobytes (KB).



R/W SIGNALS AND I/O SIGNALS



Next up, I'll explain a bit about *control signals*. Do you happen to know what *R/W* stands for?



Red and white . . . is it something related to festivities . . . ? Santa?



Ah. Well, it sure seems festive inside that head of yours. It actually stands for two really important terms related to the CPU—*read* and *write*.

Read means to extract data that has previously been stored somewhere. *Write* means to save data to some place. We also sometimes use the words *load* and *store* instead.

WHAT IS THE DIFFERENCE BETWEEN READ/WRITE AND LOAD/STORE?

Read/write is the term we use when speaking from the hardware perspective, and *load/store* is the term we use when speaking from the software perspective.

R/W is an electrical operation in the memory, and the memory doesn't care where the data is going or where we want to save something.

In contrast, a *load* operation reads some particular data to store it in a register. Conversely, a *store* operation grabs some register data and writes it to memory. So these operations deal with data flow.

R/W

L/S

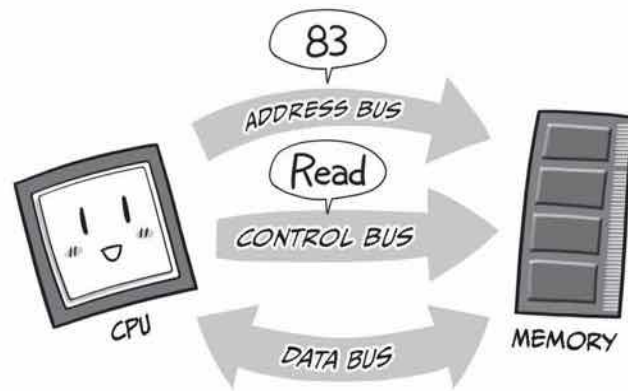


Ah, in that case, I think I kind of get it. The CPU deals with memory and data, right? It reads data to use for operations and then writes the result of those operations to memory.



Yes! It seems you're really getting the hang of it. So the CPU issues read and write instructions to the memory—instructions such as "Fetch some data" or "Save some data." We call these *R/W instruction signals*.

We've talked about the address bus and data buses, but there is one more really important bus—the *control bus*! And it's this bus that is conveying these control signals from the CPU.



Hmm. So if, for example, I wanted the data in number 83, I would send 83 on the address bus and read on the control bus, like in the picture! And the data I wanted would then arrive on the data bus.



Yeah, that's it. You seem to be getting it now, so let's move on to the next topic. Have you heard of I/O before?



Maybe . . . ice cream and oreos? Hee!



Ah, you're just choosing words at random now, huh? *I/O* stands for *input/output*.

Input is data traveling from the outside world into the computer. Conversely, output is data traveling from the computer to the outside world.

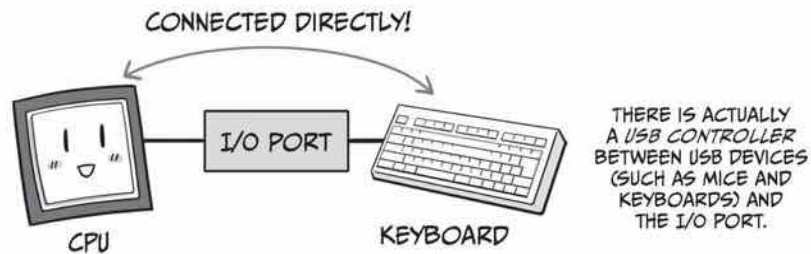


Yeah, I know. The keyboard and mouse are two input devices, and the monitor and printer are two output devices, right? Input and output!



Yeah. To control external devices like this, we use *I/O* control signals. You should also remember the term *I/O port*. Just as the word *port* suggests, ports are the gateway we use when communicating data to external devices.

The CPU is connected to external devices like keyboards* through these *I/O* ports! Have a look at the image below.



Port ... Yeah, it really feels like there are unknown seas at the end of the map! But these ports don't connect to other lands; they just connect to other devices.



Yeah. And additionally, we have an address port and a data port between the CPU and memory, which in turn connect to the address bus and data bus, respectively.**

* Other external devices, such as the display, are not necessarily in direct communication with the CPU, however.

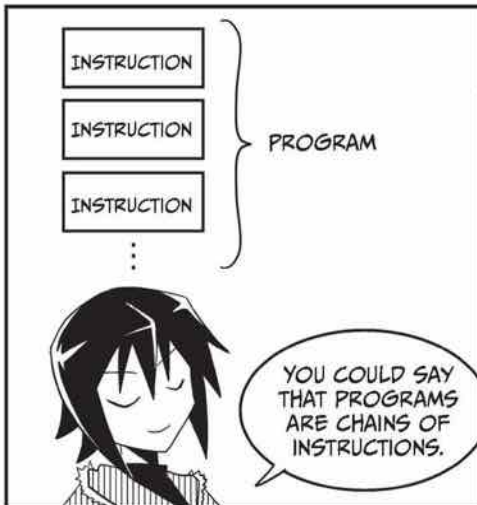
** The address port, data port, R/W control, and I/O control are shown in the helpful CPU overview diagram on page 106.

INSTRUCTIONS ARE MADE OF OPERANDS AND OPCODES



THESE

0	INSTRUCTIONS
1	INSTRUCTIONS
2	INSTRUCTIONS
⋮	⋮



YEAH. ALTHOUGH THE INSTRUCTIONS WE'RE TALKING ABOUT ACTUALLY LOOK LIKE THIS.

OPCODE
2 + 3
OPERAND OPERAND

THE *OPCODE* (SHORT FOR OPERATION CODE) IS WHAT TO DO, AND THE *OPERAND* IS WHAT THE CPU OPERATES ON.

LOTS OF OPCODES

COMPARE THESE TWO

STORE THIS

JUMP

AND THERE ARE LOTS OF INSTRUCTIONS OTHER THAN JUST "ADD THESE TWO."

OH

UH-HUH! BUT THE IMPORTANT PART IS THAT INSTRUCTIONS TELL THE CPU WHAT TO DO AND WHAT TO OPERATE ON, RIGHT?

YES, BUT BE CAREFUL! OPERANDS MIGHT ALSO BE AN ADDRESS INSTEAD OF A VALUE.*

THE OPERAND IS AN ADDRESS!
ADD THE DATA AT ADDRESS 30 TO THE DATA AT ADDRESS 31.

* THE ACCUMULATORS AND REGISTERS WE'RE GOING TO TALK ABOUT NEXT CAN ALSO BE OPERANDS.

A-HA, SO IT'S LIKE THIS THEN.

INSTRUCTIONS (THE PROGRAM) THEMSELVES ALSO RESIDE AT SOME ADDRESS...

AND THE OPERANDS THE INSTRUCTION OPERATES ON ALSO RESIDE AT THEIR OWN ADDRESSES.

IT'S NOT THAT HARD AFTER ALL!

MANAGING ALL OF THEM BY NUMBER... IT'S SO RATIONAL AND ECONOMICAL!

WEREN'T YOU JUST COMPLAINING ABOUT THE LACK OF HUMANITY?!

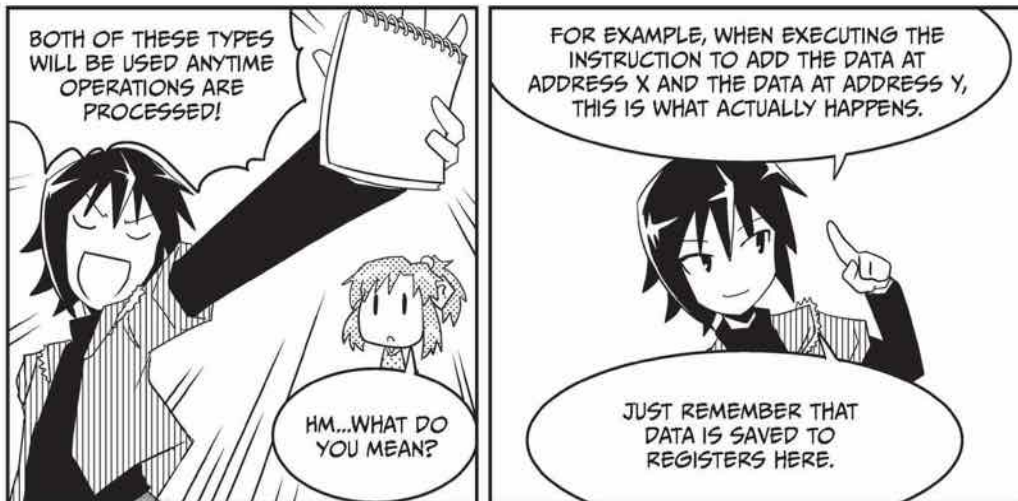
ACCUMULATORS AND OTHER REGISTERS ARE USED IN OPERATIONS

OKAY, WE JUST TALKED ABOUT THE "ADDING" INSTRUCTION...

BUT TO EXECUTE AN INSTRUCTION...

SWAT!

YOU ALWAYS NEED REGISTERS!!!!



ADDRESS X
2

ADDRESS Y
3

ACCUMULATOR

GENERAL-PURPOSE REGISTER

ALU

5
ACCUMULATOR

AFTER SAVING THE DATA STORED AT ADDRESS X (2) IN THE ACCUMULATOR REGISTER AND SAVING THE DATA STORED AT ADDRESS Y (3) IN A GENERAL-PURPOSE REGISTER, PERFORM AN ADDITION OF THE TWO.

THEN AUTOMATICALLY STORE THE RESULT OF THE OPERATION (5) TO THE ACCUMULATOR REGISTER AGAIN.

OOOOH, YEAH, I SEE A LOT OF REGISTERS BEING USED HERE!

IT SEEMS A BIT ROUNDABOUT, BUT I GUESS THAT'S JUST HOW THE CPU WORKS!

THERE ARE MANY OTHER TYPES OF REGISTERS AS WELL.

FOR EXAMPLE, THE INSTRUCTION REGISTER IS USED TO TEMPORARILY STORE PROGRAM INSTRUCTIONS READ FROM MEMORY.

INSIDE THE CPU

INSTRUCTIONS (PROGRAM)

INSTRUCTION REGISTER

MEMORY

IT EXECUTES THE INSTRUCTION AFTER DECODING* IT, HUH...

SO THERE ARE LOTS OF DIFFERENT REGISTERS FOR ALL KINDS OF PURPOSES THEN!

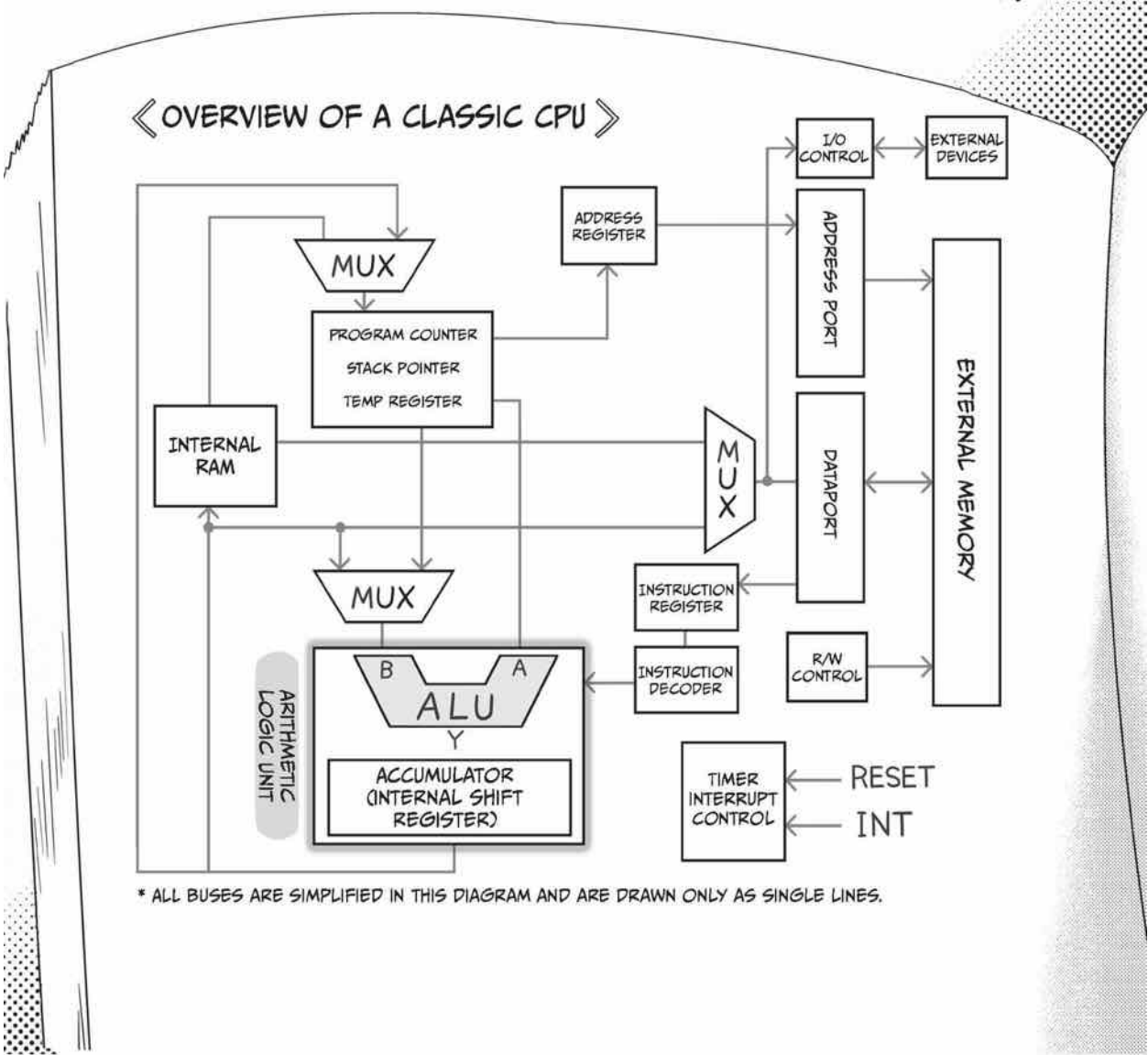
I'D BETTER REGISTER WHAT I'VE LEARNED ON THE BACK OF THIS RECEIPT!!

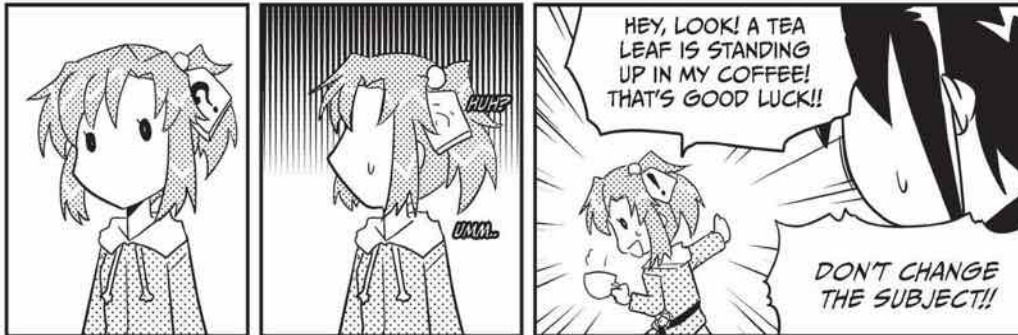
THAT'S A NOTE YOU'LL LOSE PRETTY QUICKLY.

* SEE PAGE 109.

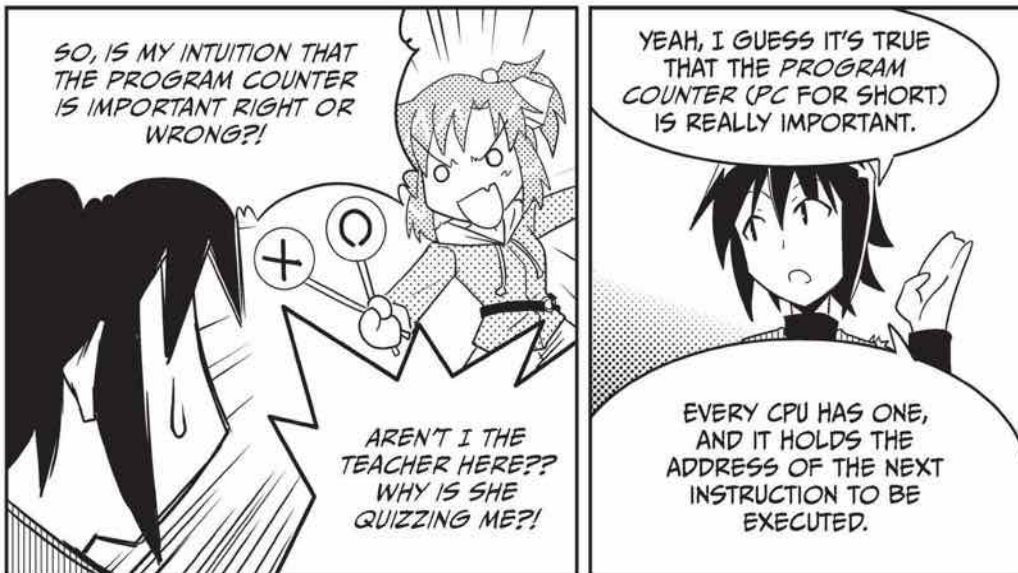
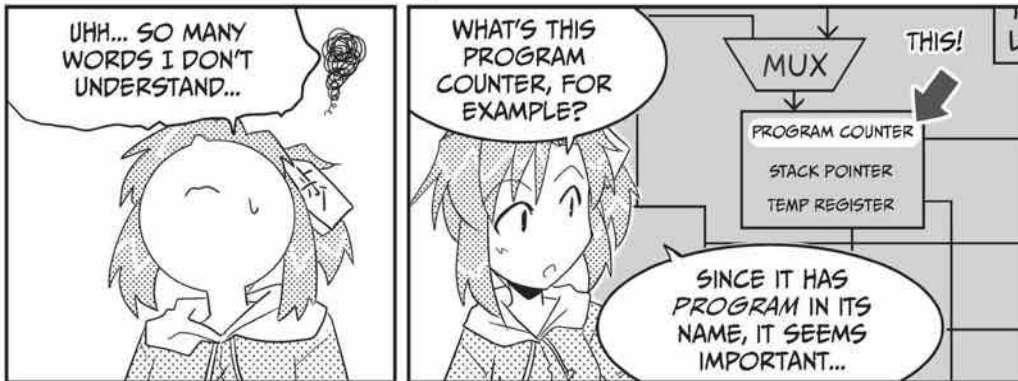
105

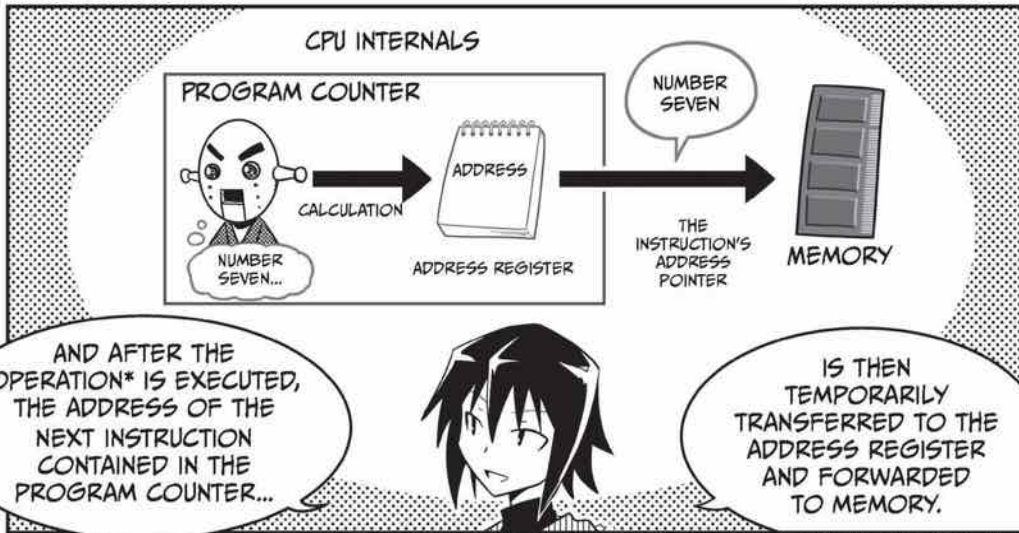
CPU Instruction Processing





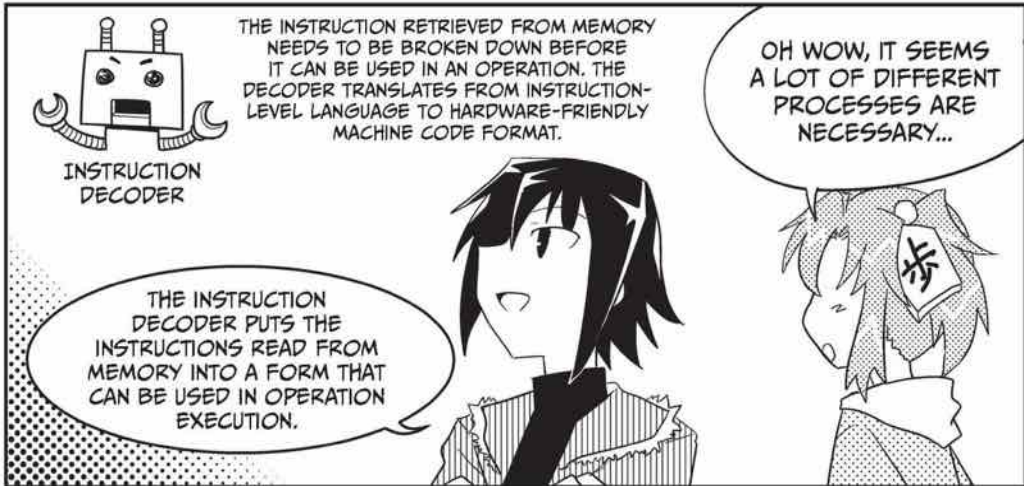
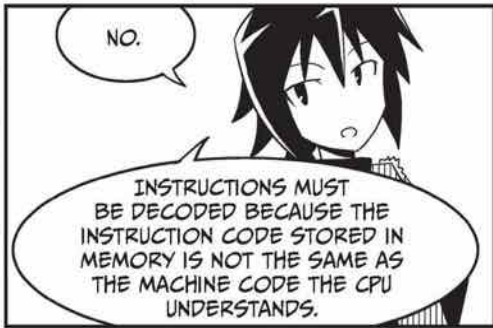
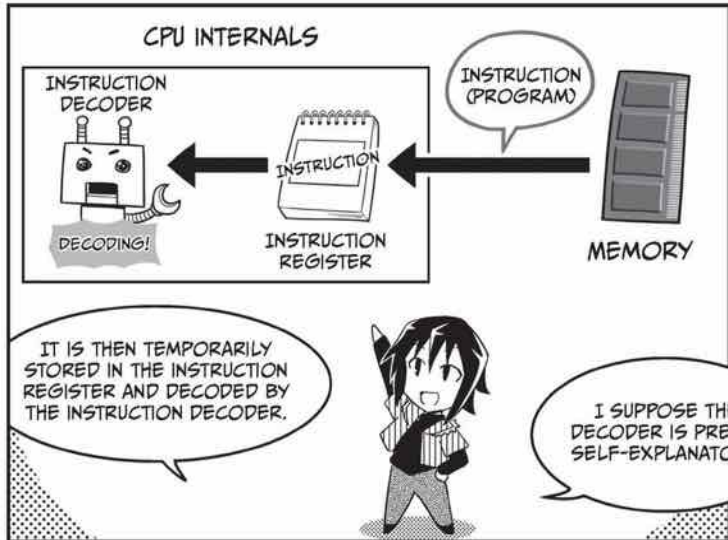
THE INSTRUCTION CYCLE

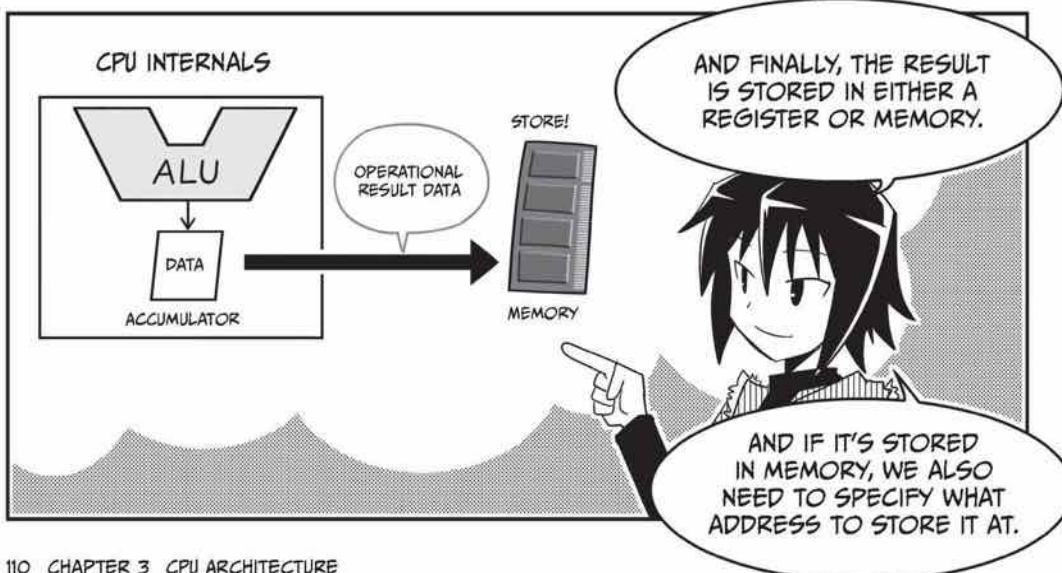




* YOU CAN ALSO PERFORM CALCULATIONS ON MEMORY ADDRESSES.

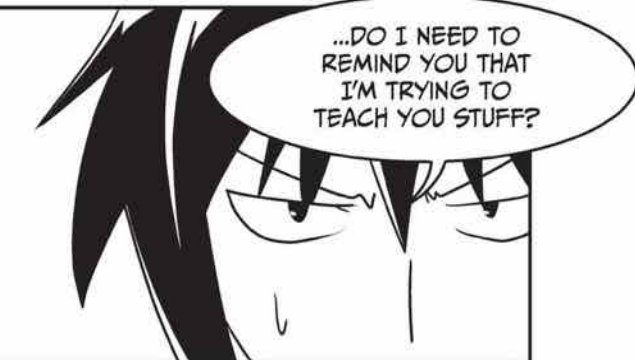
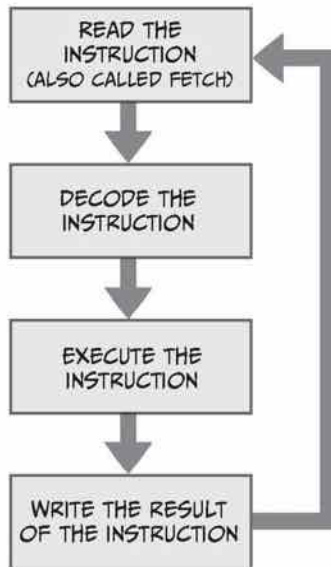






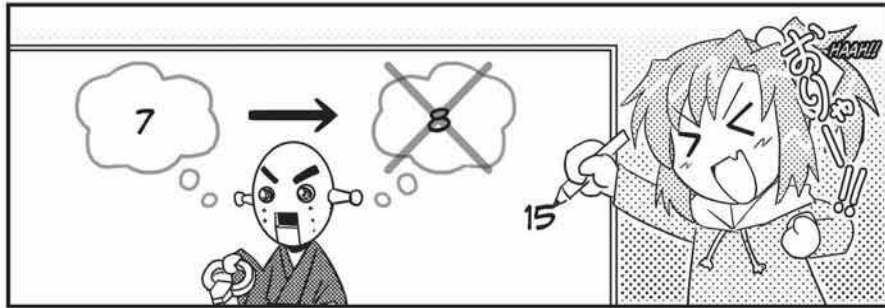


《CPU INSTRUCTION PROCESSING》





THE INSTRUCTION WE PROCESS CHANGES DEPENDING ON THE PROGRAM COUNTER



Hmm, about that CPU overview diagram from before, though (see page 106). . . . There are still a lot of words there I don't know. It all feels a bit hazy now.



Well, there's no rush. Take another look after the next lesson. But for now, let's look at the program counter (PC).

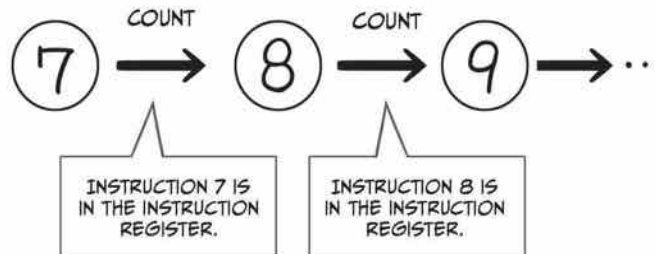


Ooh, that guy who's always one step ahead— no, I mean the guy who remembers the address to the next instruction! That reminds me, didn't we talk about counters before? The circuits that count?

After the seventh instruction is done, we go to the eighth, and then the ninth, and so on. . . . Is that how the saved address keeps changing?



Basically, yes. And by the way, the instruction register gets saved at the same time as the counter changes, like in the image below.





But be careful! Instruction number eight doesn't necessarily follow instruction number seven here.

The program counter stores the address of the instruction to be executed next. After 7, it might jump to number 15 or return to number 3.

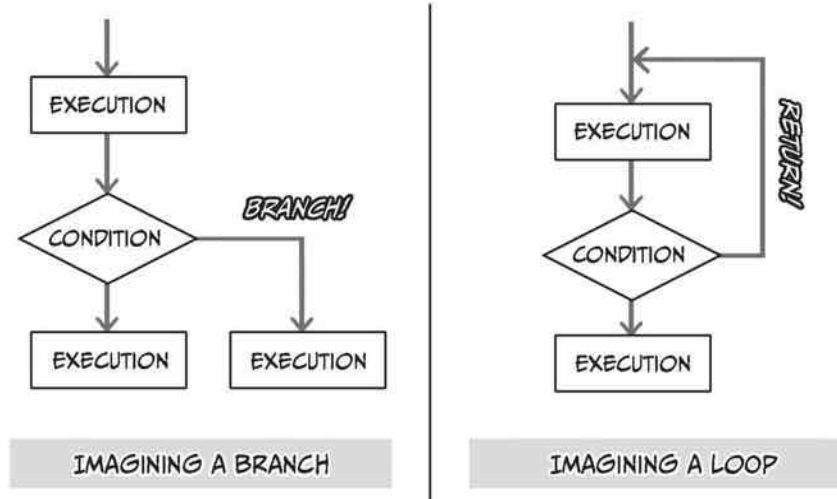


Eeeeeh, why?! Why would the address return and jump around like that?



Hah! This is important, so pay attention. The reason it can jump around like this is that a program might contain conditionals such as *branches* and *loops*!

When the program encounters one of these conditionals, the address of the instruction to be executed next might *jump*. It might be easier to understand this by looking at a diagram.



Ah! It's like the ATM example we talked about a while back! It decided that the balance was insufficient, so the outcome changed. And the ATM might throw you back to the main screen if your PIN code is wrong.



Yeah, the ATM is a great example! And to accomplish these branches and loops, we only have to rewrite the address inside the program counter to the address we want to jump to.



I see. So by rewriting the address in the program counter, we can change which instruction to execute! This way, the program is able to proceed appropriately.



It's also good to know that the bit width of the program counter (the bit width of the addresses in the PC) is the same as both the address bus bit and the address space bit width. If you think calmly about that for a bit, you should realize that they have to have the same bit width.



I see. It seems obvious, but it feels really nice to see the relationship between the different things we've talked about so far!

By the way, the program counter only knows what the next step is, right? Shogi players have to read reeeally far into the future, so maybe the program counter and shogi players are slightly different after all!



Slightly? You're joking, right?!

VIRTUAL MEMORY

Most computer programmers don't have to care about the CPU address bit length of the programs they write for any modern operating system (such as Windows). It is the operating system that decides how it will actually interact with the physical memory, and it exposes this through something called *virtual memory*. The hardware that maps this virtual memory to its physical counterparts is known as the *memory management unit (MMU)*.

All Kinds of Memory Devices


THIS MIGHT BE A BIT SUDDEN, BUT I'D LIKE YOU TO TRY TO REMEMBER...



I SAID SOMETHING LIKE THIS ON THE DAY OF THE CULTURAL FESTIVAL...

MEMORY COMES IN TWO FLAVORS...

PRIMARY MEMORY
WHEN WE SAY "MEMORY," WE GENERALLY MEAN PRIMARY MEMORY.



THERE IS MAIN MEMORY AND SECONDARY MEMORY, BUT WHEN LEARNING ABOUT THE CPU, THE MAIN MEMORY, ALSO CALLED PRIMARY MEMORY, IS A LOT MORE IMPORTANT.

UH, SURE, BUT WHY...?



WELL, IT TURNS OUT THAT SECONDARY MEMORY IS ALSO REALLY IMPORTANT!!



THE MOST REPRESENTATIVE TYPE OF SECONDARY MEMORY IS THE HARD DISK DRIVE (HDD), SOMETIMES JUST CALLED A HARD DISK!! ALMOST EVERY COMPUTER HAS ONE!



CAN YOU PLEASE STOP CHANGING YOUR MIND?!



A COMPARISON BETWEEN HDD AND MEMORY



Umm, I'm shocked by this new information. So what does this small box-looking thingy . . . this hard drive (secondary memory) . . . do?



The easiest way to answer that is to compare it to primary memory. Let's start with the first big difference! When you turn your computer off, everything in primary memory disappears! But any data stored on your hard drive does not.

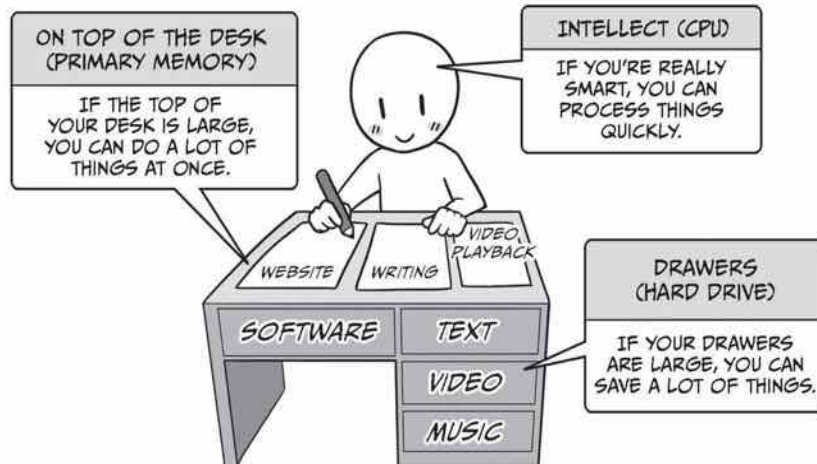
This is why the operating system running your computer (for example Windows), all your programs, and any data you might have created or downloaded (text, video, and so on) are stored on your hard drive.



Eeeeeee!! That's amazing!! But didn't you say that all data and programs used for operations are stored in primary memory?



Yeah. Actually, when you turn on your computer's power, certain parts of the data in your hard drive are copied to primary memory. Anyway, when you turn your computer off, all your data is safely stored on your hard drive. Take a look below.





Let's imagine how the CPU, memory, and hard drive interact. We could say your memory is like the top of your desk and your hard drive is like the drawers of that desk. You should be able to understand their roles better by using this analogy.

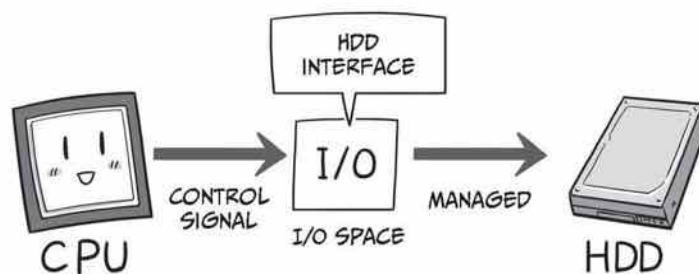


Oooh, they're really different! If primary memory is large, it becomes easier to process large amounts of data at once! And if the hard drive is large, you can save and store a lot of data.



Now, let's talk about the second difference between the two. The CPU can read directly from primary memory but not from the hard drive!

The CPU sends control signals to something called the hard disk interface located in a piece of memory called the *I/O space*. It is this hard disk interface that then controls the hard drive itself.



THE CPU CANNOT ACCESS MEMORY ADDRESSES ON THE HARD DRIVE DIRECTLY!



This might seem counterintuitive since we manipulate the data on the hard drive all the time when using our computers. But really it works like in the picture above.

That is, your CPU only works directly with the address space, which your hard drive is not part of!!

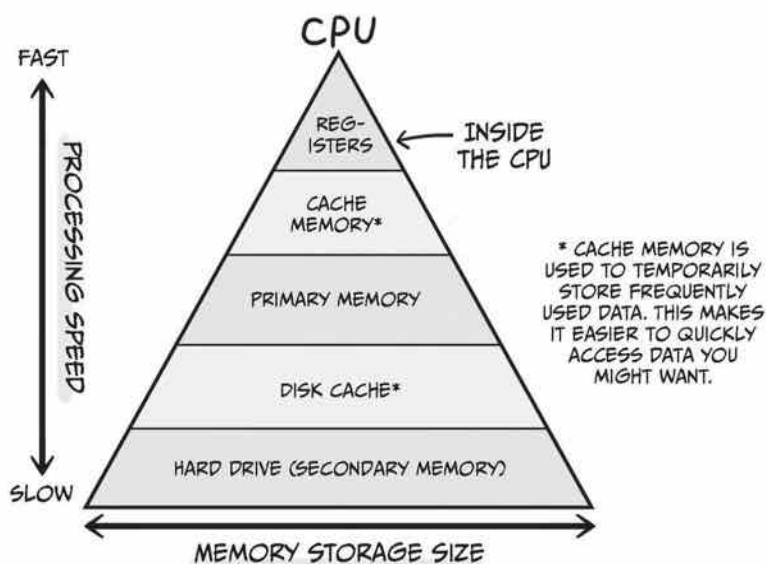


Aha. The only things that can interact with the CPU directly are the primary memory and I/O devices. So that's why you placed so much emphasis on the importance of primary memory. . . .



Then let's talk about the third and final difference: hard drives are a lot slower than primary memory!

There are lots of different types of memory inside any computer, but by comparing each of their relative storage sizes and speeds, you end up with something like this pyramid.



Huh?! So memory close to the CPU is fast but small. As you get further away from the CPU, it gradually grows larger but slower!



That's it. So, for example, registers have fast processing speeds but small memory sizes. A good comparison might be a small, handy notepad.



Yeah. Anyway, I think I understand the difference between primary memory and hard drives now. Even though they're both memory devices, their uses are completely different.

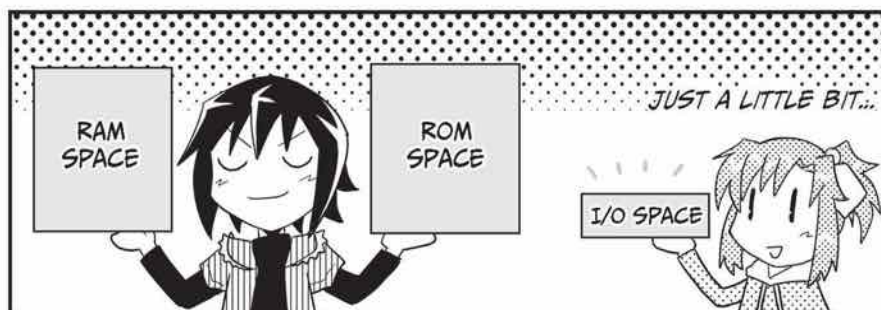


But that's why we can play to their strengths, when appropriate.

An interesting example is that today's computers, especially laptops, have started using solid state drives (SSDs) instead of mechanical hard disk drives (HDDs). SSDs store all data using semiconductor memory technology. This makes SSDs much faster and more resistant to shaking and other types of blunt force than mechanical disks.



RAM SPACE, ROM SPACE, AND I/O SPACE



Okay, let's talk a bit about address space (memory space) again. Do you remember what I taught you before?



Yeah, no problem! It's the dark space ruled by the CPU's iron fist. . . . No, I mean . . . it's the space directly managed by the CPU, right?

TYPE	ADDRESS
0	INSTRUCTION
1	INSTRUCTION
2	INSTRUCTION
3	INSTRUCTION
4	INSTRUCTION
⋮	⋮
30	DATA
31	DATA
32	DATA
33	DATA
⋮	⋮

ADDRESS SPACE
(MEMORY SPACE)



Indeed. But to be more exact, *address space* comprises all the memory space *outside* of the CPU that is controlled by the CPU.



Hmm? All the memory space outside of the CPU? That seems a bit convoluted. Are there other types of memory in there other than primary memory?



Yes. This is important. The address space is actually divided into two parts: *RAM* (memory you can read from and write to) and *ROM* (memory you can only read from). We say that we have RAM space and ROM space inside our memory space.



RAM

YOU CAN BOTH READ FROM AND WRITE TO IT. THE DATA DISAPPEARS IF YOU TURN OFF THE POWER.

EXAMPLE: PRIMARY MEMORY

ROM

YOU CAN ONLY READ FROM IT. THE DATA IS SAVED EVEN IF YOU TURN OFF THE POWER.

EXAMPLE: BIOS-ROM



Huh? What's this about rams and roms?! Okay, so RAM is our old friend the primary memory, right? We can read from and write to it, and its data disappears if you turn off the power. . . .

But what about ROM? So the data is intact even if you turn off the power and you can only read from it, and this is somehow part of the memory space? Umm, what is it, though?!



Yeah. We haven't really talked about it yet, but there is ROM on something called the motherboard inside the computer. This is where you can find the program the CPU runs when you start your computer. This program that runs before any others is called the *BIOS*.



I see. So if it couldn't run this program, the computer would just be a very expensive box? That's why the BIOS is put into a special part of read-only memory—so it won't be forgotten!

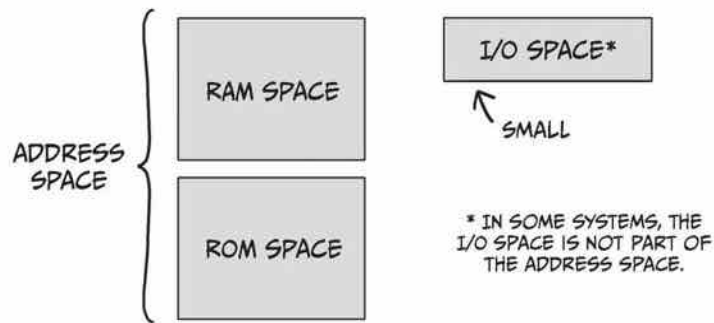
WHAT IS THE BIOS?

The *BIOS* (*Basic Input/Output System*) is a program found in ROM that the computer runs when you first turn it on. The BIOS checks that all the devices in your computer are in working order after you turn on the power. It also launches your operating system from your hard disk.





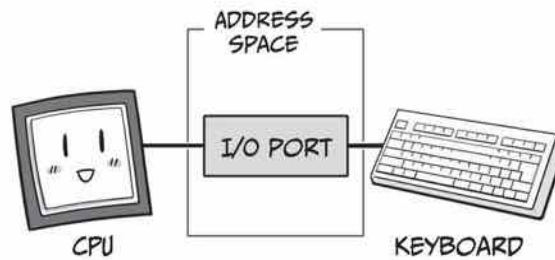
In addition to the RAM space and ROM space, there is also a very tiny space called the I/O space.



I think I remember hearing you mention I/O earlier today. Ice cream and oreos... no... input/output, right?



Yeah. The I/O ports live inside this I/O space. As I explained before, the CPU uses these I/O ports to talk to external devices directly (such as keyboards and mice). This is why your computer responds when you press a key on the keyboard.

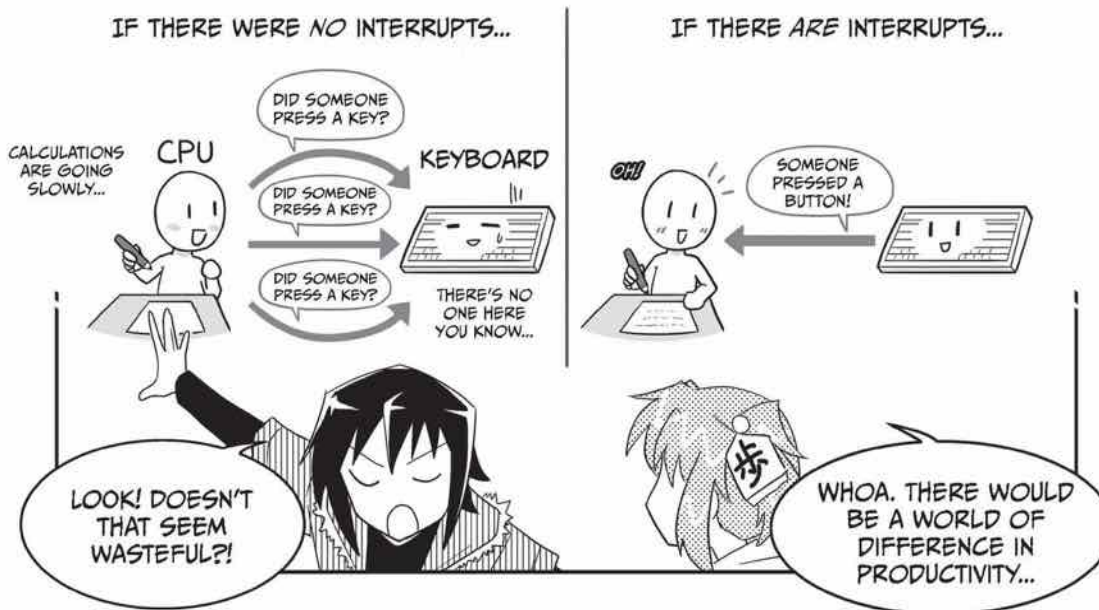
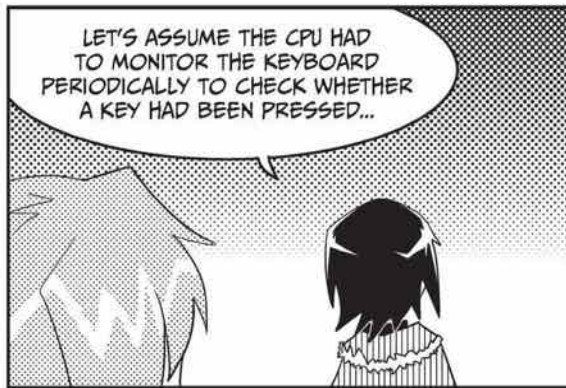
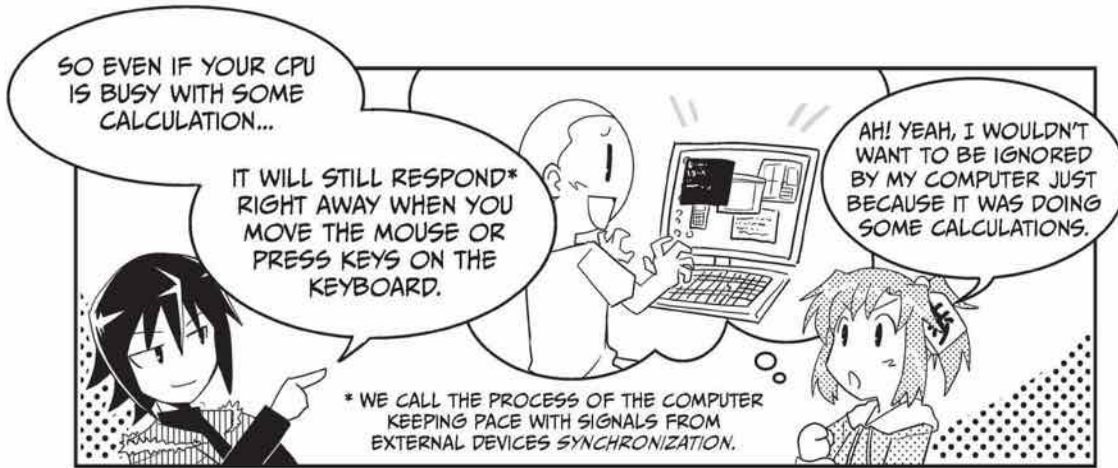


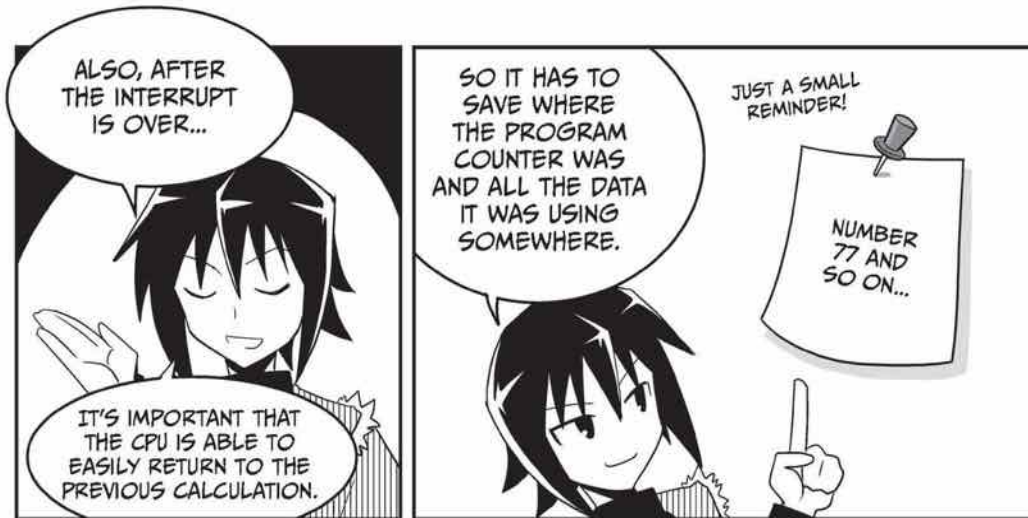
Hoho! Go CPU! So that means that since external devices use the address space managed by the CPU, they are also managed by the CPU, right? In any case, I think I get that there are different types of spaces inside the address space!

What Are Interrupts?









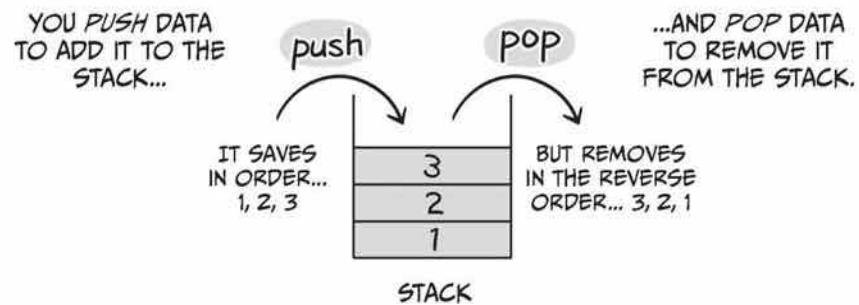


THE STACK AND THE STACK POINTER



Okay, let's get right into it. As I said, to be able to return to the task it was doing before the interrupt, the computer needs to take some memos before it starts a new task.

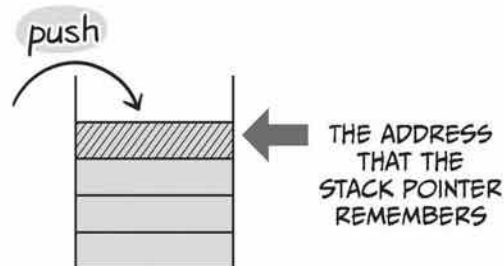
It uses something called the *stack*—a part of main memory reserved for bookkeeping—to do this. The way it does this is pretty interesting—take a look.



Ooh, that's a funny way to remember things! It's kind of like a stack of books that you can keep piling up, but if you want to take one out, you always have to take one from the top. You can't just take data from anywhere.



That's exactly right. And a special register holds something called the *stack pointer* (*SP* for short) that points to the last stack address we worked with.

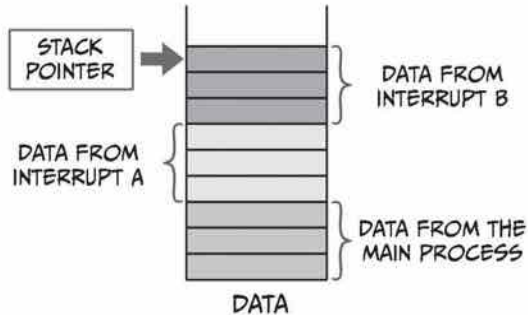


I see. While the program counter keeps track of the address for the next instruction, the stack pointer keeps track of the last address on the stack.



When using stacks, it's important to use the stack pointer correctly. Because . . .

With just one interrupt, everything is fine. But if you keep adding interrupts one after another, the stack will keep growing and eventually bad stuff will happen. . . .



The data that is saved during an interrupt consists of the accumulator, status registers, and the program counter.



Whoaaa! I'm not sure I'm getting what's happening there.



Bugs or even just frequent interrupts that the stack has no coping mechanisms for can result in the stack pointer losing control over the program's execution.

This usually means that the person or people who wrote the software didn't properly understand the implications of using a stack...



Ah! Have you ever made one of your programs go out of control like this?



When working with the CPU, interrupts can be really efficient. But if you don't know how to work with the stack, you're bound to make a program that runs into problems like this. And that's all I have to say about that!!



So you were also like that once, right? You didn't know how to work with the stack and something happened right? I'm right, aren't I?!



Fuhahahah! I'm just talking hypothetically!



INTERRUPT PRIORITY



Ahem. Let's try to refocus and talk a bit about *interrupt priority*.

Let's once again assume that you were interrupted with a phone call while cooking. Now let's assume that you also hear the doorbell while on the phone. What would you do?



Eeeeh!? That's just bad timing! I don't think I could deal with that. I'd rather people would stop interrupting me all the time. . . .



Heh heh heh. Yeah, I thought so. This is when *interrupt masks* are really useful! By using an interrupt mask, you can avoid being interrupted at all. You know, like how you can hide your face behind a mask.



Yeah, wearing a mask can stop all kinds of things!



But you can't let your guard down. You can still be forcefully interrupted by some things, even if you're using a mask. We call these *resets*!

Resets are the interrupts with the absolute highest priority. They are special interrupts that are not affected by masks.



Resets! The word does have a kind of "don't argue with me" air about it. Just like when you press the reset button on your game console, it returns to its initial state, right? It really gives me this "let's start over" vibe.



Yeah, that's right. And whether it's your computer or your console, both of them start at their initial state when you turn on the power, right? That's because when you turn on the power, the first thing the system does is issue a reset.

Resets return programs to their initial state. Put another way, they return all the circuits to their initial state. Completely. This is why when we want the computer to wake up nicely—that is, when we want it to start in a functional state—we have to issue a reset.



Oooh, it felt a bit scary when you said it was forceful, but I guess a bit of force is necessary.



There are also some interrupts of the highest priority that the CPU will not mask even though these interrupts might not be resets.

We call these interrupt signals that go through masks *non-maskable interrupts (NMI)*. They can have many uses depending on the system architecture, and they're really convenient in some cases.



It seems interrupts have all sorts of uses.



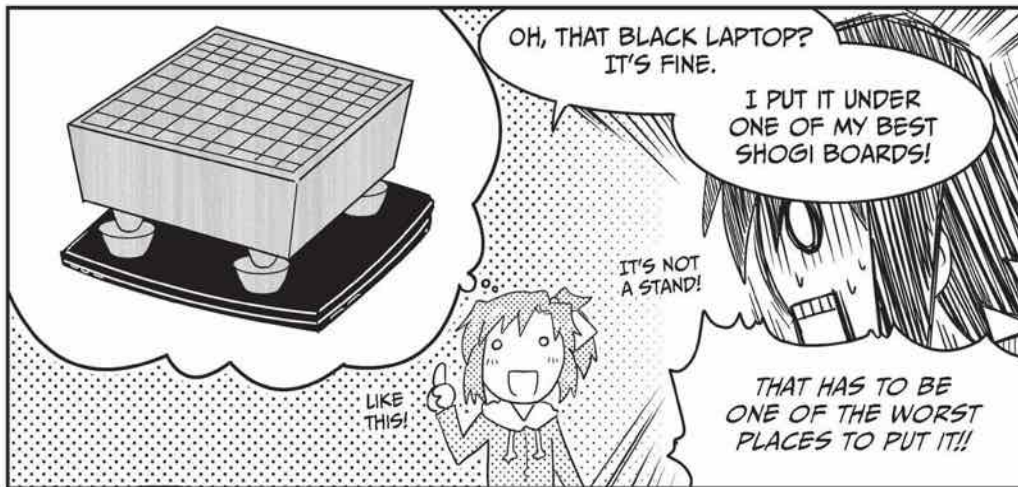
There are also *timer interrupts* that issue an interrupt when they reach zero after counting down using a decrementing counter. (Think 3, 2, 1, interrupt now!) Using these, it's possible to execute programs at regularly timed intervals.



Ah! That timer interrupt gave me an idea! There is a program that runs every day at 7 AM that sounds a bell when I'm executing my sleep task. It's issuing an interrupt right when I'm snoozing away peacefully!!



Ah. That's just your alarm clock.

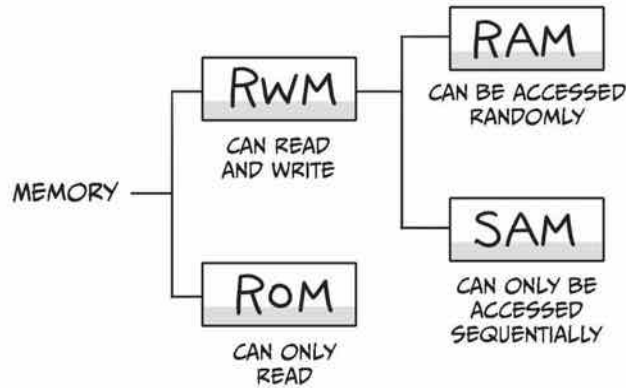




MEMORY CLASSIFICATIONS

ROM stands for read-only memory and is a type of memory that will not lose its data even if the power is turned off. As the name implies, you can only read from ROM. You can't write to it.

In contrast, with RAM, which stands for random access memory, you can read from or write to any address in any order. You might think that ROM and RAM are opposites, but that isn't necessarily the case.



As you can see in the image above, the opposite of RAM is actually *SAM* (*sequential access memory*), which was an older type of memory commonly found on magnetic tapes and drums. As the name suggests, it could only read memory addresses in order. In addition, the opposite of ROM is the now defunct *RWM* (*read-write memory*).

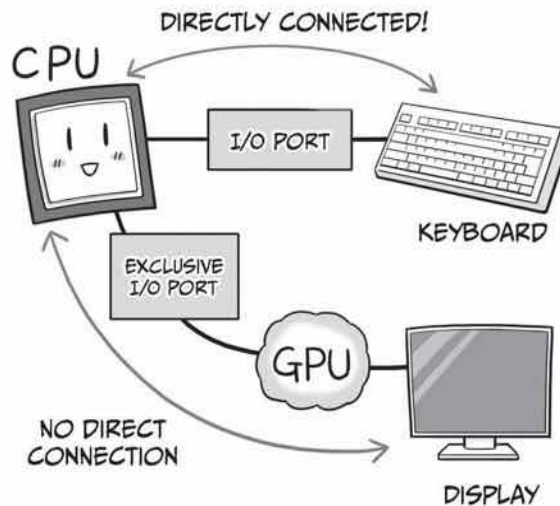
Memory that retains its data even when the power is turned off and allows this data to be accessed again when the power comes back on is called *non-volatile memory*. Memory that loses its data when the power is turned off is called *volatile memory*.

These terms are no longer commonly used, however, and have largely been replaced by RAM (instead of volatile memory) and ROM (instead of non-volatile memory).

I/O PORTS AND THE GPU

If there were no connection between input/output devices and the CPU's registers or ALU, the CPU would be unable to accept external input. External input doesn't only come in the form of character input from the keyboard; it can be a mouse click or any electrical signal. If we didn't have some output, such as LEDs that light up when an operation is complete or some other signal, it would be very hard for us to interact with any computer. In the same way we need feedback, the internal data bus needs *input and output ports* (*I/O ports*) to communicate with external devices such as memory and so on.

The most commonly used output device is the computer display. This is an example of a device that is not connected directly to the CPU. The display is instead connected to a special IC called the *GPU* (*graphics processing unit*), which generates and outputs images on demand. When the CPU needs to use the GPU, it has a special I/O port dedicated to GPU communication.



Smaller systems sometimes don't have a GPU but are still attached to a color LCD (liquid crystal display). In these cases, the CPU communicates by sending any necessary data through an I/O port to an LCD controller. This LCD controller and its driver then output the image data to the display.

CLOCK FREQUENCY AND DEGREES OF ACCURACY

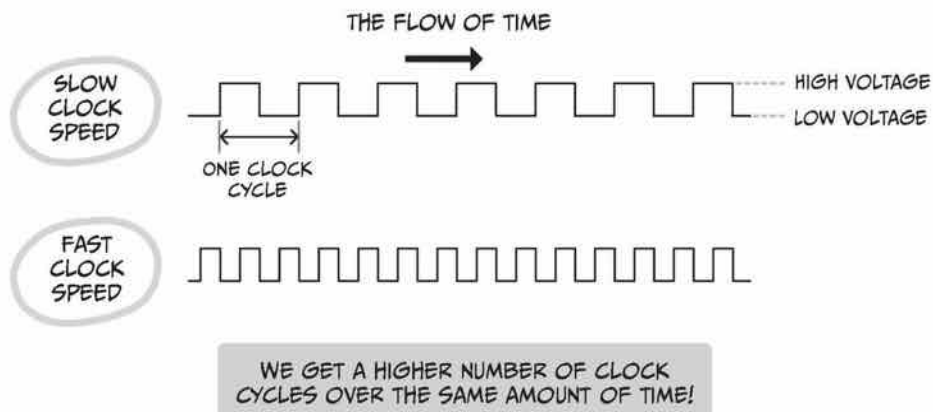
Of course, you need electricity for the CPU to work. But you also need a clock frequency. A *clock* is a signal that alternates between high and low voltage at regular intervals. The *clock frequency* is how many times this signal alternates in one second.

The clock is like the heartbeat of the CPU. It is essential for updating the CPU's internal circuits, such as the latching of the data inside the ALU and the block^{*} advancing the program counter.

Clock frequency is measured in Hz (hertz), which is a measure of how many times the clock cycles in one second. So, a clock running at 40 MHz would be cycling 40 million times per second.

This *clock speed* is also a measure of the performance of the CPU. Everything that the CPU does, like instruction decoding and ALU operations, it does in synchronization with the clock. The CPU can execute one action per clock cycle, so the higher the clock frequency, the higher the clock speed and the faster the execution speed of the CPU.

* *Block* is a term used to denote the group of things needed to realize some function.



The degree to which the clock speed matches the clock frequency is called the *degree of accuracy*. When using computers for communication applications, connecting two devices with clock frequencies that do not match can cause timing problems.

CLOCK GENERATORS

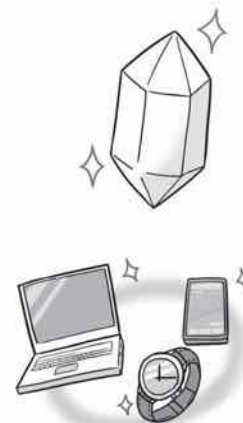
We call circuits that generate clock signals *clock generators*. Most CPUs have internal clock generators, but it is also possible to connect externally generated clock signals to a CPU. The different components inside the CPU that make up the clock generator—including the crystal oscillator, capacitors, and resistors—all contribute to the accuracy of a clock signal's frequency. Some situations don't require high accuracy, but if a CPU must be synchronized with other devices to exchange data, for example, then the accuracy of the clock signal's frequency is a high priority.

WHAT ARE CRYSTAL OSCILLATORS?

Crystal oscillators are made from small artificial crystal shards that have been cut incredibly thin. If you attach two electrodes to a shard and apply a voltage, the crystal warps. By fluctuating the direction of the voltage, it is possible to create vibrations that give rise to a stable frequency. Consequently, you can generate oscillations at very precise time intervals.

Crystal oscillators are used in many kinds of devices in which precise time intervals need to be measured, such as computers, phones, and watches. The quartz in a quartz watch is actually a crystal oscillator.

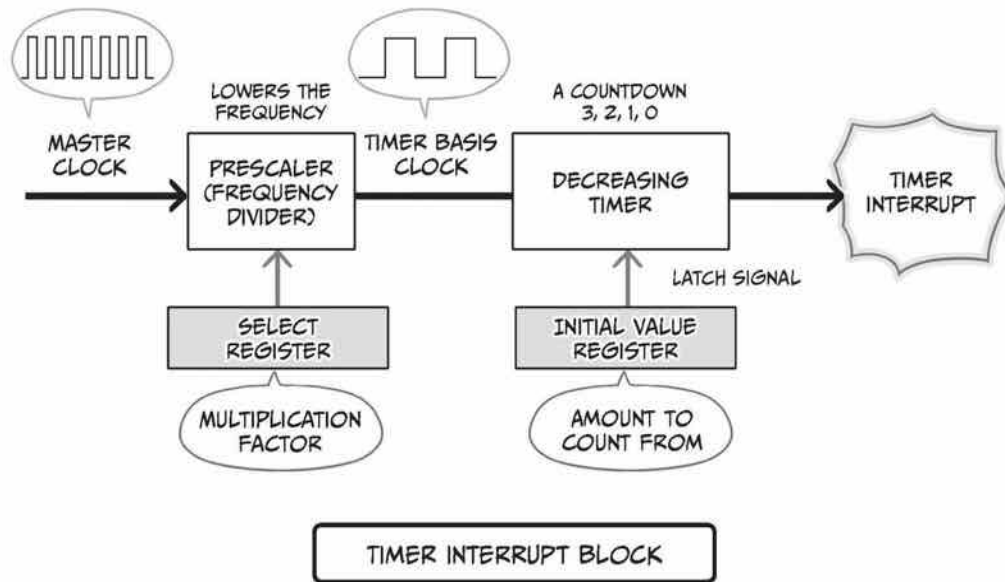
By connecting clock generators to crystal oscillators and *condensers* (electronic components that store and release electrical charge), it's possible to create an alternating signal.



To achieve a high degree of accuracy, you can use an external clock signal instead of the clock signal from the CPU's internal clock generator. External oscillators usually provide higher quality clock signals than internal clock generators.

TIMER INTERRUPTS

By using the decrementing counter inside CPUs, we can initiate interrupts whenever the timer reaches zero. We call this a *timer interrupt*.



It is also possible to use the CPU's base clock (or master clock) with a *frequency divider*.^{*} Dividing the CPU's base clock with a frequency divider allows you to increase the time required for a countdown timer to count to zero. Indeed, you can change the amount of time required for the timer to count down to zero from several seconds to several hundred seconds.

It is then possible to execute some program at given intervals by setting the initial value of the countdown timer to some value (for example, 100). To change the interrupt frequency, all you need to do is tell the CPU to rewrite the register where the "value to count down from" is stored. Changing this value from 100 to 50, for example, would double the interrupt frequency.

You can set a countdown timer to run even while other programs are running, and it will issue an interrupt when it has counted down to zero. There are many uses for this. For example, you can turn a light on and off at fixed intervals. Timer interrupts are more effective than other methods for doing this because they save valuable CPU time.

^{*} *Frequency dividers* change the period by lowering the frequency.

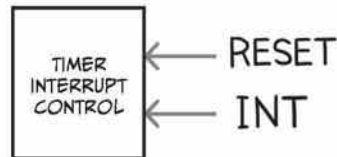
HOW TO USE TIMER INTERRUPTS

To use a timer interrupt, you must first configure it by writing a value into its control register. The value written to this register determines the clock source, whether the clock frequency is to be divided before counting and by how much, and other timer behavior.

Next, we write the initial value into the counter and set the timer to start on the reset signal. After it has started, the timer will interrupt the CPU every time it counts down to zero.

We then rig the timer to start on the reset signal (see the next page) and to cancel if commanded by the CPU to do so. After it has started, an *interrupt signal* will be sent from the timer block to the CPU control circuit every (master clock cycle) \times (multiplication factor) \times (value to count from) units of time.

Finally, let's examine the timer interrupt component present in classic CPU architectures, which can be seen in the image below. *INT* here is the signal that the CPU uses to send instructions to the timer interrupt block. *RESET* (timer reset) is the signal used to start the timer.



If you were to leave the reset input in an active state, the timer would stop and never start. If you then were to clear the timer reset, it would start counting down again and eventually issue an interrupt. After this, it would count down from the set value on every cycle of the multiplied master clock frequency, issuing an interrupt signal when it reached zero.

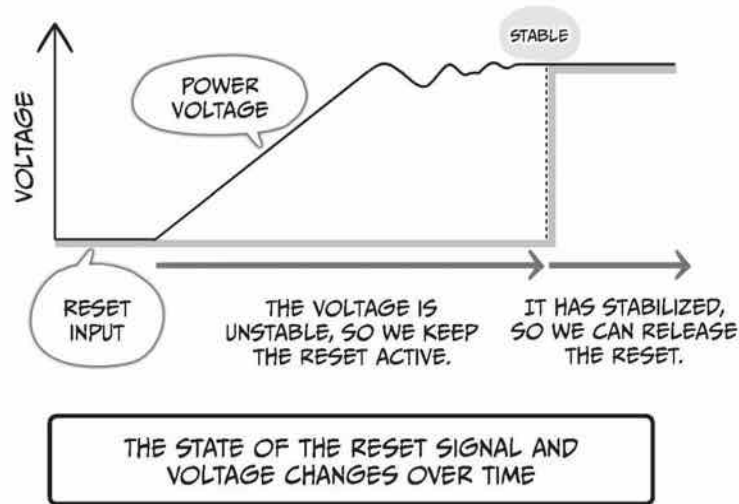
When it reached zero, it would once again latch to the value stored in the "value to count from" register and start over. By doing this over and over again, the component is able to produce interrupts at fixed intervals indefinitely.

RESET SIGNALS

To *reset* means to set programs and internal circuits to their initial state. The program counter is set to zero, and temporary operational data is cleared. A reset signal is issued every time you start your computer. This is extremely important as it makes sure that any programs you run after the start-up process work correctly.

Let's take a closer look at the reset process. The reset signal is raised by setting an active state after a low voltage state. After you turn on the power, the voltage will fluctuate a bit before finally settling down at a stable level. If the CPU were active during this period, all kinds of problems would result. This is why the reset signal is constantly active

during this period, making the CPU unable to process anything. In other words, we protect the CPU by maintaining the reset state until the voltage has stabilized. Then, when the voltage has stabilized, we release the reset signal by raising the voltage.



If, for example, the CPU were to start acting in an unexpected manner, it is possible to initiate a forced reset by lowering the voltage below the necessary level (and therefore enabling the reset) and setting all programs and circuits to their initial state. Resets are an essential function needed to ensure that the computer will work as we expect it to.

CPU PERFORMANCE IS MEASURED IN FLOPS

CPU performance is determined by the CPU clock speed and its operation execution speed. The clock speed tells us how often the logic circuits in the ALU can perform calculations. And the *operation execution speed* tells us how quickly the CPU can perform calculations one after another.

Older CPU ALU blocks worked only with integer arithmetic. Back then, the CPU's performance was measured by how many instructions it could handle in one second, or its *MIPS* (*million instructions per second*) value, rather than by how quickly it could perform calculation operations. As its name suggests, MIPS indicated how many millions of instructions the CPU could handle in one second.

These older CPUs were, of course, also able to work with floating-point values, but modern CPUs have specialized integrated hardware for just this purpose. This is why in more recent years the preferred measure of performance has become how many floating-point operations the CPU can handle in one second, or *MFLOPS* (*million floating-point operations per second*). Once again, as its name suggests, this value indicates how many millions of floating-point operations with 15 significant digits the CPU can handle in one second.

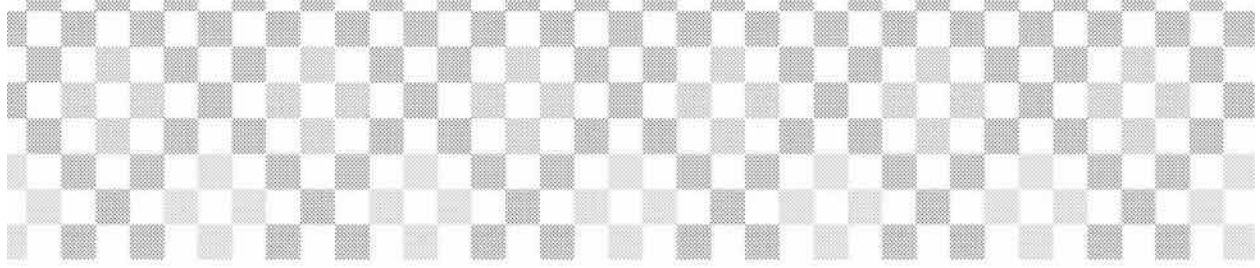
We sometimes use units other than MFLOPS, such as *GFLOPS* (*gigaFLOPS*) and *TFLOPS* (*teraFLOPS*). One GFLOPS is the processing of a billion floating-point operations with 15 significant digits in one second. One TFLOPS is the ability to process a whopping trillion floating-point operations with 15 significant digits in one second.

YOU CAN SEE THE
PERFORMANCE OF A
CPU BY LOOKING AT
ITS FLOPS VALUE!



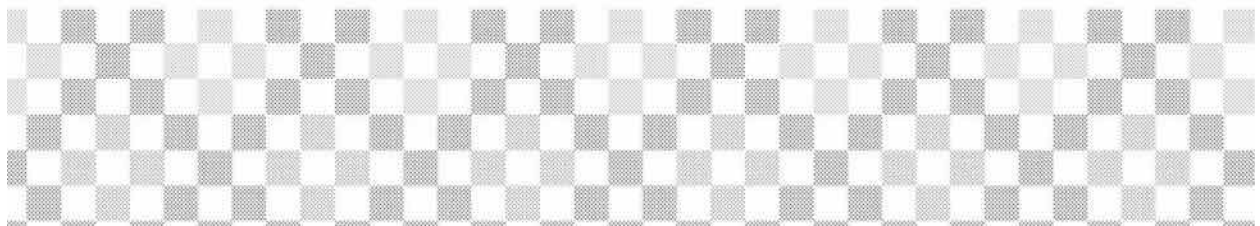
WAAH!

I'M SO
EMBARRASSED...



4

OPERATIONS



Types of Operations



AAH... YUU IS IN MY HOUSE...



WELCOME!
PLEASE
COME IN.

THANK YOU.



WHAT AM I SAYING?! WHY
AM I EVEN NERVOUS?!!

HMM...



EVERYTHING SEEMS
PRETTY CLEAN. I WAS
EXPECTING IT TO BE A
LOT MESSIER...



I WAS HOPING...

WHAT ARE YOU
UPSET ABOUT?!



THERE ARE MANY TYPES OF INSTRUCTIONS

OKAY, TODAY WE'RE GOING TO TALK ABOUT INSTRUCTIONS.



OH, I REMEMBER WE TALKED ABOUT INSTRUCTIONS BEFORE. THESE ONES, RIGHT?



DO SOMETHING...
INSTRUCTION CODE (OPCODE)
...TO SOMETHING
OPERAND

PROGRAMMING INSTRUCTIONS ARE REALLY JUST STRINGS OF 1s AND 0s, SOMETIMES CALLED MACHINE LANGUAGE.

I UNDERSTAND THIS!



0000000100000101

OPCODE
(INSTRUCTION)

OPERAND
(THE VALUE OR ADDRESS TO USE)



DEPENDING ON THE TYPE OF INSTRUCTION, THE LENGTH OF THE INSTRUCTION (HOW MANY BYTES LONG IT IS) AND THE NUMBER OF OPERANDS MIGHT CHANGE.

CPUs ONLY UNDERSTAND MACHINE LANGUAGE.

TYPES OF OPCODES

COMPARE

STORE

JUMP

UH-HUH...



SO THERE ARE LOTS OF DIFFERENT KINDS OF INSTRUCTIONS, RIGHT?

THAT'S RIGHT! I'VE CATEGORIZED DIFFERENT TYPES OF INSTRUCTIONS HERE IN THIS TABLE.

INSTRUCTION TYPES	
INSTRUCTIONS THAT DEAL WITH CALCULATIONS	INSTRUCTIONS THAT DON'T DEAL WITH CALCULATIONS
1. Arithmetic instructions	1. Data transfer instructions
2. Logic instructions	2. Input and output instructions
3. Bit shift instructions	3. Branching instructions
	4. Conditionals (comparison instructions and so on)



I'LL BE GOING THROUGH THESE IN ORDER TODAY.



WOW, THERE ARE THAT MANY...?



WELL, YOU'VE ALREADY LEARNED A LOT, SO I WOULDN'T BE THAT WORRIED.

IF YOU UNDERSTAND THESE INSTRUCTIONS, THEN YOU'LL KNOW WHAT IS HAPPENING INSIDE THE CPU.



I SEE... THEN GIVE ME A DETAILED EXPLANATION OF ALL THESE INSTRUCTIONS IN UNDER THREE SECONDS!!

DON'T GIVE ME IMPOSSIBLE INSTRUCTIONS LIKE THAT!!!

INSTRUCTIONS FOR ARITHMETIC
AND LOGIC OPERATIONS

LET'S START WITH
THESE TWO.

ARITHMETIC
OPERATIONS

PLUS (ADDITION)

MINUS
(SUBTRACTION)

LOGIC
OPERATIONS

AND (LOGIC INTERSECTION)

OR (LOGIC UNION)

NOT (LOGIC NEGATION)

DO YOU
UNDERSTAND WHAT I MEAN
BY ARITHMETIC OPERATION
AND LOGIC OPERATION
INSTRUCTIONS?

THINGS LIKE ADDITION
ARE ARITHMETIC, AND
THINGS LIKE "AND" ARE
LOGIC, RIGHT?

SO, IT'S BASICALLY WHAT
TYPE OF INSTRUCTION
THEY ARE!

YEAH, BUT TO GET A DEEP
UNDERSTANDING OF THESE
THINGS, WE REALLY NEED TO
LOOK AT WHAT'S HAPPENING
INSIDE THE ALU...

ALU

BUT LET'S LEAVE THAT
PIECE OF FUN FOR LATER
AND CONTINUE WITH THE
OTHER INSTRUCTIONS.

WHAT ARE BIT SHIFTS?

SO, NEXT UP ARE BIT SHIFT OPERATIONS?

WELL, I GUESS IT HAS SOMETHING TO DO WITH MOVING BITS, BUT OTHER THAN THAT...

YEAH, THAT'S RIGHT. LOOK AT THE NEXT FIGURE.

LOGICAL RIGHT SHIFT (USING TWO BITS)

REMOVE THE BITS ON THE RIGHT...

0 1 1 0 0 1 0 0

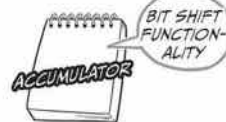
0 0 0 1 1 0 0 1

AND ADD 0s TO THE LEFT.

WE MOVE THE REST OF THE BITS TWO PLACES TO THE RIGHT!

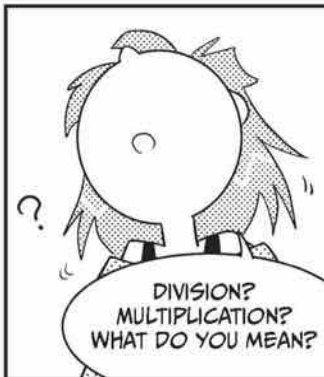
OOH!! JUST LIKE YOU SAID, THEY ALL MOVED! THEY WERE SHIFTED TOGETHER.

THIS OPERATION IS PERFORMED IN THE ACCUMULATOR, THE REGISTER WHERE OPERATIONAL RESULTS ARE TEMPORARILY STORED.



(BIT SHIFT FUNCTIONALITY RESIDES IN THE ACCUMULATOR.)

AS YOU CAN SEE, BIT SHIFTING MOVES THE BITS LEFT OR RIGHT ALL AT ONCE!



THE RESULT IS
ACTUALLY EQUAL TO 100/4
(100 DIVIDED BY 2²) OF
THE ORIGINAL VALUE!

(BINARY)

0	1	1	0	0	1	0	0	...	100	(DECIMAL)
↓ (100/4)										
0	0	0	1	1	0	0	1	...	25	(DECIMAL)

RIGHT SHIFT BY
TWO BITS

RIGHT SHIFTING A BINARY NUMBER
BY N BITS IS EQUAL TO DIVIDING
THAT NUMBER BY 2^N.

LEFT SHIFTING A BINARY NUMBER
BY N BITS IS EQUAL TO MULTIPLYING
THAT NUMBER BY 2^N.

THIS REALLY IS
USEFUL! BUT THIS IS
ONLY POSSIBLE IN
BINARY, RIGHT?



THE SIGN BIT LETS US EXPRESS NEGATIVE BINARY NUMBERS



Before I explain bit shifts more, I want to talk briefly about *sign bits*.

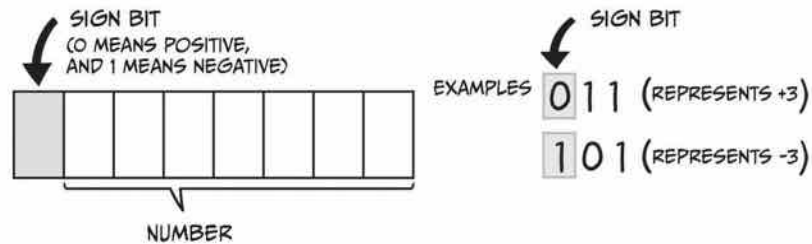


Sign bits . . . ? What are those?



In a binary number, the sign bit is the bit to the far left, and it tells us if the number is positive or negative. If the left digit is 0, the number is positive, and if it's 1, the number is negative.

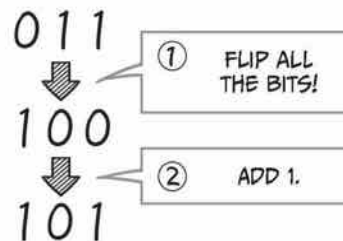
Look at the image below. The most significant bit, which is the leftmost bit, is the sign bit. The sign bit, along with the rest of the bits, determines what numerical value is being represented.



Uhh . . . I can see how 011 would make +3 just by thinking in simple binary. But why does 101 equal -3? That doesn't make any sense.



Remember complements? When expressing negative numbers in binary, we use the *two's complement*.



Oh, now I see it! So to express the negative value of 3 (011), we get 101. With the important part being the *sign bit to the far left*.



Yes. Using three bits without a bit sign, we could express eight different numbers from 0 to 7, but using three bits including one sign bit, the range changes to -4 to 3. We still get eight numbers though, as shown in the table below.

NUMBER	TWO'S COMPLEMENT
3	0 1 1
2	0 1 0
1	0 0 1
0	0 0 0
-1	1 1 1
-2	1 1 0
-3	1 0 1
-4	1 0 0

↑
SIGN BIT

SIGNED THREE-BIT NUMBERS



Doesn't that mean that if I have some binary number—say 101—I could interpret that number as -3 if I assumed it was signed or as 5 if I assumed it was not signed?

They look the same, but the expressed values are completely different. . . . That's just confusing, don't you think?! What were they thinking??

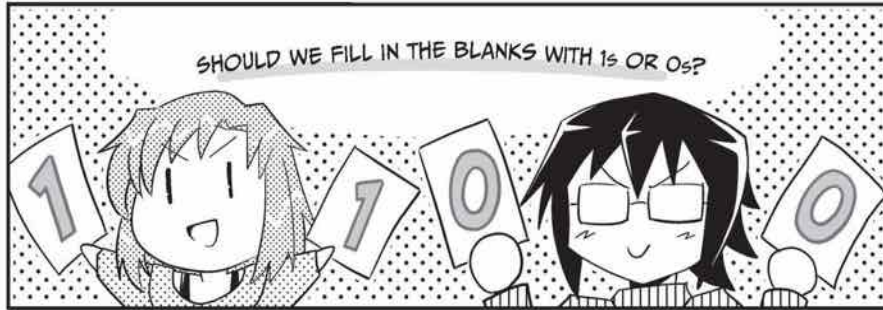


Ah, it's true that humans wouldn't be able to tell the difference. Computers, however, have dedicated systems that keep track of this.* So don't worry about it!

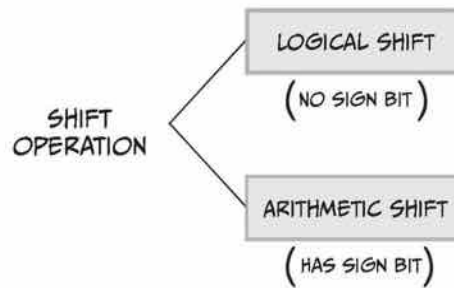
* Programs have a flag that changes depending on the calculation's result to track changes to the sign. If the program monitors this flag, it's possible to tell whether any unforeseen changes occur to the sign of a number. Not all CPUs support this feature, though, and if the CPU doesn't, it's up to the program to keep track of the sign bit.



LOGICAL SHIFTS AND ARITHMETIC SHIFTS



Now let's return to bit shifts. There are two types, *logical shifts* and *arithmetic shifts*. Essentially, the difference is whether we are using the sign bit we talked about before.



Oho! So logical shifts don't have sign bits, but the arithmetic shifts do. I see where this is going.

The outcome of a logic operation is always either true or false, right? That means that concepts like sign bits or negative numbers wouldn't exist. But since arithmetic operations deal with adding and subtracting numbers, those concepts would be necessary.

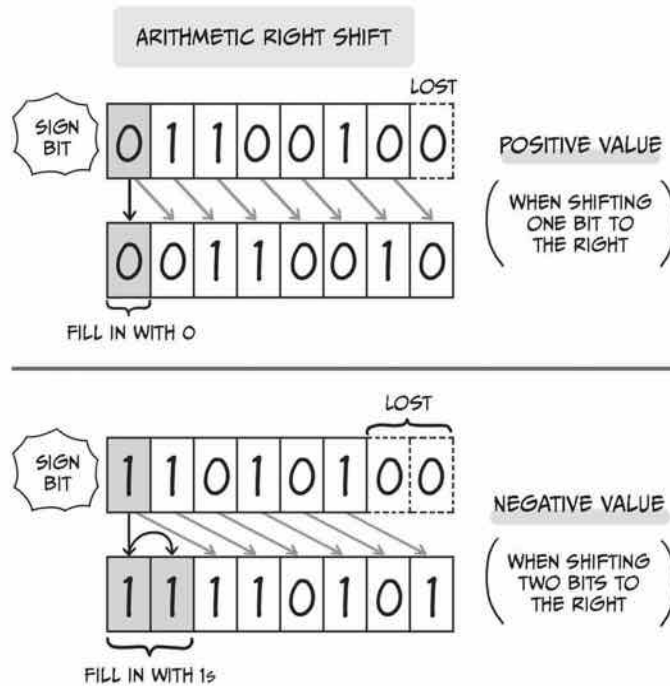


Mm, yes! That is an astute observation—you are correct.

Logical shifts are very simple, and we've already talked about them. Arithmetic shifts, on the other hand, are a bit tricky.



Look at the next figure. When performing arithmetic shifts, we fill in the blank spaces with 1s if the sign bit is 1 and with 0s if the sign bit is 0. You have to pay attention to the sign bit, essentially.



Ohh! With logical shifts, you could just fill the blank spaces with 0s without a second thought, but with arithmetic shifts, you have to keep the sign bit in mind.



There is another really important thing, though. Please look at the next image. We shift a positive number (the sign bit is 0) to the left and . . . whoa! A 1 from the number value bits might end up becoming the most significant bit.



Oh my . . . that can't be good. It would look like the number turned negative all of a sudden (since the sign bit is 1).

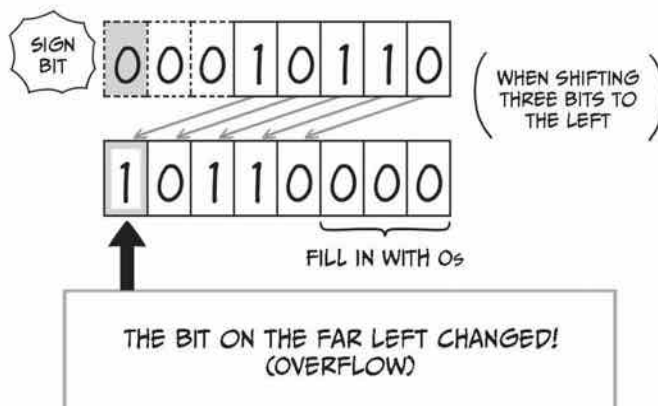


Yeah. While the operation was only supposed to multiply the value 2^n , it ended up flipping the sign bit instead. We call this *overflow*, just like how water can overflow from a cup if you pour too much in. When this happens, it means that the calculation result used more digits than it was allowed and therefore "overflowed."



ARITHMETIC LEFT SHIFT

TAKE NOTE! NOT ALL CPUs ARE GUARANTEED TO HAVE THIS FEATURE.



A serious state of affairs, for sure! I guess this is an error? It's not like you can pretend it didn't happen . . . and you couldn't continue the calculation like this.



Mhm. When overflow occurs, the *overflow flag* (*overflow bit*) of the status register is set. It makes sure to remember that an overflow occurred as a result of a calculation.



Hah! So another register is in charge of taking notes about any grave errors that might occur in the accumulator register. This way they won't be missed!

OVERFLOW AND UNDERFLOW

Calculations using floating-point numbers (as opposed to the integer operations we've been talking about) can both overflow and underflow if the algorithm (the method of calculation) used produces a result that falls outside of the acceptable range of values.

For example, if the result of some calculation is a value that is so close to zero that it cannot accurately be expressed using the available bits (think 0.0000000000 . . . 1), it would generate an underflow.

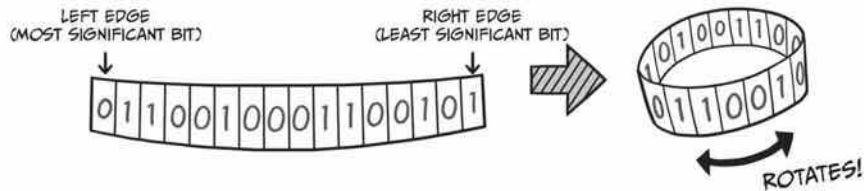


CIRCULAR SHIFTS (ROTATING SHIFTS)



Before we move on to the next subject, I would like to talk a bit about *circular shifts* (*rotating shifts*), which are important in cryptography.

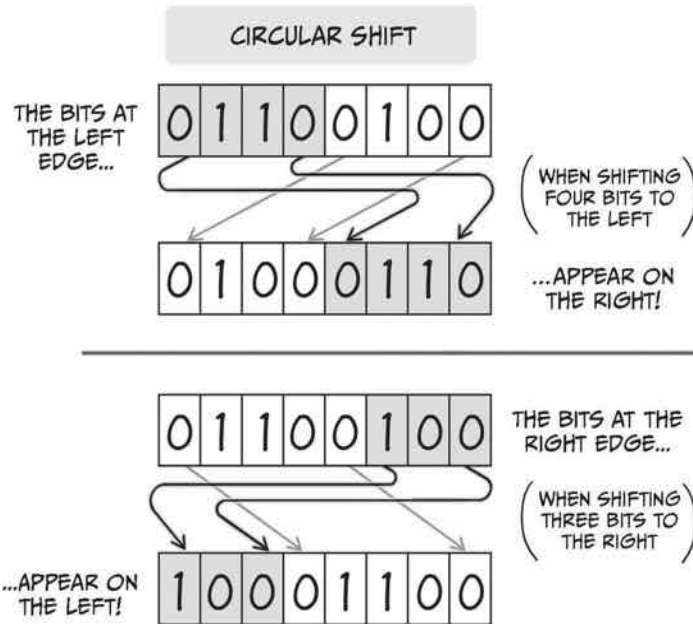
The easiest way to think about it is as if the edges of the bit sequence in the accumulator were glued together into a wheel that can rotate freely.



Oh. It's like we stuck the two edges of the tape together. *Spinnn!*



Applying circular shifts has the following effect. Remember that the left edge (most significant bit) and the right edge (least significant bit) are connected.



DATA TRANSFER OPERATIONS

OKAY, LET'S TALK A BIT ABOUT SOME INSTRUCTIONS THAT AREN'T CALCULATIONS.

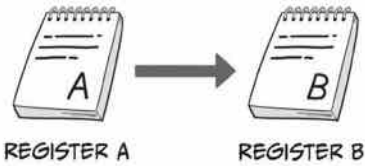
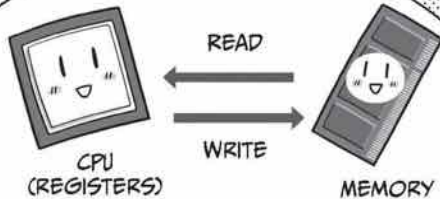
JUST WHAT I WAS HOPING FOR!



FIRST OFF, WE HAVE THE DATA TRANSFER INSTRUCTION. AS YOU MIGHT GUESS, IT'S AN INSTRUCTION THAT DEALS WITH THE TRANSFER OF DATA.

I KNOW THIS!

THEY'RE THE INSTRUCTIONS USED WHEN THE CPU REGISTERS READ AND WRITE DATA FROM MEMORY, RIGHT?

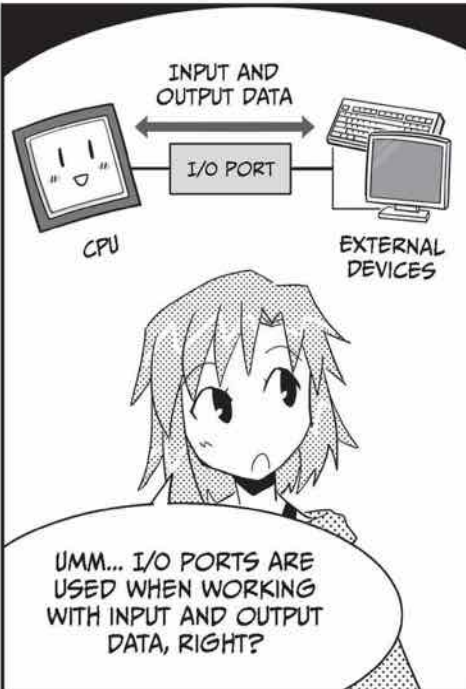


YEAH, BUT THAT'S NOT ALL. THEY'RE ALSO USED TO TRANSFER DATA BETWEEN REGISTERS IN THE CPU.

INPUT/OUTPUT INSTRUCTIONS

NEXT UP ARE INPUT/OUTPUT INSTRUCTIONS.

THESE INSTRUCTIONS ARE USED WHEN THE CPU EXCHANGES DATA* WITH EXTERNAL DEVICES (I/O DEVICES AND SO ON).



YEAH, YOU REMEMBERED!

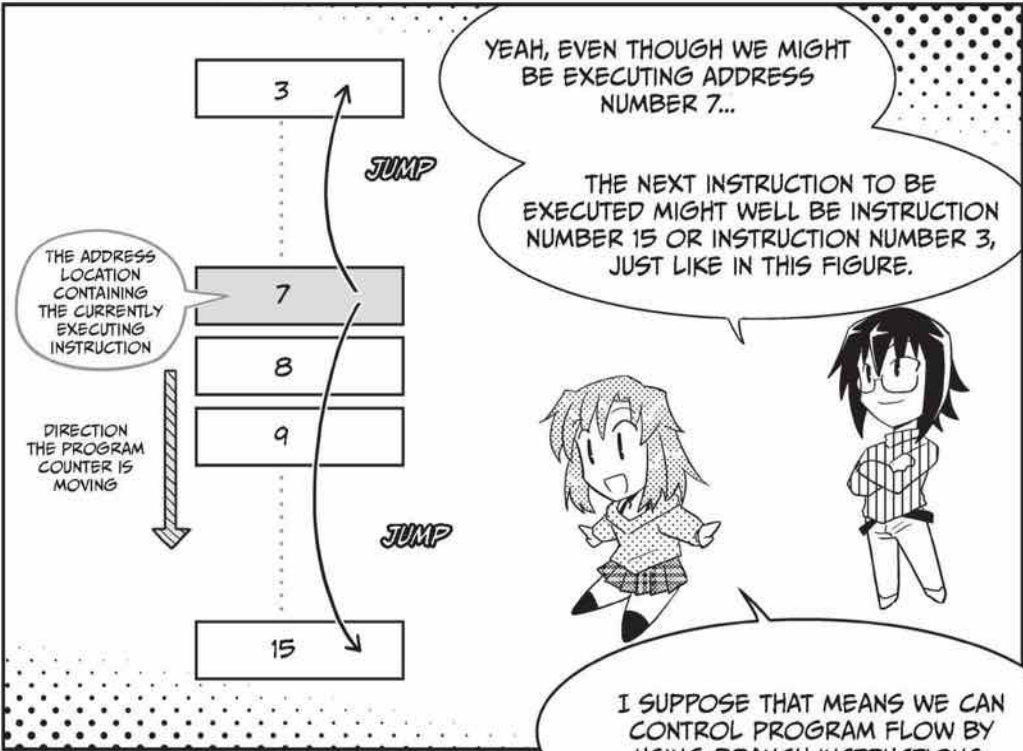
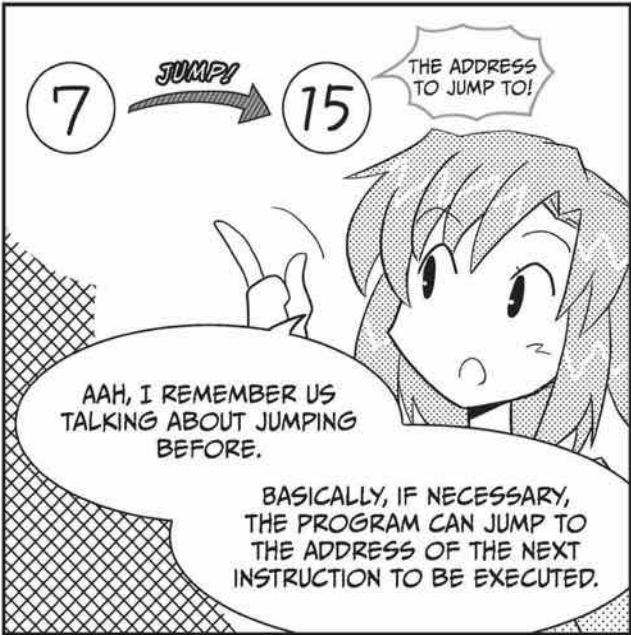
WELLLL, IF YOU HAVE MY INPUT CAPABILITIES, YOU DON'T FORGET ANYTHING YOU'VE LEARNED...

IGNORES

RIIIIGHT... LET'S MOVE ON TO THE NEXT INSTRUCTION!

* THERE ARE TWO TYPES OF DATA TRANSFER METHODS. SEE PAGE 185 FOR MORE INFORMATION.

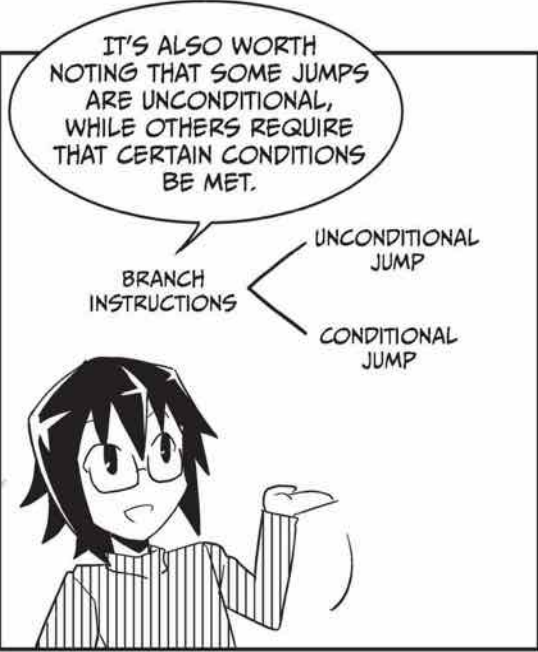
BRANCH INSTRUCTIONS



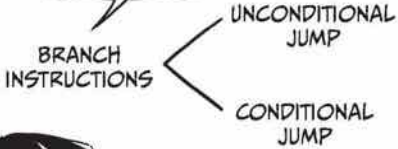
* THERE ARE CASES IN WHICH WE DISCRIMINATE BETWEEN BRANCH INSTRUCTIONS AND JUMP INSTRUCTIONS.



SO... I GUESS IT'S ONE SMALL STEP FOR MAN, ONE GIANT LEAP FOR COMPUTERS! RIGHT?



IT'S ALSO WORTH NOTING THAT SOME JUMPS ARE UNCONDITIONAL, WHILE OTHERS REQUIRE THAT CERTAIN CONDITIONS BE MET.



I SEE! THERE ARE DOLPHINS THAT JUMP WHENEVER THEY FEEL LIKE IT, WHILE OTHER DOLPHINS JUMP ONLY WHEN THERE'S FOOD!

SURE... WHATEVER HELPS YOU REMEMBER...

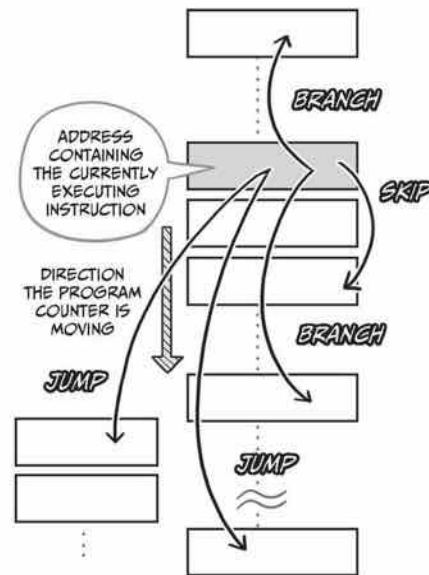
IT ALL MAKES SENSE!!

THAT'S AN ORCA, BY THE WAY.

BRANCH INSTRUCTIONS, JUMP INSTRUCTIONS, AND SKIP INSTRUCTIONS



When it comes to branch instructions, there is unfortunately no standard terminology. Depending on the CPU maker, the instructions might be known as branches, jumps, or even skips. But lately, it's become popular to differentiate among them in the following way.



THE DIFFERENCES AMONG THE THREE

1. *Branch instructions* branch to addresses not far from the execution site.
2. *Jump instructions* jump to addresses farther away from the execution site than branch instructions do.
3. *Skip instructions* can either skip or not skip the next instruction to be executed.



Ohh! So they're different in terms of the distance moved. Pretty cool.



There are also other program control instructions, such as STOP and SLEEP instructions.

DIFFERENT CPUs USE DIFFERENT TERMINOLOGY

If you look at the mnemonic tables of CPUs from companies like Intel (the i8080) and Zilog (the Z80) at the dawn of the 8-bit CPU, you can't even find the word *branch* mentioned. If you instead look at the single-chip 16-bit TMS9900 CPUs made by Texas Instruments (TI) in 1974, *jump* was used for short branch operations, while *branch* was used solely for branch operations concerning registers. Then, the ATmega series CPUs made by Atmel, which are part of the Arduino micro-controllers, use *jump* for changing the current execution address unconditionally, but they also have skips and branch instructions that relate to the currently executing address.

CONDITION EVALUATION AND STATUS FLAGS

FINALLY, LET'S TALK A BIT ABOUT CONDITION EVALUATION (COMPARISON AND OTHER INSTRUCTIONS).

A GOOD WAY TO THINK ABOUT IT IS TO CONSIDER THE ATM EXAMPLE AGAIN.

THE OPERATION RESULT WAS POSITIVE.



(SEE PAGE 26.)

THE OPERATION RESULT WAS NEGATIVE.



AH, SUCH COLD-HEARTED JUDGEMENT! TO HAVE YOUR FATE DECIDED BY A COMPARISON BETWEEN THE VALUES OF YOUR ACCOUNT BALANCE AND THE AMOUNT YOU WANT TO WITHDRAW...

I SUPPOSE. IN THIS CASE, TWO INSTANCES OF DATA WERE EVALUATED USING A COMPARISON INSTRUCTION THAT HAS SOME KIND OF CONDITION.



IS DATA A BIGGER?
WHAT'S IT GOING TO BE?

COMPARES THEM AND DECIDES!



CPU

UGHH. AND WHAT A CALLOUS DECISION IT IS...

NOW, I WANT YOU TO PAY ATTENTION TO...

EXAMPLE: $5 - 3 = 2$

THE STATUS FLAG* THAT IS USED WHEN EVALUATING WHETHER A CONDITION IS MET.

STATUS FLAG... DIDN'T WE TALK ABOUT STATUS OUTPUT BEFORE? WASN'T THAT SOME VALUE THAT INDICATED THE STATE OF SOME OPERATIONAL RESULT, LIKE WHETHER IT WAS POSITIVE AND STUFF LIKE THAT?

* ALSO CALLED A STATUS BIT

YEAH. THE PURPOSE OF THE STATUS FLAG IS TO RECORD INFORMATION LIKE THAT.

THE FLAG IS UP!	THE FLAG IS NOT UP...
SET (1)	RESET (0)
THE RESULT OF THE CALCULATION IS NEGATIVE!	THE RESULT OF THE CALCULATION IS POSITIVE!

IT SIGNALS THE RESULT OF A CALCULATION USING EITHER A ZERO OR A ONE.

HMM. SO A FLAG IS SET WHENEVER A CONDITION IS EVALUATED TO BE NEGATIVE?

THERE ARE ACTUALLY MANY TYPES OF FLAGS, EACH OF THEM RAISED (SET TO 1) IF SOME PARTICULAR CONDITION ASSOCIATED WITH THE FLAG EVALUATES TO TRUE.



DECISIONS ARE MADE IN ACCORDANCE WITH EITHER A SINGLE FLAG STATE OR SOME COMBINATION OF SEVERAL FLAG STATES.



SIGN FLAG
SET WHEN THE RESULT OF A CALCULATION IS NEGATIVE



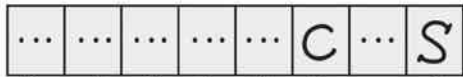
CARRY FLAG
SET WHEN THE CALCULATION RESULTS IN A CARRIED DIGIT

IN ADDITION TO THESE, WE INTRODUCE SOME OTHER COMMON FLAGS ON PAGE 187.

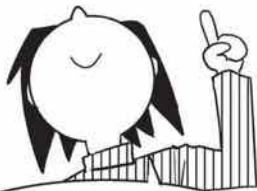
SO WE CAN LOOK AT SINGLE FLAGS OR COMBINATIONS OF FLAGS TO DECIDE WHAT TO DO, DEPENDING ON WHETHER SOME CONDITION IS MET.



STATUS REGISTERS



EVERY BIT STORES DIFFERENT STATUS FLAGS
CARRY FLAG
SIGN FLAG
THEY ARE EITHER 1 OR 0.



A STATUS REGISTER IS SIMPLY THE 8-BIT OR 16-BIT COMBINATION OF A LOT OF THESE FLAGS (EACH OF THEM ONE BIT).

OOH, STATUS REGISTERS! THEY'RE LIKE HARDWORKING DETECTIVES, EACH OF THEM REMEMBERING DIFFERENT INFORMATION ABOUT AN OPERATOR!!



WHO ARE YOU SUPPOSED TO BE?

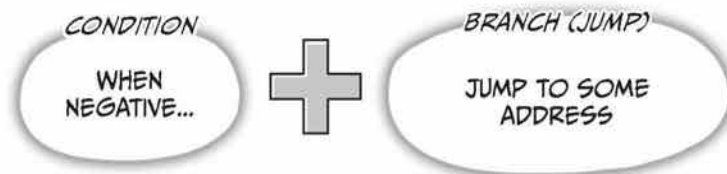


PUTTING BRANCHES AND CONDITION EVALUATION TOGETHER



Okay, we've learned about branch instructions and condition evaluation, but we can get some truly useful instructions by putting the two together.

One example is the *jump on minus* instruction. It simply states that the program should jump to some address if the value in the accumulator is negative.



So basically, jump to this address if these conditions are all met!

Or, put another way, the program is able to change its execution depending on some condition.



Yeah, we can also make other combinations of instructions like *conditional jumps*, *conditional skips*, and *conditional branches*. Thanks to these, we can do some really useful things.

SOME THINGS WE CAN DO USING CONDITIONAL JUMPS AND OTHER INSTRUCTIONS

1. We can run a different program depending on some condition.
2. We can decide not to run a program (skipping it) depending on some condition.
3. We can set and reset bits on output ports depending on some condition.

For example, we could control a lamp by setting or resetting some I/O port value to turn the lamp on and off.



Whoa! This seems absolutely essential not only for computers but also for any electrical application really!

Operand Types



LET'S SEE. OPERANDS ARE THE DATA AND ADDRESSES USED AS THE TARGET OF AN OPERATION, RIGHT?

OPCODE	OPERAND
--------	---------

TYPE OF INSTRUCTION

THE TARGET DATA OR ADDRESS OF AN OPERATION

THE NUMBER OF OPERANDS ALSO DEPENDS ON THE TYPE OF INSTRUCTION WE'RE DEALING WITH. LOOK AT THE NEXT FIGURE.

BUT THEY CAN ALSO BE REGISTERS, IF I REMEMBER CORRECTLY.

OPCODE TWO OPERANDS
 ↓ ↓

ADD	a	b
-----	---	---

『 ADD A AND B 』

IN THIS EXAMPLE, IT SEEMS THE ADD OPERATION NEEDS TWO OPERANDS.

IS THAT CORRECT?



YEAH. AND THE ADD HERE IS ACTUALLY SOMETHING CALLED A *MNEMONIC** THE HUMAN-READABLE REPRESENTATION OF THE OPCODE.

ALL INSTRUCTIONS HAVE EITHER ZERO, ONE, OR TWO OPERANDS. THIS INSTRUCTION HAPPENS TO HAVE TWO.



* IN ENGLISH, *MNEMONICS* ARE MENTAL TOOLS THAT HELP WITH REMEMBERING THINGS.

HUH? HOW COULD AN INSTRUCTION HAVE NO OPERANDS?! THAT SEEMS COMPLETELY POINTLESS!


HEH. WELL OPCODES THAT HAVE NO OPERANDS DO EXIST.



FOR EXAMPLE... THE SET ACCUMULATOR TO 1 OPCODE!!

SET ACCUMULATOR TO 1

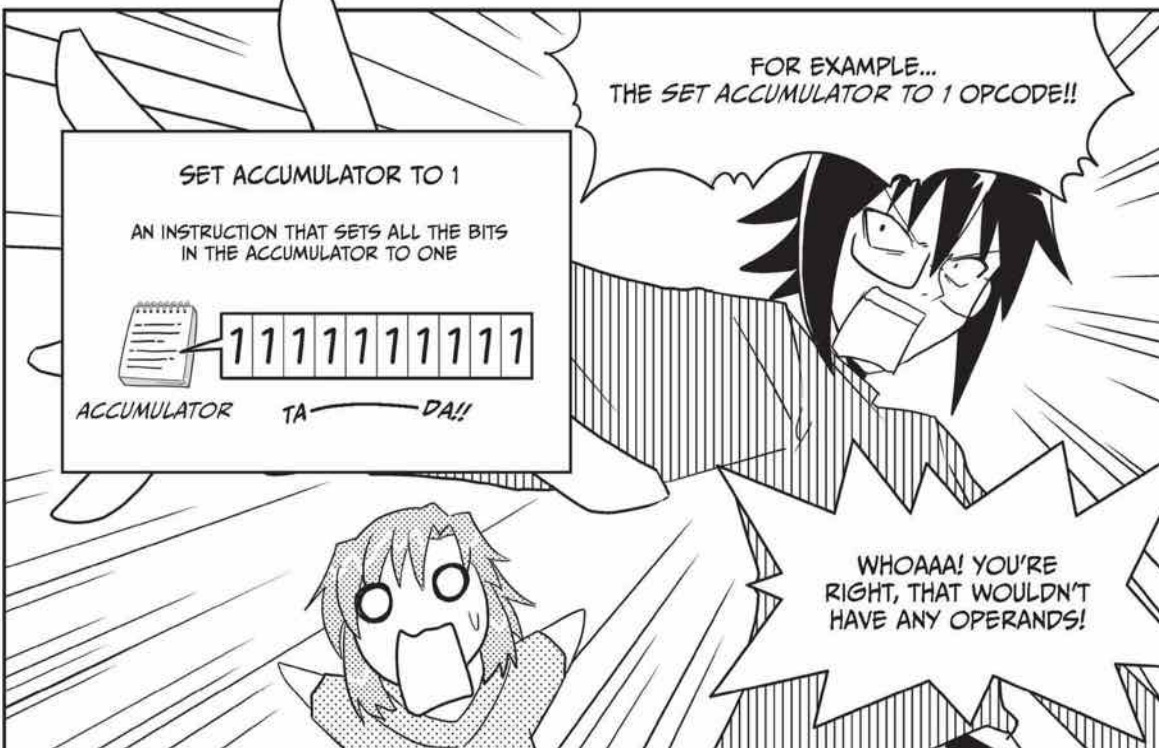
AN INSTRUCTION THAT SETS ALL THE BITS IN THE ACCUMULATOR TO ONE



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

ACCUMULATOR TA DA!!

WHOOAAA! YOU'RE RIGHT, THAT WOULDN'T HAVE ANY OPERANDS!





ALSO, FOR TWO-OPERAND OPERATIONS WHERE BOTH OPERANDS ARE ADDRESSES...

WE CALL THE FIRST OPERAND THE *SOURCE OPERAND* AND THE SECOND THE *DESTINATION OPERAND*.

SO THEIR ROLES ARE DECIDED ALREADY.

OPCODE	FIRST	SECOND
--------	-------	--------

AS YOU CAN TELL FROM THE NAMES, OPERATIONS LIKE THIS USE THE DATA IN THE SOURCE OPERAND TO AFFECT DATA IN THE DESTINATION OPERAND.

OPERANDS TAKE MANY FORMS

LET'S FINALLY
APPROACH THE
CORE SUBJECT
HERE...



DIFFERENT KINDS OF OPERANDS

- Immediate value processing
- Address reference

★ ADDRESSING MODE

(HOW WE POINT TO ADDRESSES AND OPERANDS)

1. Absolute addressing
2. Relative addressing
3. Indirect addressing
4. Address modification

LOOK AT THIS! THERE
ARE LOTS OF DIFFERENT
OPERANDS, TOO!

AH... SO MANY! ESPECIALLY THE
ADDRESSING MODES! WHY ARE
THERE SO MANY TYPES?!



DON'T WORRY.
I'LL EXPLAIN THEM
ONE BY ONE.

COME! IT'S TIME TO
OPERATE WITH OPERANDS!



SMACK!

I'VE DEDUCED
THAT...

YOU'RE REALLY
ENJOYING THAT
COSTUME.



IMMEDIATE VALUE PROCESSING



Let's start with *immediate values*.

The word *immediate* here means that the value will be used right away, just as it is. In other words, the operand itself is a value.

Add 2
MNEMONIC "ADD" IMMEDIATE VALUE OPERAND



You're right! So I guess this would mean, "Add two to the value in the accumulator."

A Shift L 2
ARITHMETIC SHIFT LEFT IMMEDIATE VALUE OPERAND



And this example shows a two-bit arithmetic left shift. Immediate value operands can be used with many different operations—for example, arithmetic operations, shift operations, and logic operations.



In the end, it's just a concrete value though, right? I learned about immediate values immediately!



ADDRESS REFERENCES



I guess *address references* have to be operands that are addresses, like address number 1 or number 2....



Yeah. Either internal* or external memory addresses, to be exact. The operation will grab the data on the address in question and use it.

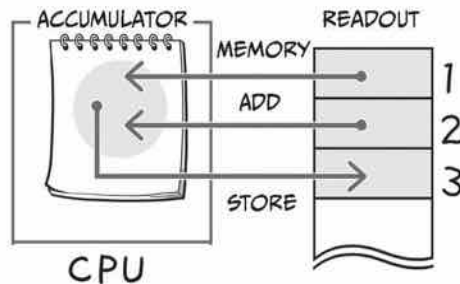


Hmm, so for example. I could instruct the CPU to get the data on address 1 and address 2, add them, and store the result on address 3—right?



Yeah, in that case, it would turn out like this.

LDA ADDRESS 1 Read the data on address 1 and store it in the accumulator.
ADD ADDRESS 2 Add the data on address 2 to the accumulator.
STA ADDRESS 3 Store the value in the accumulator to address 3.



Oh! So is it important that calculations are always done in the accumulator then?



That's right. The accumulator even has its own mnemonic: *A*. The mnemonic *LDA* means *LoaD to Accumulator*, while *STA* means *STore Accumulator*.

* If you look at the architectural diagram on page 106, you can see that classic CPUs had internal RAM. These were also referenced using memory addresses.

WHAT ARE ADDRESSING MODES?



HERE ARE THE ONES WE LISTED BEFORE.

HMM...

★ ADDRESSING MODES

1. Absolute addressing	3. Indirect addressing
2. Relative addressing	4. Address modification



SURE, IT CERTAINLY WOULD BE EASIER TO SAY, "IT'S ON FIVE," AND HAVE THE DATA ALWAYS BE IN NUMBER FIVE.

1
2
3
4
5 DATA

NUMBER FIVE

THIS IS WHAT WE CALL ABSOLUTE ADDRESSING.

YEAH, YEAH! THAT'S THE ONLY THING THAT MAKES COMMON SENSE, RIGHT?

EFFECTIVE ADDRESS

DATA

BY THE WAY... WE CALL THE ADDRESS THAT ACTUALLY CONTAINS THE DATA WE'RE AFTER THE EFFECTIVE ADDRESS.

IN THIS CASE, THAT WOULD BE ADDRESS NUMBER FIVE.

YEAH, IT WOULD.

LET'S LOOK AT SOME OF THE OTHER METHODS.

LET'S SAY WE POINTED OUT NUMBER TWO, AND WHEN WE OPENED IT...

NUMBER TWO

1
2
3
4
5

DATA

A REFERENCE TO NUMBER FIVE (ADDRESS NUMBER FIVE).

WE FOUND OUR DATA WAS AT NUMBER FIVE!

THIS IS WHAT WE WOULD CALL *INDIRECT ADDRESSING*.

WHAAAT? WHY WOULD YOU EVER DO SOMETHING SO UNNECESSARILY COMPLICATED...

HA! IT'S LIKE FINDING A LONG-LOST WILL OR SEARCHING FOR HIDDEN TREASURE! ONLY THE ONES WHO ARE TENACIOUS ENOUGH TO MAKE IT TO THE END CAN GET THE PRIZE.

CALM DOWN.

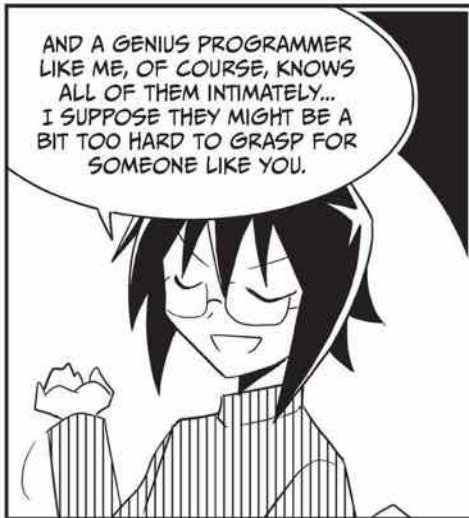
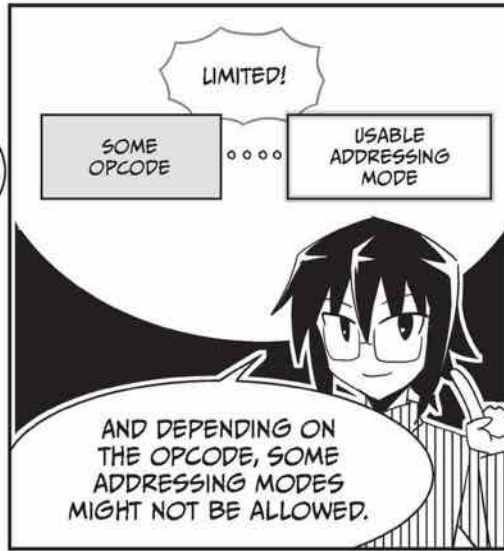
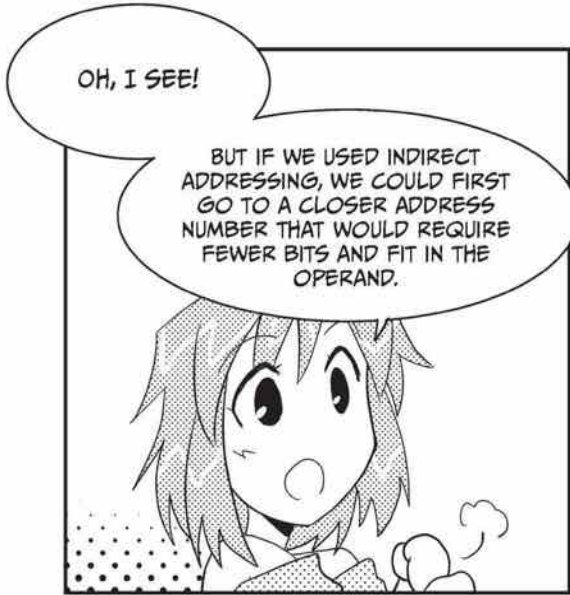
THERE ARE SOME MERITS TO INDIRECT ADDRESSING. IF, FOR EXAMPLE, YOU TRIED TO DIRECT ADDRESS AN ADDRESS WITH A VERY LONG NUMBER LIKE "ADDRESS NUMBER 9999...9..."

NOPE!

LOOOO — NG...

THE NUMBER OF BITS RESERVED FOR THE OPERAND IS LIMITED, AND WE CAN'T HAVE ARBITRARILY LONG NUMBERS!

THE NUMBER WOULD REQUIRE MORE BITS THAN WE HAVE AVAILABLE FOR THE OPERAND, AND THAT WOULDN'T WORK, RIGHT?



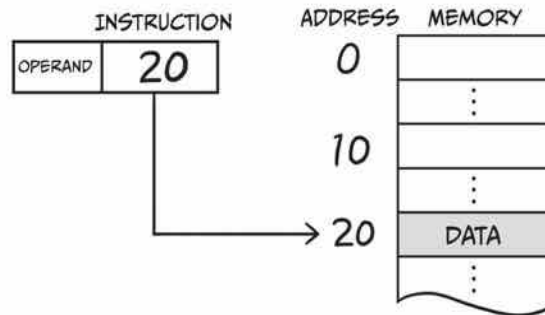


ADDRESSING MODE OVERVIEW

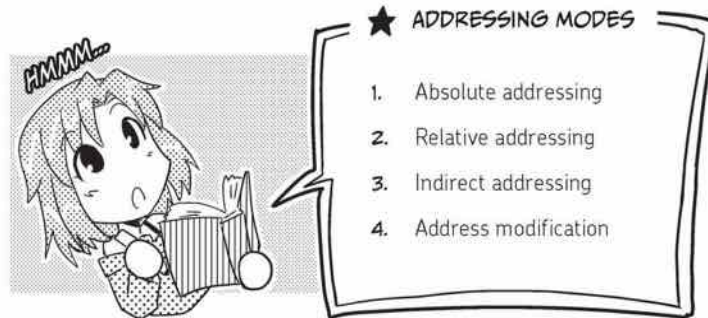
Modern CPUs can address memory in several different ways, which we call addressing modes.

ABSOLUTE ADDRESSING

Absolute addressing is when the operand's value is used as the effective address (the address where the operation's target data is located). It is also sometimes called *direct addressing*.



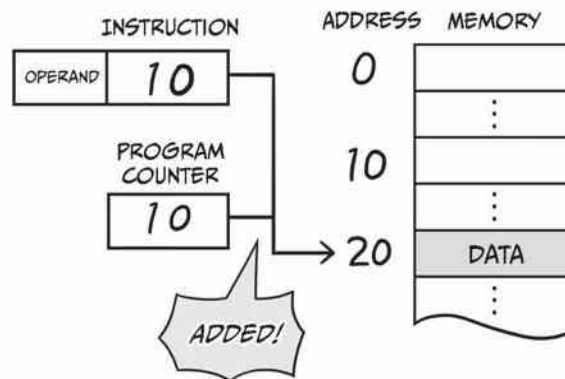
Depending on the CPU, there are cases where the size of the opcode makes it so the CPU can't address the entire address space. It's possible to lengthen the operand size if need be, however. In 16-bit CPUs, it's common practice to store the opcode and the operand in 16 bits (2 bytes), but if the operand is lengthened, the instruction could end up being 4 bytes, or even 8.





RELATIVE ADDRESSING

Relative addressing is when the effective address is the result of the sum of the operand value and the program counter.



Relative addressing is most commonly used for jump instructions. Since the distance to the address we want to point out is limited by the range expressed by the two's complement of the number of bits available in the operand, relative addressing is best used for conditional branching instructions in the program and is not recommended for any larger jumps in the address space.

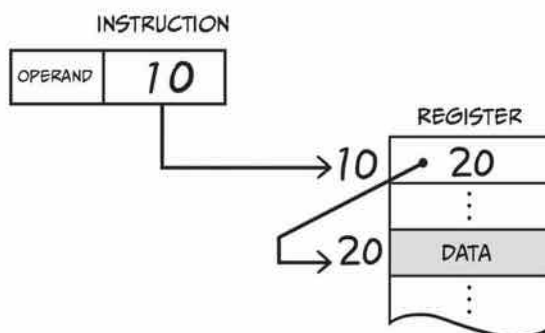
The base value for an operation that uses relative addresses is the current value of the program counter, or PC. As soon as the PC has read an opcode, it immediately starts pointing to the next opcode to be processed.

Also, besides using the program counter as the base value for the relative address, we can use the address in a register instead. We call addresses like these *xx-register relative addresses*.



INDIRECT ADDRESSING

Indirect addressing is used when the operand contains the address to some register and that register, in turn, contains the effective address for our target data.



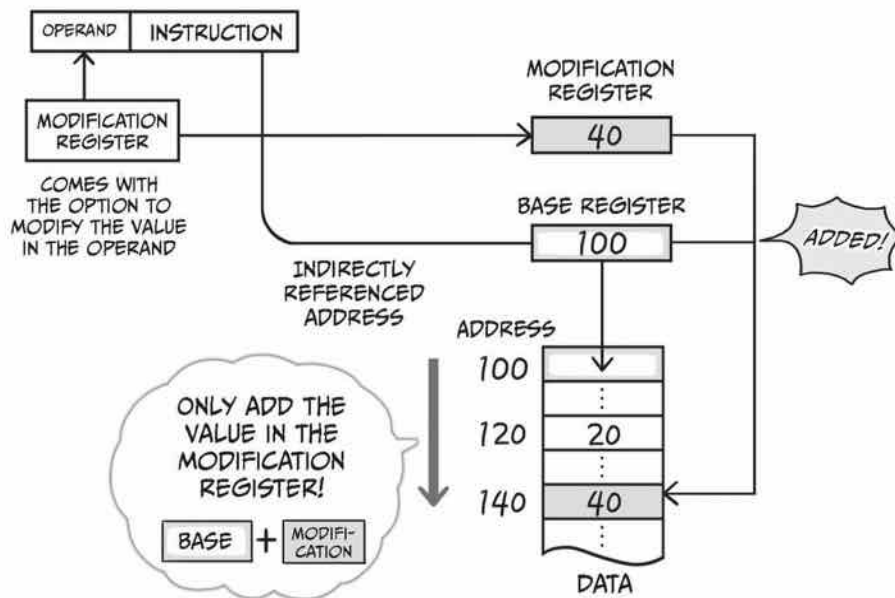
The best way to think about indirect addresses (and the address modification mode coming up next) is their close relationship with arrays in the programming language C. When working with arrays, you generally use the effective address as a starting point and add or subtract some offset values to or from it, ending up with a new address in the end. This process is what we call address modification.





ADDRESS MODIFICATION

Address modification is the process of using the value stored in a modification register to modify a number or address. We get the effective address by adding the value in the modification register to a base value, which may be stored in another register or in the program counter or even in an immediate value.



One of the most commonly used modification registers is the *index register*. We usually call the register containing the value we want to use as a base the *base register*. Most CPUs create the effective address in this case by simply adding the value in the base register and the value in the modification register (the index register, for example) together.

By using address modification in this way, you can achieve very practical effects. For example, you could extract some particular data from a set by specifying the offset in the index register and the start of the data set in a base register.

The Structure of Operations in the ALU

A LOOK INSIDE THE ALU

WE'VE FINALLY ARRIVED AT TODAY'S HIGH POINT! IT'S THE THING YOU'VE BEEN WAITING FOR...

LET'S GO INTO FUN-TIME MODE!!

...HUH? I WASN'T WAITING FOR ANYTHING.

SO COLD!

REMEMBER, TO UNDERSTAND ARITHMETIC OPERATION INSTRUCTIONS AND LOGIC OPERATION INSTRUCTIONS, YOU HAVE TO...

ARITHMETIC LOGIC UNIT

ALU

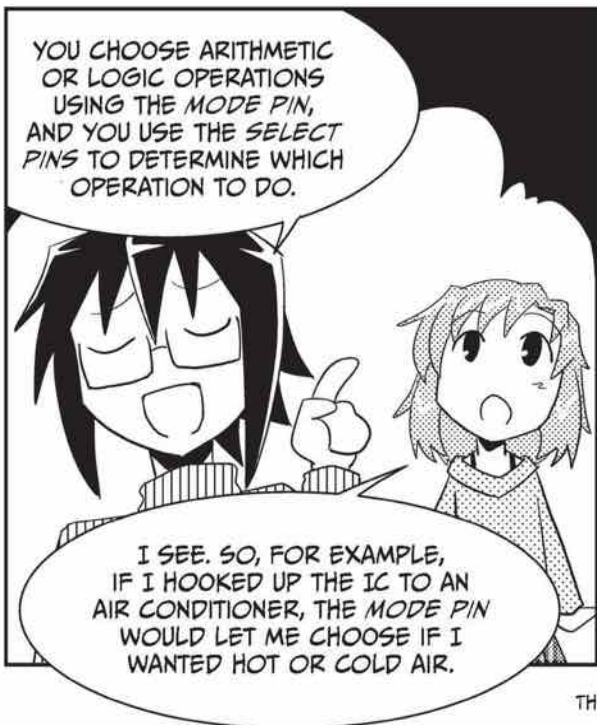
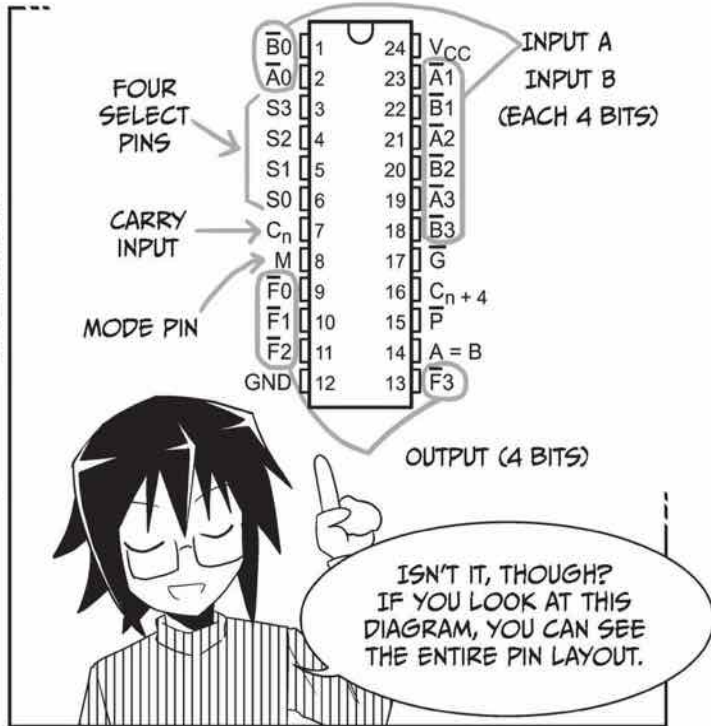
AAH! RIGHT, WE WERE TALKING ABOUT THE ALU!

YEP! I'M GOING TO USE THIS 4-BIT ALU IC AS AN EXAMPLE. ITS NAME IS 74S181.*

A 74S181, MADE BY TEXAS INSTRUMENTS

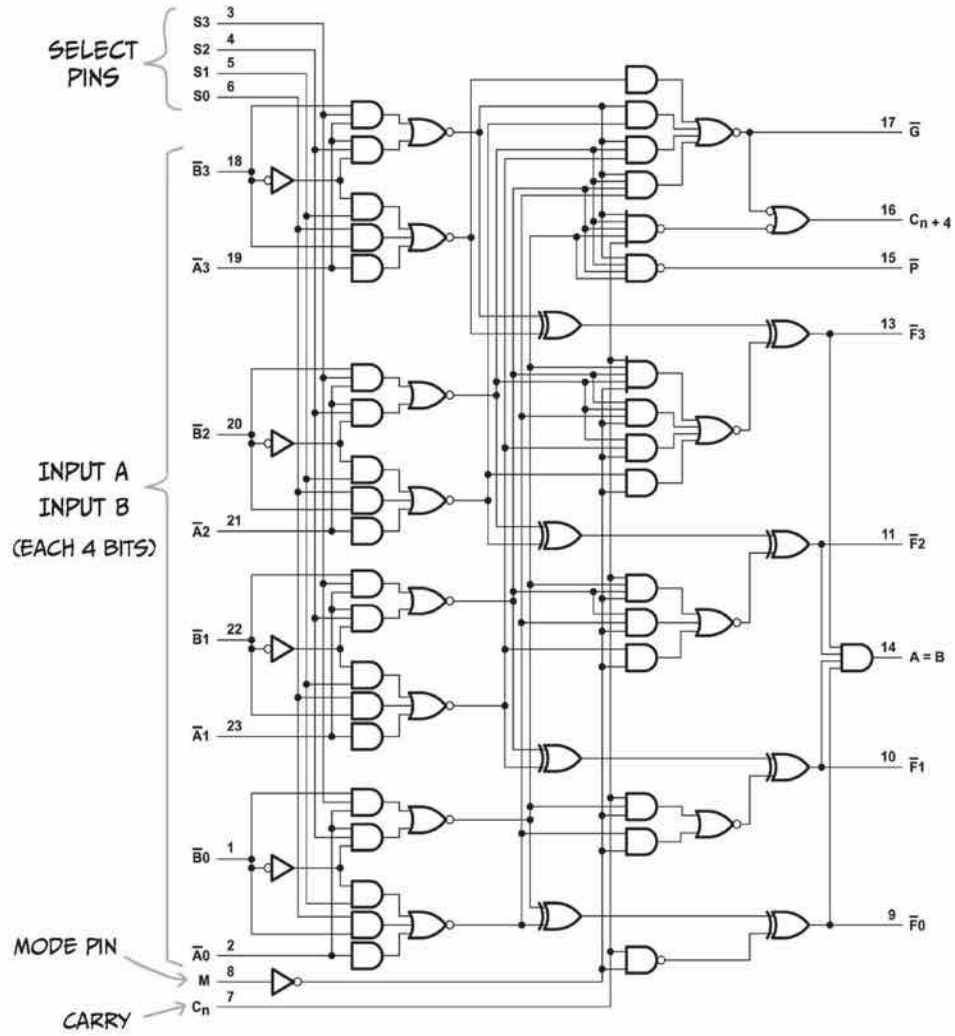
BUT IT'S SOMETIMES CALLED A BIT SLICE.

* THE TI MICROCONTROLLER WE TALKED ABOUT ON PAGE 157 USED FOUR OF THESE 74S181 CIRCUITS. THEY WERE USED IN MANY HIGH-SPEED CALCULATION COMPUTERS—FOR EXAMPLE, IN AIRCRAFT SIMULATORS AND THE LIKE. THE 74S181 CIRCUIT WAS EVENTUALLY SIMPLIFIED INTO THE 74S381 CIRCUIT.





BASIC CIRCUIT ARCHITECTURE OF THE 74S181
REFERENCED FROM A TEXAS INSTRUMENTS DATA SHEET (PARTIALLY REVISED)



Whaa! It's really complicated, but I can see the four-bit inputs A and B clearly. I also see the select pins S0 through S3 and the mode pin M.



Yeah, that's right. The carry is also there, denoted by C_n .



74S181 FUNCTION TABLE

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC OPERATIONS	C _n = H (no carry)	C _n = L (with carry)
L	L	L	L	$F = \bar{A}$	$F = A$	$F = A \text{ PLUS } 1$
L	L	L	H	$F = \bar{A} + \bar{B}$	$F = A + B$	$F = (A + B) \text{ PLUS } 1$
L	L	H	L	$F = \bar{A}B$	$F = A + \bar{B}$	$F = (A + \bar{B}) \text{ PLUS } 1$
L	L	H	H	$F = 0$	$F = \text{MINUS } 1 \text{ (2'S COMPL)}$	$F = \text{ZERO}$
L	H	L	L	$F = \bar{A}\bar{B}$	$F = A \text{ PLUS } \bar{A}\bar{B}$	$F = A \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$
L	H	L	H	$F = \bar{B}$	$F = (A + B) \text{ PLUS } \bar{A}\bar{B}$	$F = (A + B) \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$
L	H	H	L	$F = A \oplus B$	$F = A \text{ MINUS } B \text{ MINUS } 1$	$F = A \text{ MINUS } B$
L	H	H	H	$F = \bar{A}\bar{B}$	$F = \bar{A}\bar{B} \text{ MINUS } 1$	$F = \bar{A}\bar{B}$
H	L	L	L	$F = \bar{A} + B$	$F = A \text{ PLUS } AB$	$F = A \text{ PLUS } AB \text{ PLUS } 1$
H	L	L	H	$F = \bar{A} \oplus \bar{B}$	$F = A \text{ PLUS } B$	$F = A \text{ PLUS } B \text{ PLUS } 1$
H	L	H	L	$F = B$	$F = (A + \bar{B}) \text{ PLUS } AB$	$F = (A + \bar{B}) \text{ PLUS } AB \text{ PLUS } 1$
H	L	H	H	$F = AB$	$F = AB \text{ MINUS } 1$	$F = AB$
H	H	L	L	$F = 1$	$F = A \text{ PLUS } A^*$	$F = A \text{ PLUS } A \text{ PLUS } 1$
H	H	L	H	$F = A + \bar{B}$	$F = (A + B) \text{ PLUS } A$	$F = (A + B) \text{ PLUS } A \text{ PLUS } 1$
H	H	H	L	$F = A + B$	$F = (A + \bar{B}) \text{ PLUS } A$	$F = (A + \bar{B}) \text{ PLUS } A \text{ PLUS } 1$
H	H	H	H	$F = A$	$F = A \text{ MINUS } 1$	$F = A$

* Each bit is shifted to the more significant position.

For more information on the symbols used in these formulas, please see pages 55–59. PLUS and MINUS are exactly what they seem. The symbols +, −, and ⊕ are symbols used in Boolean algebra (logical algebra).

There are also some redundant or unnecessary operations in the diagram, as you might see.



The most important parts of the 74S181 function table are marked with gray.



First off, *M* is the mode pin, *H* stands for high, and *L* stands for low. When *M* = *H*, we are using logic operations. If *M* = *L*, arithmetic operations are being used instead.

Arithmetic operations then further differ depending on whether we have a carry or not. If *C_n* = *H*, that means we do not have a carry, and if *C_n* = *L*, we do have a carry.



And *S* is the four select pins, right? Depending on the combination, we have 16 (2⁴) different operations to choose from!



Now let's take a closer look at the opcodes in the function table. For convenience, let's assign a number to each of the opcodes: 0-15, or 16 in total. Of course these numbers may not be the same for other CPUs.

I'll explain the ones with gray backgrounds in detail.

Arithmetic Operations			Logic Operations	
	No Carry	With Carry		
0	$F = A$	$F = A \text{ PLUS } 1$	0	$F = \bar{A}$
1	$F = A + B$	$F = (A + B) \text{ PLUS } 1$	1	$F = \overline{A + B}$
2	$F = A + \bar{B}$	$F = (A + \bar{B}) \text{ PLUS } 1$	2	$F = \bar{A}B$
3	$F = \text{MINUS } 1 \text{ (2'S COMPL)}$	$F = \text{ZERO}$	3	$F = 0$
4	$F = A \text{ PLUS } \bar{A}\bar{B}$	$F = A \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$	4	$F = \bar{A}\bar{B}$
5	$F = (A + B) \text{ PLUS } \bar{A}\bar{B}$	$F = (A + B) \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$	5	$F = \bar{B}$
6	$F = A \text{ MINUS } B \text{ MINUS } 1$	$F = A \text{ MINUS } B$	6	$F = A \oplus B$
7	$F = \bar{A}\bar{B} \text{ MINUS } 1$	$F = \bar{A}\bar{B}$	7	$F = A\bar{B}$
8	$F = A \text{ PLUS } AB$	$F = A \text{ PLUS } AB \text{ PLUS } 1$	8	$F = \bar{A} + B$
9	$F = A \text{ PLUS } B$	$F = A \text{ PLUS } B \text{ PLUS } 1$	9	$F = \bar{A} \oplus \bar{B}$
10	$F = (A + \bar{B}) \text{ PLUS } AB$	$F = (A + \bar{B}) \text{ PLUS } AB \text{ PLUS } 1$	10	$F = B$
11	$F = AB \text{ MINUS } 1$	$F = AB$	11	$F = AB$
12	$F = A \text{ PLUS } A$	$F = A \text{ PLUS } A \text{ PLUS } 1$	12	$F = 1$
13	$F = (A + B) \text{ PLUS } A$	$F = (A + B) \text{ PLUS } A \text{ PLUS } 1$	13	$F = A + \bar{B}$
14	$F = (A + \bar{B}) \text{ PLUS } A$	$F = (A + \bar{B}) \text{ PLUS } A \text{ PLUS } 1$	14	$F = A + B$
15	$F = A \text{ MINUS } 1$	$F = A$	15	$F = A$

IMPORTANT ARITHMETIC OPERATION INSTRUCTIONS

OPCODE 6

No carry: The calculation result F is the difference between A and B minus 1.

With carry: The calculation result F is the difference between A and B .

OPCODE 9

No carry: The calculation result F is the sum of A and B .

With carry: The calculation result F is the sum of A and B plus 1.



IMPORTANT LOGIC OPERATION INSTRUCTIONS

OPCODE 1: NOR (A, B)

The operational result F is the negated output of the OR between the A and B bits. That is, it is the NOR of the bits in A and B.

OPCODE 3: ZERO

The operational result F is 0, regardless of the input.

OPCODE 4: NAND (A, B)

The operational result F is the negated output of the AND between the A and B bits. That is, it is the NAND of the bits in A and B.

OPCODE 5: NOT (B)

The operational result F is the NOT of input B. That is, every 0 bit in B is flipped to a 1, and every 1 bit in B is flipped to a 0.

OPCODE 6: EXOR (A, B)

The operational result F is the EXOR of the bits in A and B.

OPCODE 9: EXNOR (A, B)

The operational result F is the negated output of the EXOR of the bits in A and B.

OPCODE 10: B

The operational result F is simply B.

OPCODE 11: AND (A, B)

The operational result F is the AND of the bits in A and B.

OPCODE 12: ONES

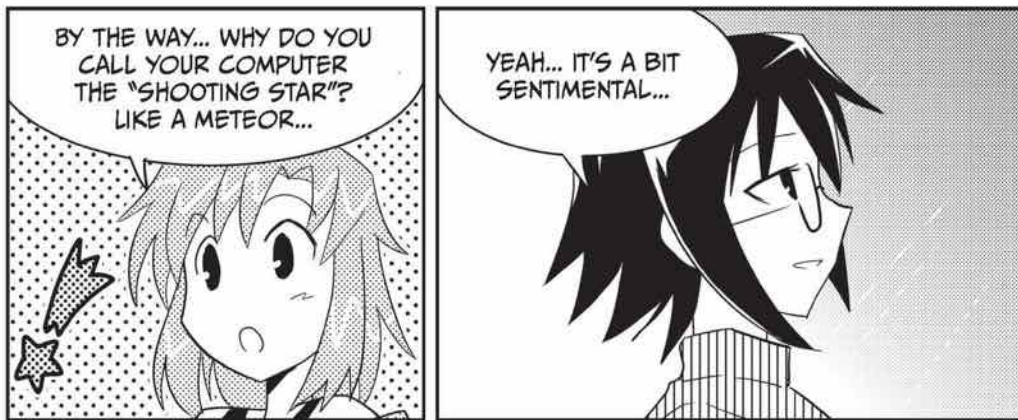
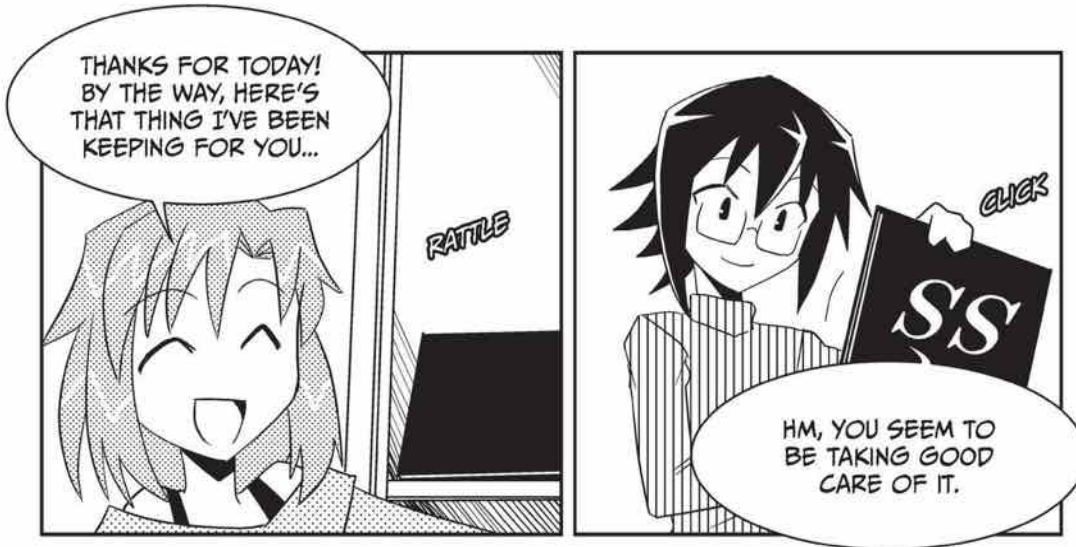
The operational result F is all 1s, regardless of the input.

OPCODE 14: OR (A, B)

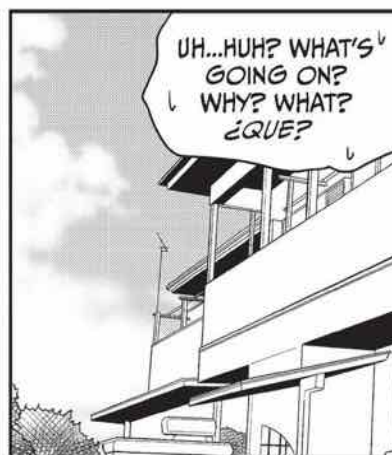
The operational result F is the OR of the bits in A and B.

OPCODE 15: A

The operational result F is simply A.

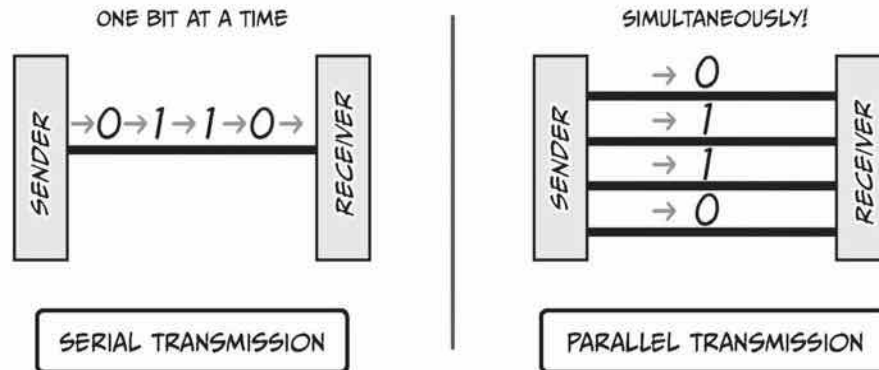






SERIAL TRANSMISSION AND PARALLEL TRANSMISSION

There are two types of digital data transmission: *serial transmission* and *parallel transmission*.



Systems that use serial transmission send data one bit at a time; systems that use parallel transmission send several bits at a time. An interesting thing to note is that *USB* (as in USB memory or USB connectors) is short for Universal Serial Bus, which, as you might have guessed, uses serial transmission.

SHIFT REGISTERS AND PARALLEL-SERIAL CONVERSION

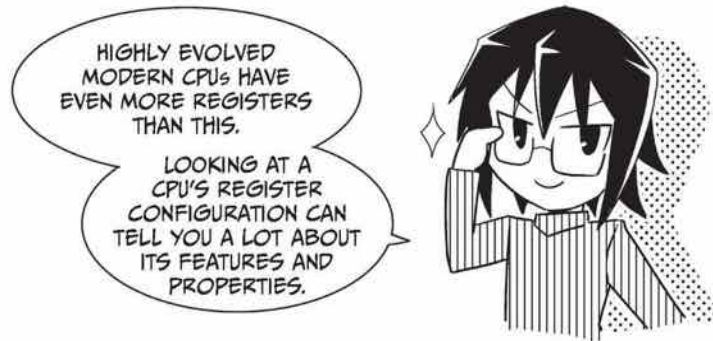
One of the components often used in logic circuits is a *shift register*. This type of register can perform shift operations and nothing else. An example is the accumulator within the ALU.

The most common use for shift registers is to parallel shift several bits of data (for example, 8 bits) to the right in one clock cycle. The rightmost bits are then converted and sent as serial data.

There is some discussion about whether this serial transmission function should be seen as part of CPU functionality or as part of I/O. Overall, it's easier to think of it as the means by which the CPU communicates with devices other than the memory, all bundled together as "I/O devices," and as distinct from things not directly operated by the CPU block.

AN OVERVIEW OF SOME BASIC REGISTERS

Registers are useful in many contexts, and they are essential to the CPU. Here are some basic registers and their functionalities.



ACCUMULATOR

This register stores calculation results from the ALU. It's designed in such a way as to be ready for the next calculation as soon as the previous one is done. It's called the accumulator in part because it's generally used when incrementing values and counting sequentially, but also since it's often used as the input for a follow-up sum right after another calculation has finished.

INSTRUCTION REGISTER & INSTRUCTION DECODER

These registers store and decode the program instructions. This decoding process determines not only which operation to execute but also the operands on which to operate.

STATUS REGISTER

The status register is a collection of flags that take the value 1 or 0 as a result of calculations and operations. These flags can determine the order of program execution and how the CPU interacts with I/O devices. Since flags are only 1 bit each, it is very common to lump them together in 8-bit or even 16-bit registers. There are many different kinds of flags, and you can read more about them starting on page 187.

MODIFICATION REGISTERS (BASE REGISTERS, INDEX REGISTERS)

These registers serve as the starting point in certain addressing modes. The base register serves as a basis for address calculations. In relative addressing, adding an offset to the base register yields an effective address.

Index registers hold fixed values that modify operand immediate values in special circumstances to form the effective address. For example, you would add the offset found in the index register to a data array's base address to find a specific value in the array.

TEMP REGISTER (TEMPORARY REGISTER)

Temp registers are used to save temporary data during the many tasks undertaken by the CPU. Depending on the CPU, some blocks of the circuit might have several temp registers available. You can see a temp register labeled in the diagram of classic CPU architecture on page 106.

PROGRAM COUNTER (PC)

The program counter holds the address to the next instruction to be executed. All CPUs have this register.

STACK POINTER

Necessary when working with a stack, this register holds the last used stack address.

AN OVERVIEW OF SOME BASIC STATUS FLAGS

When the CPU calculates a result, status flags (status bits) might be set or reset. The CPU makes decisions by evaluating the status flags, either just a single flag or a combination of several flags. As a result of these decisions, the program might take different branches or end up doing different calculations.

ZERO FLAG (Z-FLAG)

Indicates whether the accumulator (the result of a calculation) is zero. If the CPU doesn't have a dedicated module for doing comparisons, the Z-flag might also double as the flag that reports the outcome of a comparison test (the EQ-flag).

SIGN FLAG (S-FLAG) OR NEGATIVE FLAG (N-FLAG)

If the accumulator contains a number, this flag tells you whether the number is negative or positive.

CARRY FLAG (C-FLAG) OR OVERFLOW FLAG (OV-FLAG)

Indicates whether a carry or an overflow occurred in the latest arithmetic add operation. It is also set if a shift operation resulted in overflow. In the case of an arithmetic subtraction operation, it is not set if borrowing (the inverse of carrying) didn't occur.

BORROW FLAG

Indicates whether a borrow occurred during a subtraction. More often than not, a borrow is indicated by the carry flag not being set, but in some cases, the borrow flag might be used instead.

GT FLAG

This flag is set if the outcome of a comparison operation was "greater than." The GT flag's associated symbol is >.

LT FLAG

This flag is set if the outcome of a comparison operation was "less than." The LT flag's associated symbol is <.

ODD FLAG

Indicates whether the result of a calculation is an odd number.

INTERRUPT MASK

Set beforehand, the interrupt mask determines what types of interrupts will occur. Setting it to all 1s will disable interrupts.

INTERRUPT FLAG

Indicates whether an interrupt is in progress or not. This flag will be set even if interrupts have been disabled.

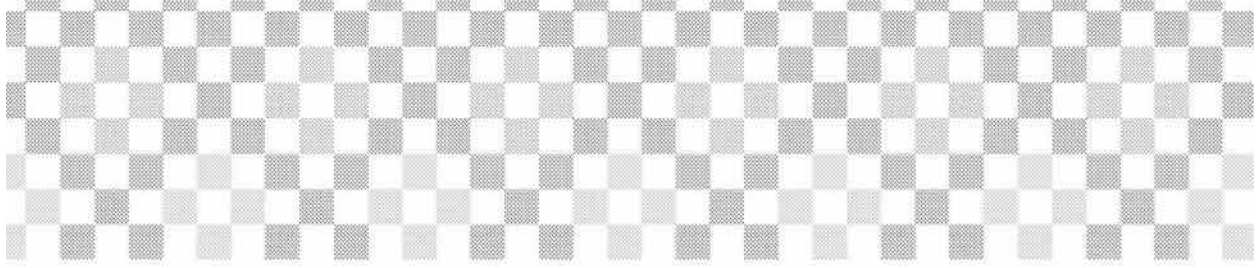


THE SLEEP INSTRUCTION

In addition to other control instructions, such as branches and jumps, there are instructions like STOP and SLEEP. The *SLEEP instruction* disables the program completely, putting it into a resting state temporarily until some input (such as an interrupt) occurs. This function exists on the system level as well.

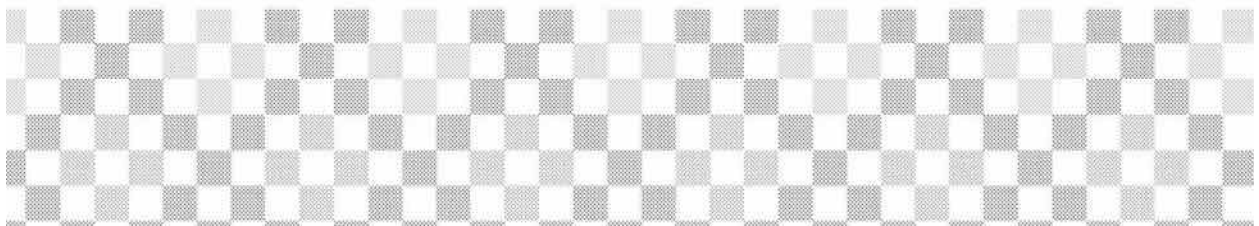
By using the SLEEP instruction, the CPU is able to slow the period of the clock and thereby the program, leading to lower power consumption. To return the CPU to its normal state, some kind of button on the device usually has to be pressed to trigger an interrupt in the CPU itself, rousing the system and programs back to full speed.





5

PROGRAMS



Assembly and High-Level Languages

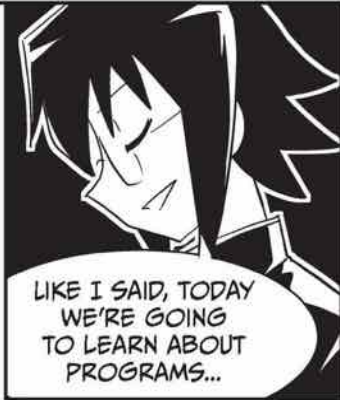




YOU'D BETTER LISTEN RESPECTFULLY TO GENIUS PROGRAMMER YUU KANO'S EVERY WORD VERY CAREFULLY!



WHAT ARE ASSEMBLY LANGUAGES?



LIKE I SAID, TODAY WE'RE GOING TO LEARN ABOUT PROGRAMS...

BUT WE SHOULD SKIP STRAIGHT TO THE CONCLUSION BECAUSE YOU ACTUALLY ALREADY KNOW ABOUT THEM!!



WHAAA, SO MY MEMORY REALLY IS BAD AFTER ALL?!

DO YOU REMEMBER THAT WE LEARNED ABOUT THESE INSTRUCTIONS USING MNEMONICS?



LDA ADDRESS 1
READ THE DATA AT ADDRESS 1 AND STORE IT IN THE ACCUMULATOR

ADD ADDRESS 2
ADD THE CONTENT AT ADDRESS 2 TO THE ACCUMULATOR

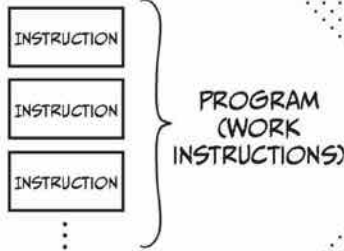
STA ADDRESS 3
STORE THE CONTENT IN THE ACCUMULATOR TO ADDRESS 3

ANY COMBINATION OF THESE INSTRUCTIONS IS ALREADY A PROGRAM (OR RATHER, THE SOURCE CODE FOR ONE*).

AH! I REMEMBER THESE!



NOW THAT YOU MENTION IT, YOU DID SAY PROGRAMS ARE WORK INSTRUCTIONS...



THAT ARE ALL LIKE A CHAIN OF INSTRUCTIONS.

* YOU CAN LEARN ABOUT THE DIFFERENCE BETWEEN A PROGRAM AND ITS SOURCE CODE ON PAGE 199.

AND WE CALL ANY LANGUAGE MADE TO WRITE PROGRAMS A **PROGRAMMING LANGUAGE**.

A PROGRAMMING LANGUAGE LIKE C USES MNEMONICS ARRAYS OF 0s AND 1s

HIGH-LEVEL LANGUAGES ASSEMBLY LANGUAGES MACHINE LANGUAGE

EASY FOR PEOPLE TO UNDERSTAND EASY FOR THE CPU TO UNDERSTAND

AS YOU CAN SEE, THERE ARE MANY TYPES OF LANGUAGES. THE ONES THAT USE MNEMONICS ARE CALLED **ASSEMBLY LANGUAGES**.

HMM... HIGH-LEVEL, ASSEMBLY, AND MACHINE LANGUAGES...

I DON'T REALLY GET IT, BUT AT LEAST IT MAKES SENSE THAT THE HIGH-LEVEL ONES ARE ON TOP!!

CLASSY CARS AND HOTELS—YOU CAN GET A LOT BY SIMPLY USING SOPHISTICATED LANGUAGE.

NO, IT'S HIGH-LEVEL, NOT HIGH-CLASS.

HIGH-LEVEL SIMPLY MEANS THAT IT'S EASY FOR PEOPLE TO UNDERSTAND AND CAN BE USED WITH ANY TYPE OF CPU.

LET'S TALK A BIT ABOUT THE DIFFERENCE...

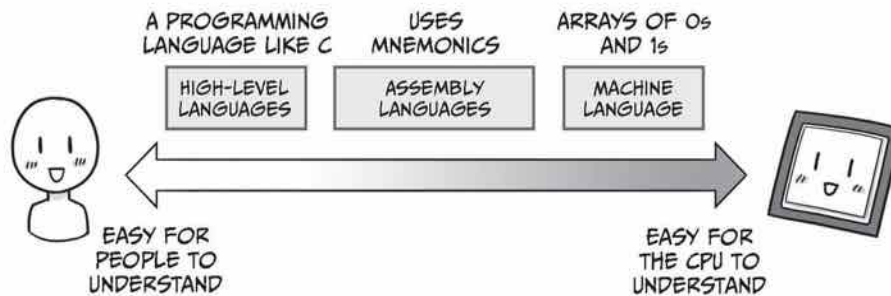
BETWEEN ASSEMBLY LANGUAGES AND HIGH-LEVEL LANGUAGES.



THE CHARACTERISTICS OF ASSEMBLY LANGUAGES AND HIGH-LEVEL LANGUAGES



Okay, let's talk a bit about the assembly languages that are easy for CPUs (machines) to understand and the high-level languages that are easy for people to understand.



Umm, I don't think I really understand what you're saying. Because machine language is made up of arrays of 1s and 0s, I see how that would only be understandable to CPUs and not people.

But wouldn't assembly languages be pretty easy for people to understand because they use mnemonics...? Instructions like ADD are just plain English...

So how could high-level languages be even easier to understand than that?!



Heh, that's a valid question. It's true that assembly languages are rather easy to understand.



But that's because you already know how a CPU works and you've learned about registers (like the accumulator), addresses, and different kinds of instructions!

With a high-level language, you don't have to care about things like registers, addresses, and instructions if you don't want to. Some high-level languages don't even let you work with low-level concepts like that. For example, if you want to add two and three together in a high-level language, you can just write, "a = 2+3"!

ADDITION IN A HIGH-LEVEL LANGUAGE

$$a = 2 + 3$$

THIS IS A VARIABLE. IT'S THE RESULT OF THE ADDITION AND IS STORED TO AN ARBITRARY LOCATION. YOU DON'T HAVE TO SPECIFY WHERE IT'S GOING (WHETHER TO A REGISTER OR A MEMORY LOCATION) IF YOU DON'T WANT TO.



Whaaa??! But that's completely different from everything we've learned so far!

So high-level languages are easy for people to understand. You're saying they let us write more intuitive instructions without having to care about how the CPU works! Is that right? If it is, that would be groundbreakingly useful, and it makes a lot of sense why it would appeal to people. It's really close to how we think.

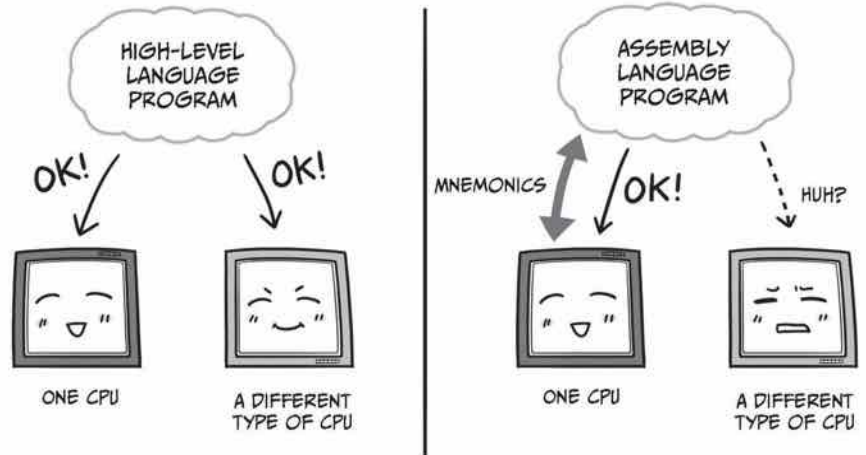


Heh heh heh. It seems you understand the appeal of high-level languages then. High-level languages are used for all large-scale program development, essentially.

There are other advantages of high-level languages, as well. Let's look at those.



Programs written in high-level languages can be used on a variety of CPUs. In contrast, assembly language instructions (represented by mnemonics) that run on one CPU probably will not run on other CPUs. They are *CPU type dependent*. Mnemonics relate directly to the instructions offered by a specific CPU instruction set, and they can't be run on CPUs that don't support that set of instructions.



THE DIFFERENCE BETWEEN HIGH-LEVEL LANGUAGES AND ASSEMBLY LANGUAGES



Hmm. I see how high-level languages are super useful. . . . But what are the advantages of using the assembly languages you've been teaching me, then?

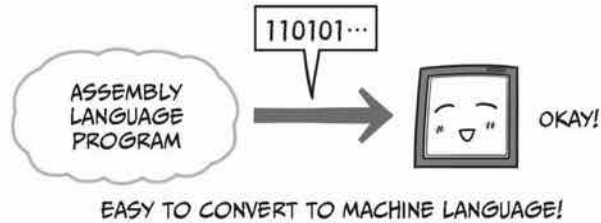
I mean, if high-level languages are this useful, why did you bother teaching me about the CPU structure and assembly in the first place? I kind of get the feeling that using high-level languages is the new way to do things and assembly languages were the old way. . . .



No, that's not true! Especially in scenarios where execution speed is paramount, assembly languages are very useful since they can push the CPU closer to its potential limit.

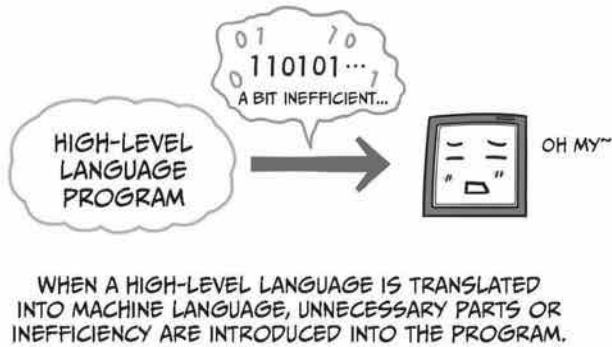


Assembly languages are essentially using mnemonics for a specific CPU instruction set, right? This means that assembly languages are easy to convert to machine language and don't waste much CPU time.



Some human-readable programming languages, like C, need to be *compiled* to convert them into machine language that the CPU can understand.

Even though the code might be easy for us to understand, it comes with a price. Because the code is being translated from a high-level language into a form the CPU can understand, it might end up executing slower* than if you had crafted it yourself in assembly code. The calculations will turn out correct, but the way that the translation system ends up performing the calculation might not be the most efficient. In the end, this means you might not be able to use all of the CPU's potential if you use a high-level language!



Ohh, I see. Using assembly languages, you can write more efficient code that uses the CPU to its full potential! Assembly languages are so incredible, they are still in use today.

* But it's worth mentioning that modern CPUs are so fast that the delay doesn't really affect us much in most cases, even if some operations do take slightly longer.



LARGE-SCALE SOFTWARE DEVELOPMENT

The computer programs that we use every day include word processors, chat programs, and spreadsheets. We call these *application programs*, or just *applications*. Creating applications requires an incredible amount of work from many programmers over an extended amount of time. We call programs like this *large scale*, and the languages used to create them are generally high-level languages. Some examples include C, the slightly newer C++, and other languages such as Java and Python.

When developing with a high-level language, you don't have to be aware of the CPU's machine language instructions in the same way that you would if you were developing with an assembly language. You also don't have to pay attention to the different addressing modes we talked about before.

Program source code that's written in a high-level language has to be compiled so it can be converted into the machine language that the CPU can execute. Since this process is automatic and tries to optimize the use of addressing modes, among other things, the developer doesn't need to rack their brains paying attention to the CPU's instruction set or the different registers or even the addressing modes, themselves.

But when writing smaller-scale device software, it is still not uncommon to use *assembly languages*. In these situations, if you don't know everything there is to know about the CPU's peculiarities, its addressing modes, and more, writing correct software will be more or less impossible.

The mnemonics you use when writing assembly code in a particular CPU's instruction set are automatically converted into binary opcodes through a process called *assembly*. In effect, you are assembling the assembly language source code into a CPU's machine language.

Today, even basic software like your operating system (Windows, for example) is mostly developed using high-level languages like C. But parts that are critical for performance may still be developed using assembly languages. This is also true for software such as simulation applications, where assembly code might be used to optimize certain parts of the program that need to be blazing fast.



THE DIFFERENCE BETWEEN PROGRAMS AND SOURCE CODE



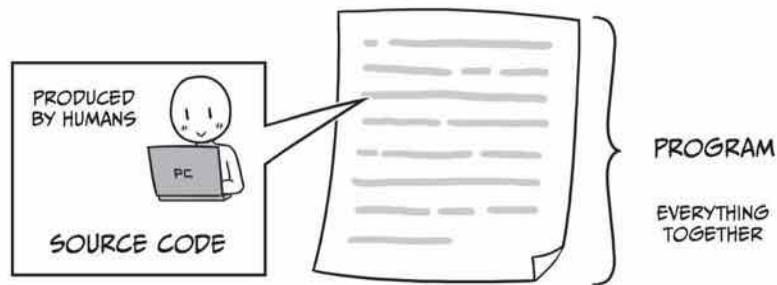
Let's see. We talked a bit about programs and source code before. The two words might seem to mean the same thing, but they are different, strictly speaking.



Hmm, it might be cool to know the difference. I'm all ears!



Sure. A *program* usually refers to the chain of instructions fed to a computer to make it do something. A program combined with all the other resources it needs to perform its task is referred to as *object code*, while the word *source code* is usually reserved for the code (machine or high-level) used to create the program.



Source code includes all of the instructions and text produced by humans, while the object code is the machine code that is produced when the source code is compiled, which is then executed by the CPU. Some recent AI (artificial intelligence) can even automatically produce source code.



Huh. I think I get it. Programs are the work instructions and their resources. Source code, on the other hand, is the instructions and text produced by humans to generate the work instructions.



You might also run into the term *source program*, but for simplicity's sake, you can just think of this as being the same thing as source code.

Program Basics

WHAT CAN YOU MAKE USING CONDITIONS AND JUMPS?



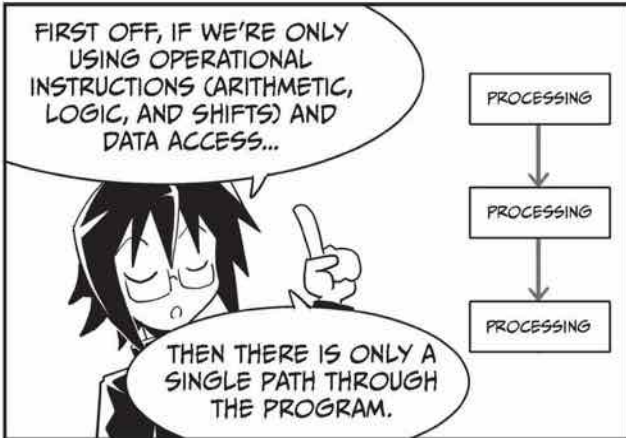
OKAY, LET'S SUMMARIZE WHAT MAKES THE BASIS OF A PROGRAM.

IT'LL ALSO SERVE AS A REVIEW OF THE THINGS WE'VE LEARNED SO FAR.



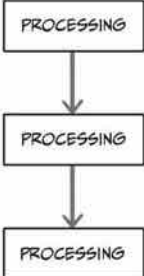
COME AT ME! I'LL SERVE BACK ANYTHING YOU THROW AT ME!

THIS IS NOT A GAME!!

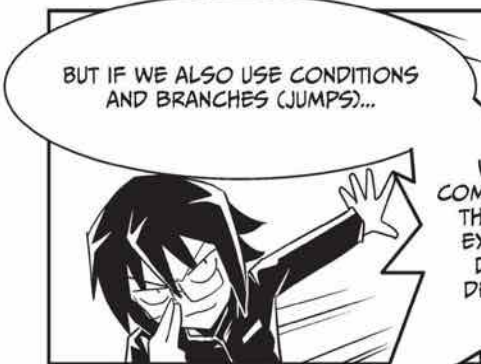


FIRST OFF, IF WE'RE ONLY USING OPERATIONAL INSTRUCTIONS (ARITHMETIC, LOGIC, AND SHIFTS) AND DATA ACCESS...

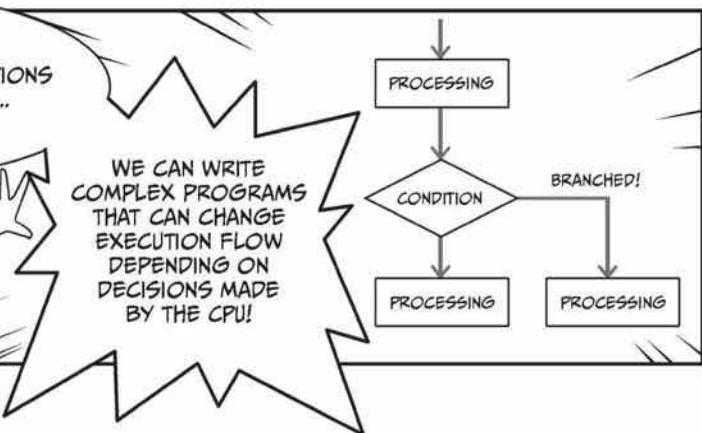
THEN THERE IS ONLY A SINGLE PATH THROUGH THE PROGRAM.



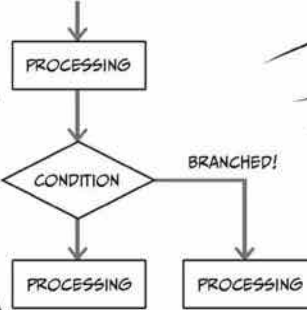
UH-HUH. IT JUST KEEPS PROCESSING ONE INSTRUCTION AFTER ANOTHER, RIGHT?

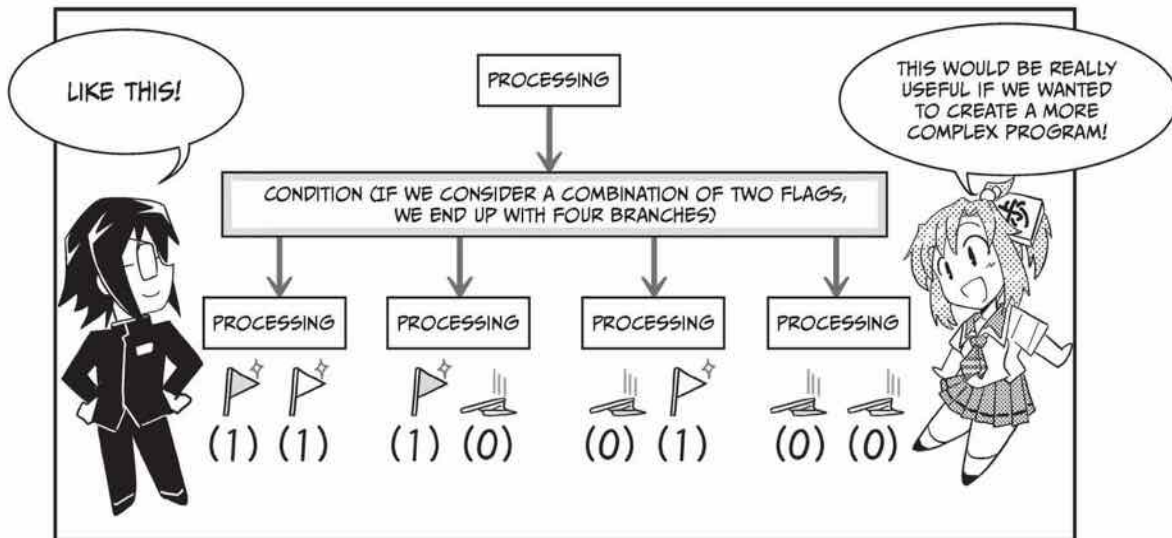
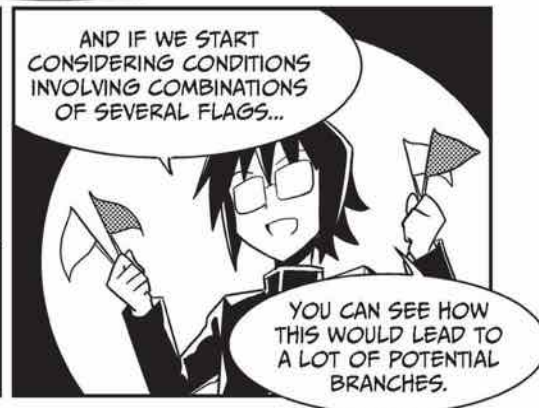
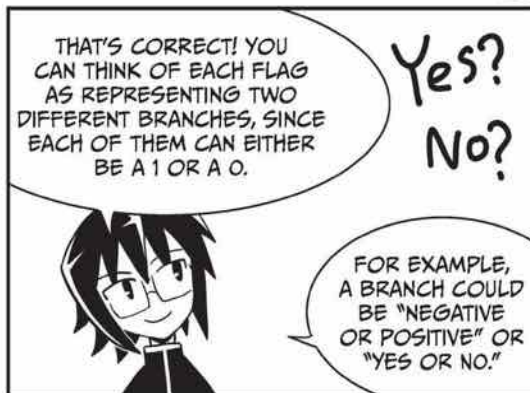
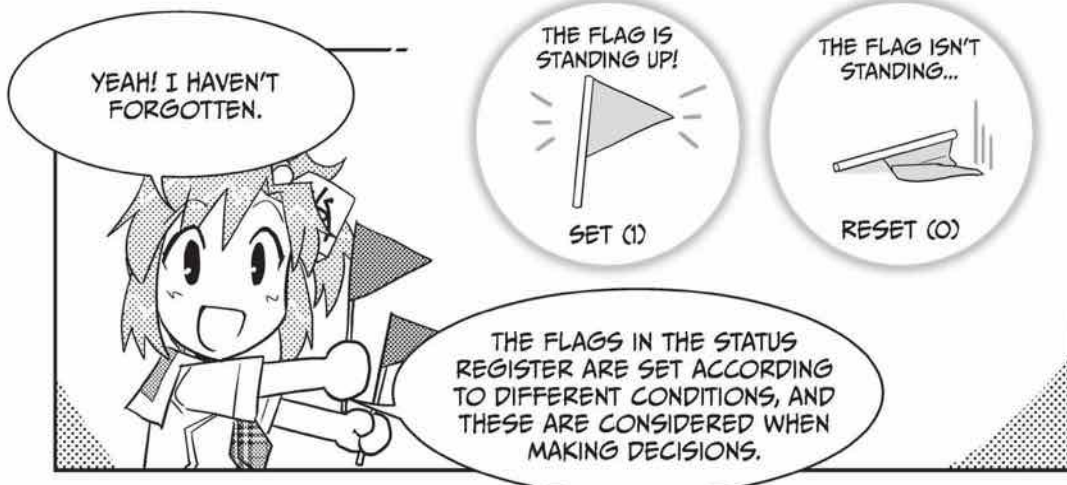


BUT IF WE ALSO USE CONDITIONS AND BRANCHES (JUMPS)...



WE CAN WRITE COMPLEX PROGRAMS THAT CAN CHANGE EXECUTION FLOW DEPENDING ON DECISIONS MADE BY THE CPU!





AND IF WE ALSO APPLY CONDITIONAL JUMPS...

IN THIS NEAT LITTLE DIAGRAM, WE REPEAT THE SAME INSTRUCTIONS OVER AND OVER.

```

    graph TD
      Start(( )) --> Processing[PROCESSING]
      Processing --> Condition{CONDITION}
      Condition -- Yes --> Processing
      Condition -- No --> End(( ))
  
```

THE CONDITION IS NO LONGER MET. ONWARD!

YOU CAN KEEP REPEATING THE SAME PROCESS IN A LOOP AS LONG AS SOME CONDITION IS MET.

WE CALL THIS A REPEATING PROCESS.

OOOH! THIS SEEMS REALLY USEFUL.

YOU CAN REPEAT A SERIES OF INSTRUCTIONS WITHOUT HAVING TO WRITE THEM MULTIPLE TIMES.

...WELL, YES, BUT LET'S STOP TALKING ABOUT THE BASIC CONCEPTS HERE.

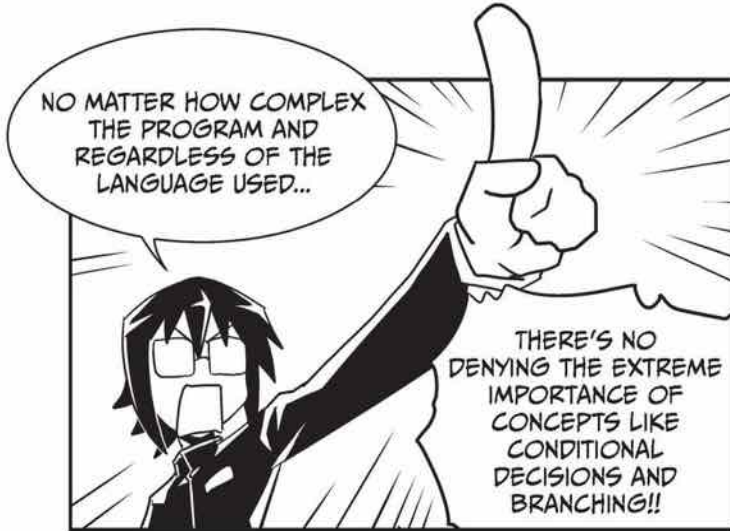
EH?! THAT'S IT??

WELL, NO. TO BE ABLE TO CREATE PROGRAMS, YOU WOULD, OF COURSE...

NEED TO HAVE A LOT MORE IN-DEPTH KNOWLEDGE.

HOWEVER!

?





WHAT SHOULD WE MAKE THE COMPUTER DO?



By the way, we've been learning about programs today, but try to remember when we talked about digitization of information. In modern society, by digitizing things like music, images, video, and any other type of information, it becomes a lot easier to process information using a computer.



Ah, I remember us talking about something like that. Now that I think about it, it's actually kind of amazing!

I mean, if computers can handle any type of information, then you could do all kinds of things if you just created an amazing enough program!



Yeah. It's just as you say. Things that weren't even conceivable in the past are gradually becoming reality today.

One good example is the facial-recognition software used in some modern security applications. These programs convert human facial features (such as the distance between the eyes, the position and size of the mouth and nose, and so on) into numerical values and use them for calculations. Some programs can then differentiate among human faces by using this information.



I see. It feels a bit like science fiction that computers are actually able to tell people's faces apart. It might even be a bit scary. But on the other hand, it could be used for fighting crime.

It seems like it might be a lot of fun to create a really cool program. I wonder what I would have it do. Maybe stock market or horse race predictions . . . ? Some program that would automatically make me a lot of money. . . .



Ah! Let's put your personal desires aside for now. But thinking about what you want to make your computer do and what people would find useful are two very important aspects of creating a program.



THANKS FOR TODAY.
I LEARNED A LOT.

SURE, DON'T
WORRY
ABOUT IT.



UM... BY THE WAY...
I ASKED MOM ABOUT WHEN
WE WERE KIDS...



WELL YOU ALWAYS
BEAT YUU IN SHOGI...



YOU'RE WEAK.
I'M BORED.

EVEN IF HE
STARTED CRYING...



AND THEN YOU'D
SAY SOMETHING
LIKE...

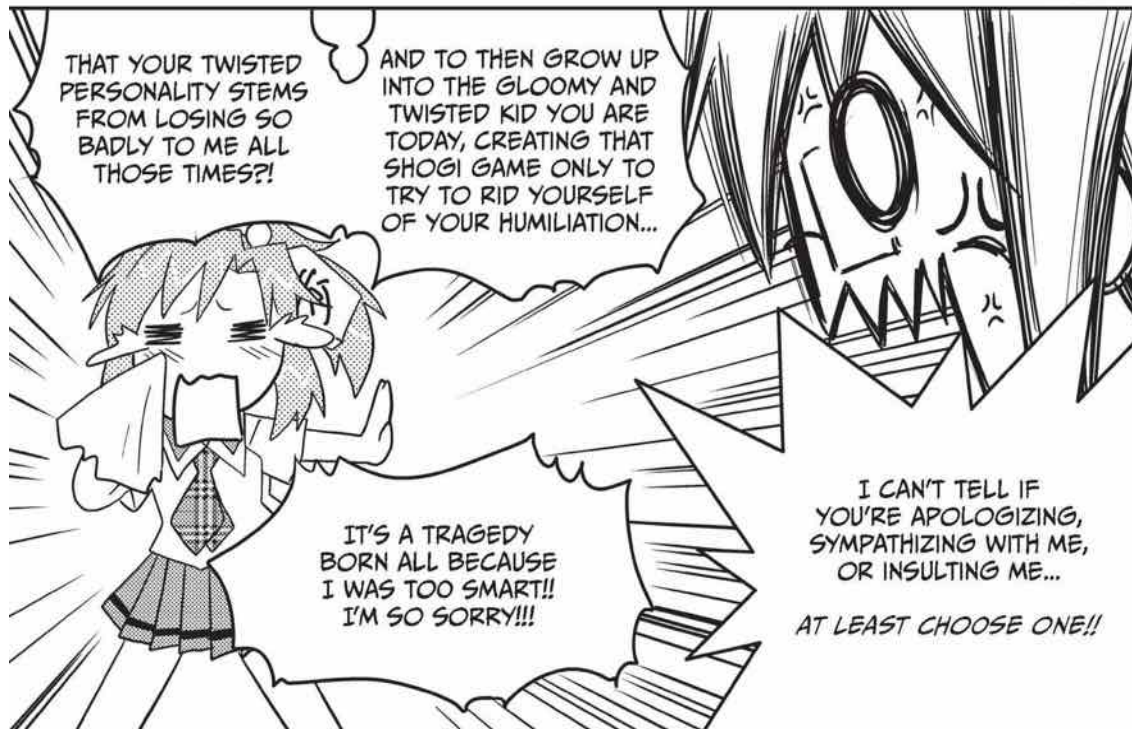
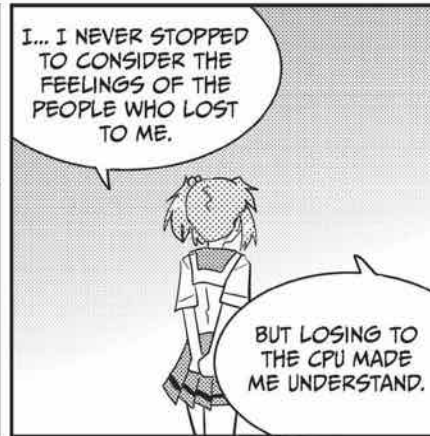


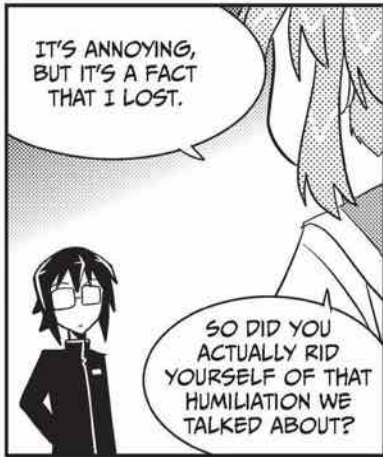
YOU WERE PRETTY
RUTHLESS.

THAT
BAD...?



EVEN THOUGH IT'S
IN THE PAST NOW,
I DO FEEL KIND
OF BAD ABOUT IT...





WHERE ARE PROGRAMS STORED?

Programs for small devices that use microcomputers are usually stored in ROM. In personal computers, only the BIOS (the Basic Input/Output System) is stored in ROM, which in turn is used to load the operating system (OS) from an external device (such as a hard drive) into RAM. Programs are also loaded into RAM before execution.

At the dawn of the CPU era some decades ago, miniature OS-like systems called machine code monitors were used when developing assembly code line by line.

Nowadays, even assembly programming is done on personal computers. Each CPU maker provides development tools to allow programmers to more easily develop assembly language programs for their CPUs. You can create your program using these tools on your computer, attach a ROM writer to the system to embed your program into ROM, and finally integrate the ROM into your target system.

A more recently developed method allows programmers to transfer the program from a computer to the device's non-volatile memory. This saves a lot of time because you can check how the device performs without having to rewrite the ROM every time.



It's also worth mentioning that the method of rewriting a program on CPU ROM without detaching it from the system is called *on-board programming*.

WHAT HAPPENS BEFORE A PROGRAM IS EXECUTED?

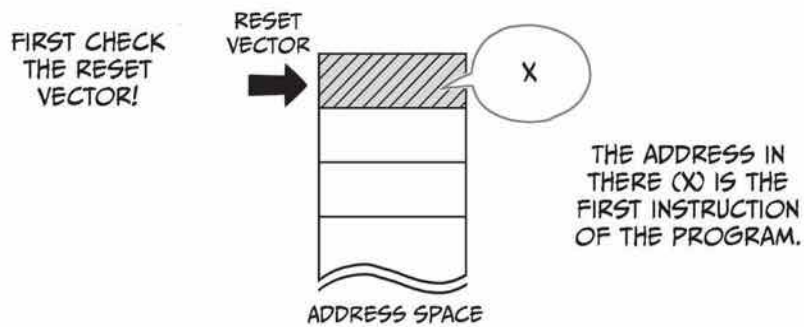
Let's talk a bit about what happens when you load a program you've written into ROM. What does the CPU do as soon as you turn on the power?

Simply turning on the power doesn't actually do anything, as there is a significant risk that the system will not perform as expected before the voltage has climbed to a certain level. To ensure that the CPU will operate properly, the circuitry on the CPU board must keep the reset pin low until the power supply voltage stabilizes and the CPU's clock generator starts functioning normally.

The clock generator normally starts operating before the power supply voltage stabilizes, so when the power supply voltage reaches the correct level, the CPU board's reset circuit sets the reset pin to high, and the CPU can begin executing instructions. The voltage needed for this is generally specified in the CPU's documentation.

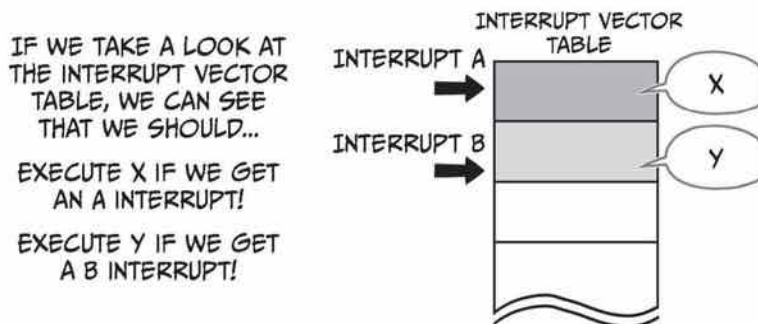
At this point, all the preparations are done for loading the first line of the program. After releasing the reset state, the first thing the CPU does is load a *reset vector*.

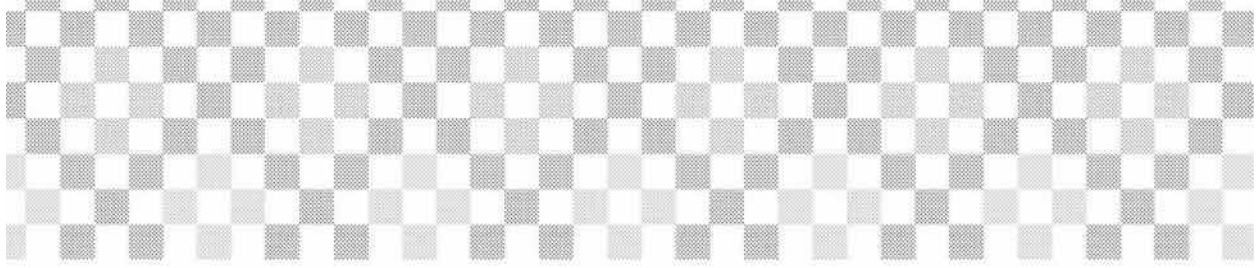
The reset vector is usually written to the first or last part of the memory the CPU manages, and it tells the CPU where to find the first instruction of the program to run after a reset. For a PC, this would be the BIOS.



The CPU will then run the instruction at the address specified by the reset vector and proceed normally from that point. It would execute the program the instruction belongs to, perform calculations, and process data in accordance with the program flow. If the reset pin were to become active for any reason, the CPU would instantly cease all activity, no matter what it was currently working on, and return to its initial state.

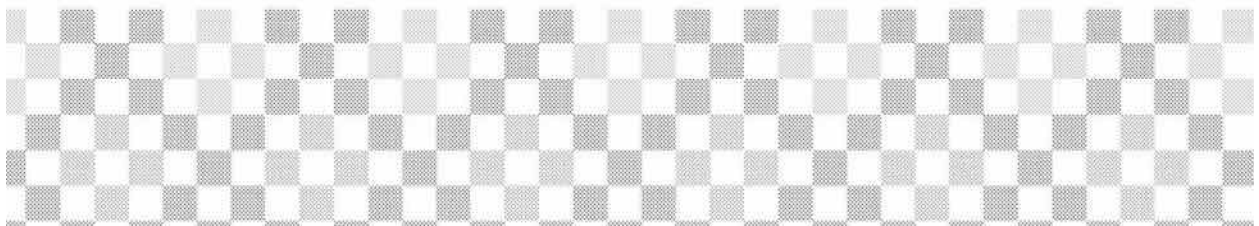
A reset is actually a type of interrupt, like the ones we learned about in earlier chapters. Although we learned previously that interrupts can stop a CPU from running its current set of instructions and make it run instructions at another address, we haven't learned how the CPU knows which address to jump to. Each type of interrupt has an address associated with it, and the data structure that stores the addresses to execute depending on which type of interrupt occurs is called the *interrupt vector table (IVT)*. The reset vector is at a set location in memory, and it is the first value in the interrupt vector table. That's how it works at a very high level, but IVTs vary from CPU to CPU, so the location of the reset vector will depend on the CPU's specifications.





6

MICROCONTROLLERS



What Are Microcontrollers?



AHH~
WHAT WONDERFUL
WEATHER!



NNNGH!

LEARNING
BENEATH AN
OPEN SKY ISN'T
BAD AT ALL.



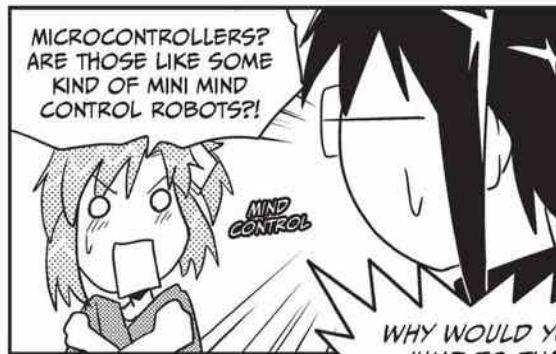
BUT TO THINK
A SHUT-IN LIKE
YOU STUDIES AT
THE PARK.

WELL,
I SUPPOSE
IT'S NOT FAR
FROM HOME.



...HEH, IN ANY
CASE, THIS WILL
BE OUR LAST
CLASS.

TODAY'S THEME IS
MICROCONTROLLERS!



MICROCONTROLLERS?
ARE THOSE LIKE SOME
KIND OF MINI MIND
CONTROL ROBOTS?!

MIND
CONTROL

WHY WOULD YOU
JUMP TO THAT
CONCLUSION?!

MICROCONTROLLERS ARE IN ALL KINDS OF PRODUCTS

AHEM.

"MICRO"
+
"CONTROLLER"

(THERE ARE ALSO MICROCOMPUTERS!)

AS THEIR NAME SUGGESTS, MICROCONTROLLERS ARE SMALL CONTROLLER CHIPS.

I'M NOT SURE I UNDERSTAND FROM JUST THE NAME.

WHAT DO THEY CONTROL EXACTLY?

ARE THEY DIFFERENT FROM THE CPUs IN COMPUTERS?

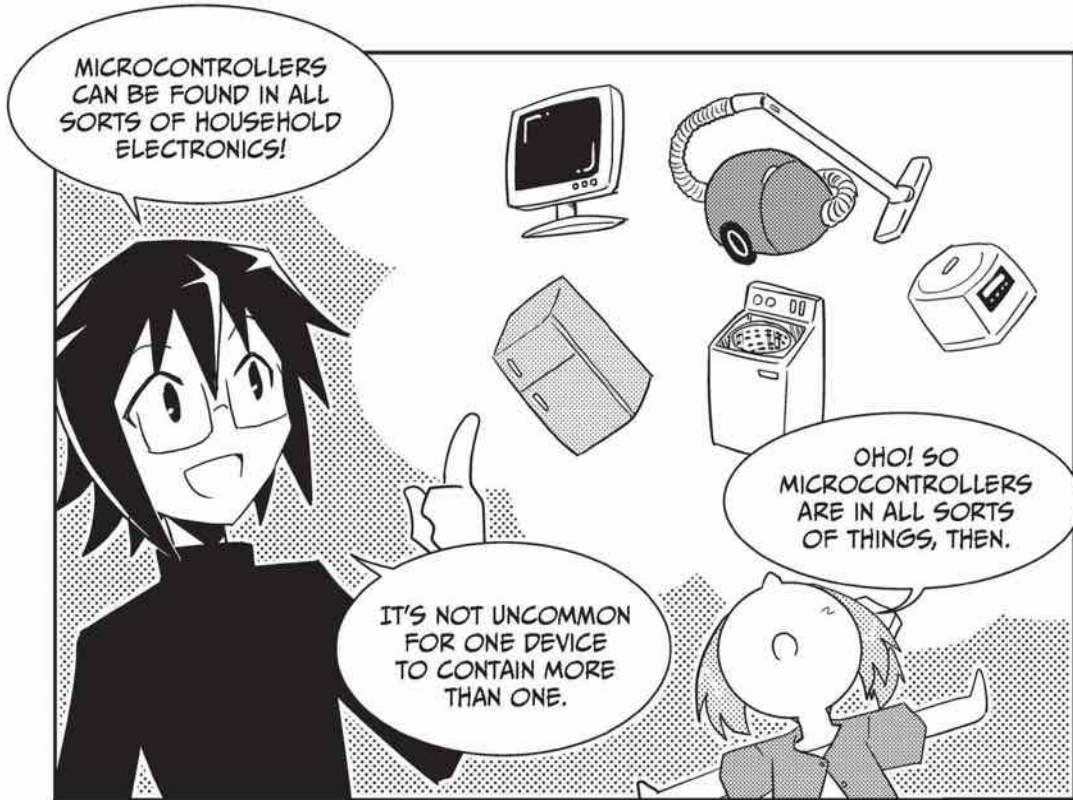
MICROCONTROLLER



THERE ARE ALSO LONGER MICROCONTROLLERS, SUCH AS THE ONE SEEN ON PAGE 49.

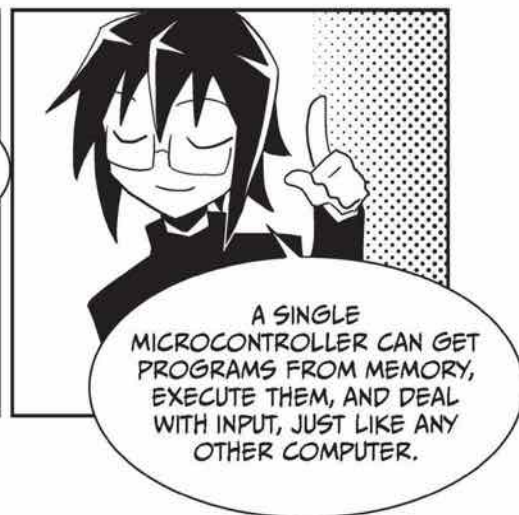
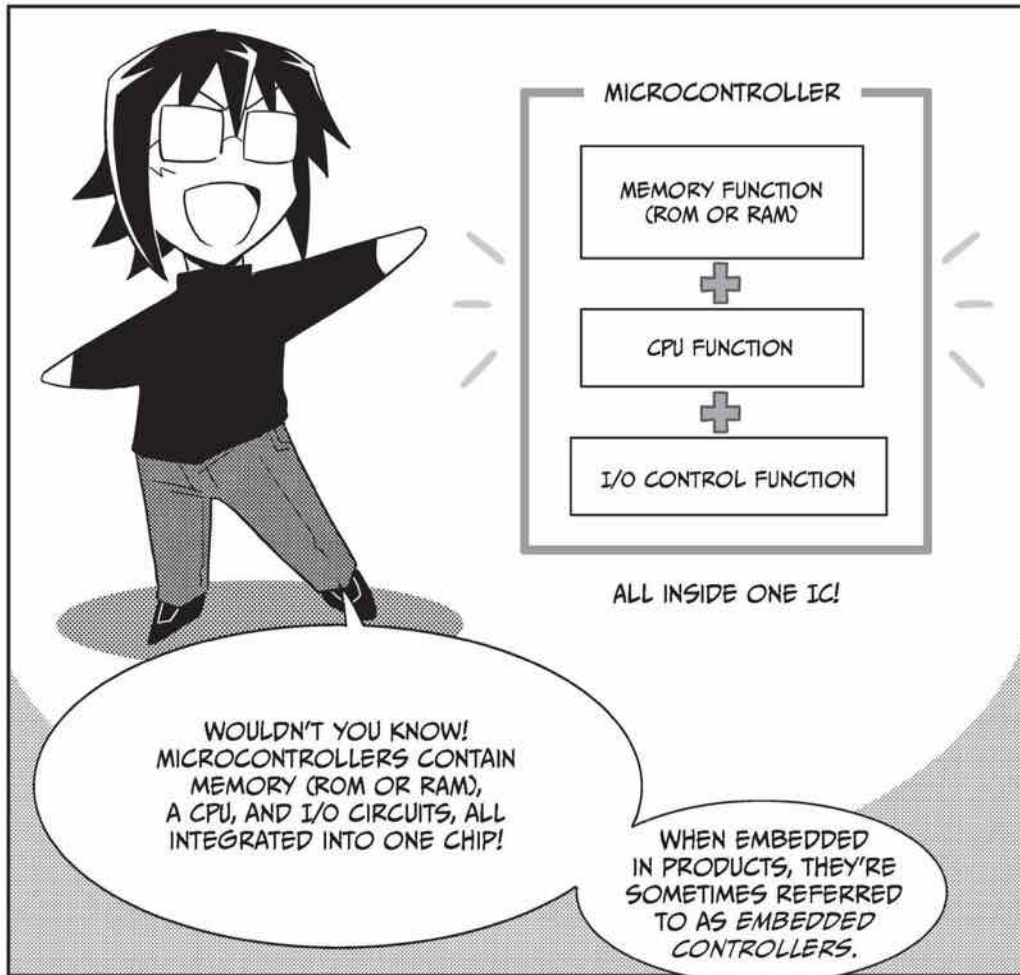
HERE, TAKE A LOOK AT THIS.

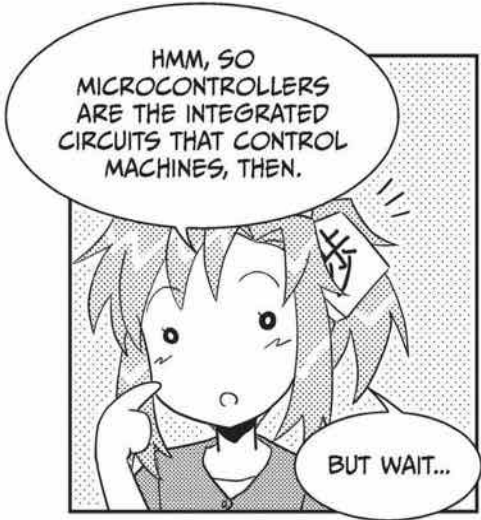
MICRO-CONTROLLERS ARE SINGLE INTEGRATED CIRCUITS (ICs), LIKE THIS ONE.



THE FUNCTION OF A MICROCONTROLLER




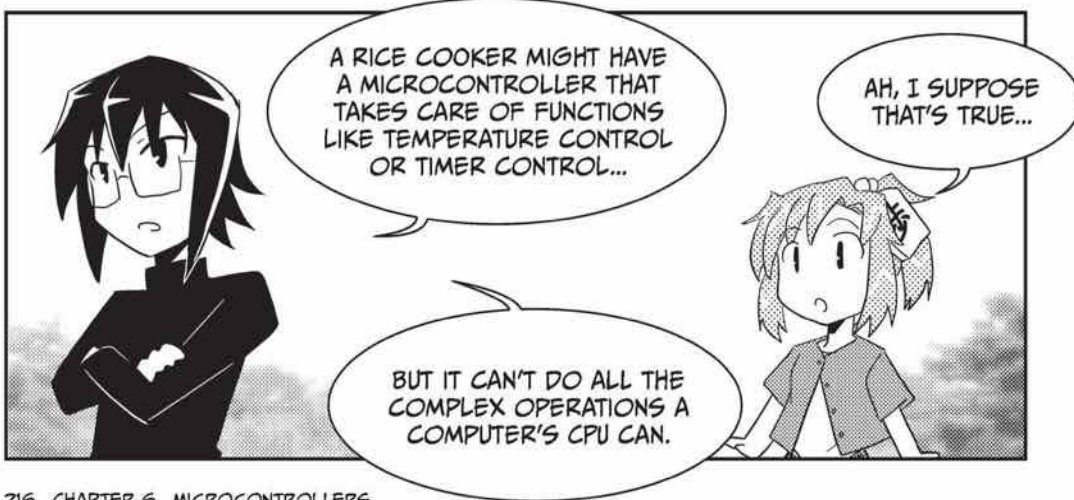


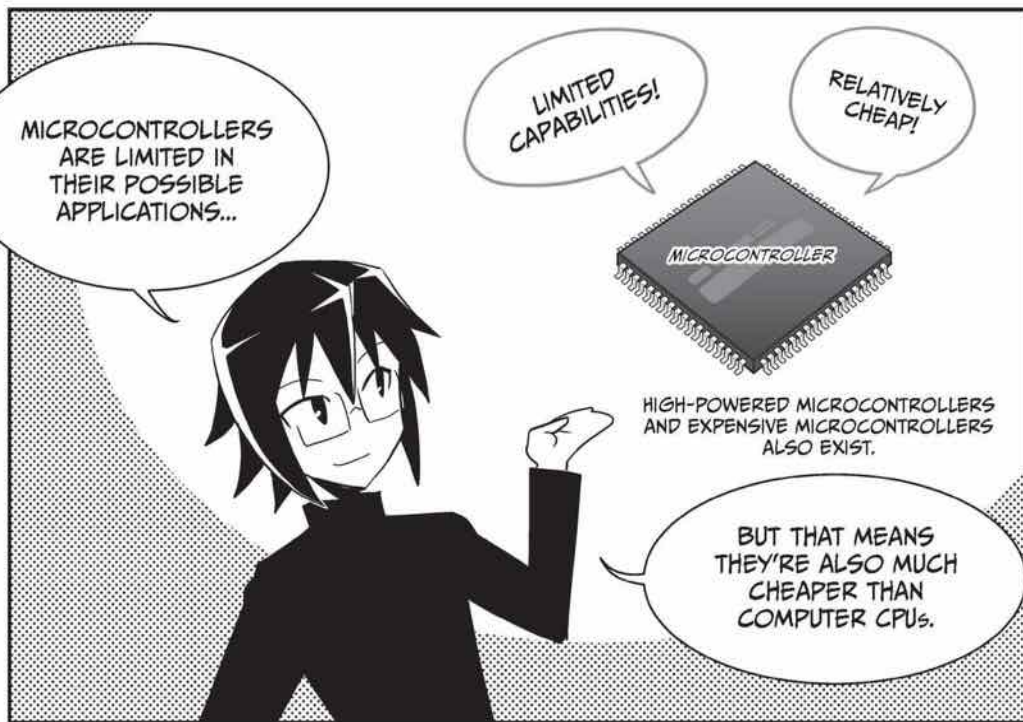
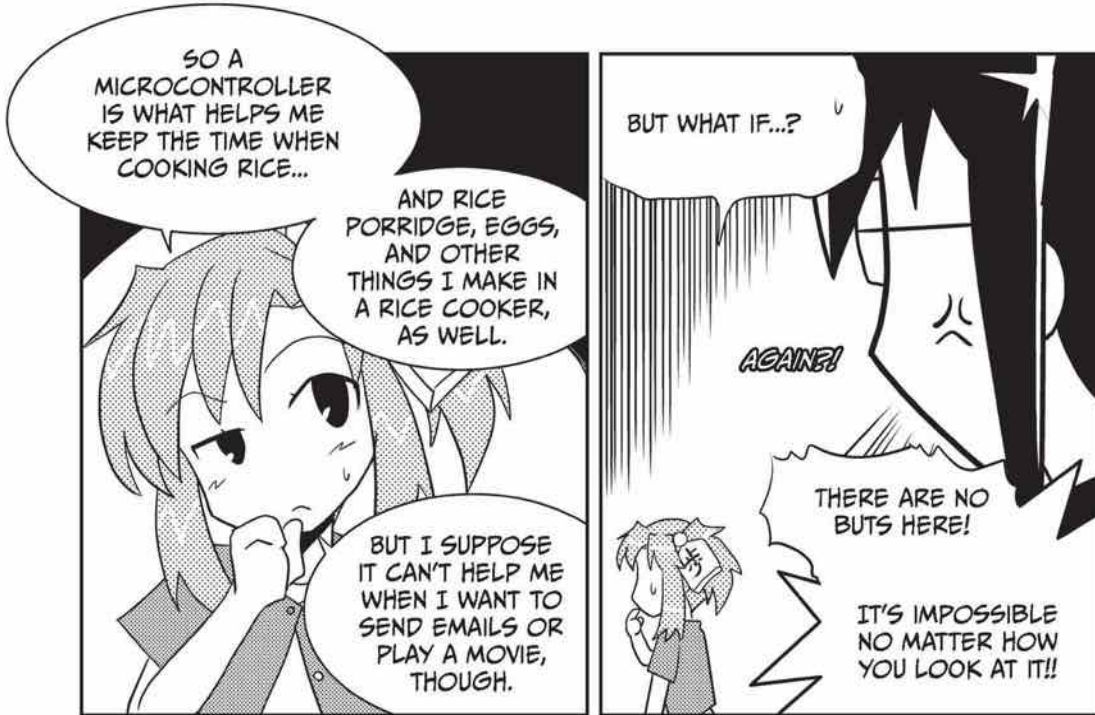


BUT MICROCONTROLLER CPUs AND COMPUTER CPUs ARE COMPLETELY DIFFERENT!

EXAMPLE FUNCTION OF A MICROCONTROLLER CPU

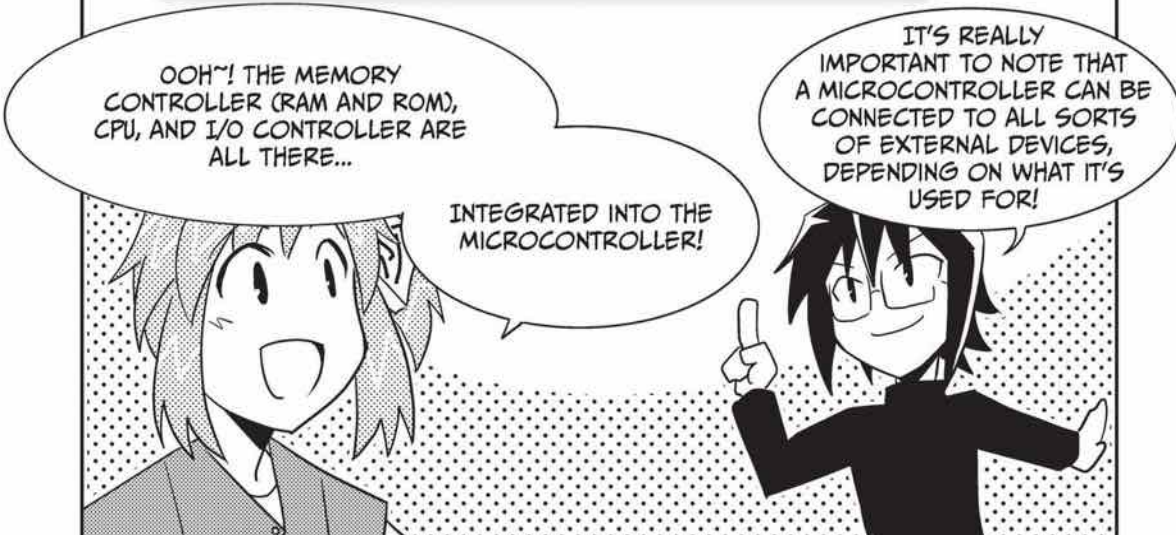
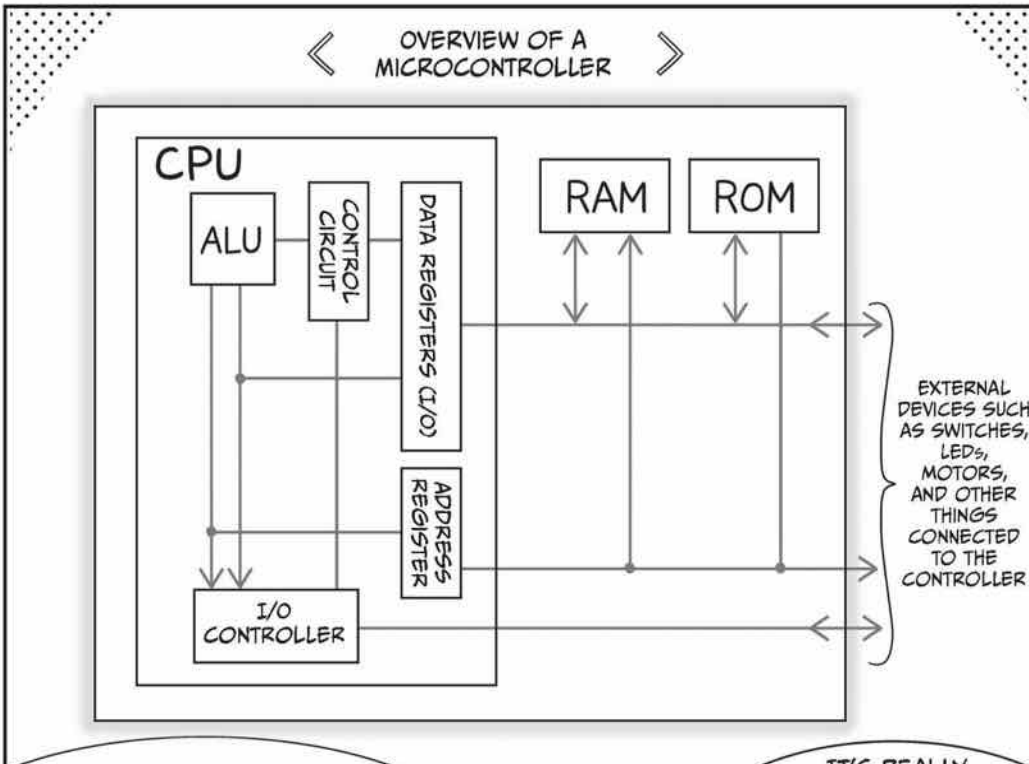
TEMPERATURE CONTROL	MICRO-CONTROLLER	TIMER CONTROL
TO MAINTAIN 70°C, FOR EXAMPLE...		TO ACTIVATE AT 6 PM, FOR EXAMPLE...







ARCHITECTURE OF A MICROCONTROLLER





HISTORY OF THE MICROCONTROLLER

Microcontrollers have a very interesting history. The very first electronic calculator used an ALU made with electron vacuum tubes. It was very large by today's standards, taking up an entire room the size of a volleyball court. This was around the time of World War II, in the 1940s. England and other countries were in the middle of developing electronic calculating machines to decipher the codes that the German army was using. Unlike modern computers, these were not developed in an open international marketplace but instead in top-secret military research laboratories. Because of this, it's not clear whether the American ENIAC computer, presented to the world in 1946, was truly the first computer or not.

The first transistor was invented in 1947, displacing electron vacuum tube calculators with semiconductor-based technology. And with the invention of the integrated circuit in 1958, the miniaturization of electronic calculators progressed significantly.

Even so, the 16-bit minicomputer seen in Chapter 4, which used four 74S181 chips, was not developed until 1972. Removing all external devices, it had a width of 60 cm, a height of 30 cm, and a depth of 45 cm. It could handle an address space of 16KW (kilowords, where 16 bits of data is equal to 1 word), which is what we would call 32KB today. (Compare this to modern SD cards, which are able to store 32GB of data or, in terms of memory size, about a million times more data.)

In the second half of the 1970s, Intel introduced its one-chip CPU as a mass-produced commodity. This product instantly dominated the market, lowering costs across the board.

The one-chip CPU also heralded a new age from 1976 onward in which amateur electronics enthusiasts could afford learning kits (for example, the Altair 8800 microcomputer). With these, they could learn how to program in machine language with the one-chip CPU as a base.

This is also around the time when the term *personal computer*, or *PC*, came into popular use, since you could finally have one of these technical marvels for yourself.



But progress did not stop there! One-board controllers that fit the CPU, memory, and I/O controllers onto a board about the size of an A4 sheet of paper were developed. Eventually, these components fit onto a single IC chip that could be used to control all sorts of devices.

Because these ICs had CPUs that were inseparable from their ROM or RAM memory and integrated I/O ports, they could store programs, perform calculations, and handle input and output. These ICs were called microcontrollers since they were comparatively small and controlled other devices

After this, there was a desire for larger memory, which stimulated a demand for larger computers, as well. But the small-scale devices that use integrated one-chip microcontrollers are still in very high demand today and are used in everything from household electronics to toys.

And so microcontrollers contain everything from a CPU to memory to I/O controllers.





WHAT ARE DSPs?



We should take this opportunity to talk a bit about DSPs, as well.



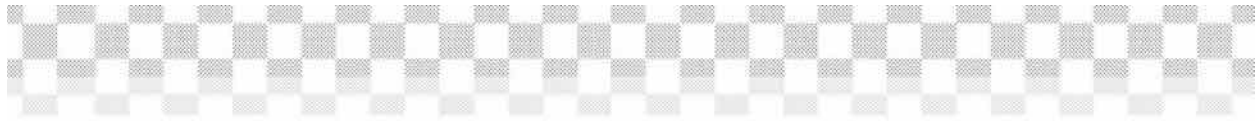
DSP? Yet another strange acronym. So, what are they?



DSPs, much like CPUs, are ICs that perform operations. But compared to CPUs, they're extremely fast.



Their "brain" is made up of what is called a *multiplier-accumulate operation circuit*. This essentially means DSPs are really good at doing multiplication and addition at the same time!



Whoa! So what's that good for? I suppose it has to be good for something.



Yeah. It turns out you have to do a lot of multiplication and addition when processing digital signals, especially for audio recordings. In fact, that's what DSP is short for—*digital signal processor*.



Audio . . . so . . . when I'm talking on my cell phone, for example? I suppose my analog voice has to be converted to digital form somehow for it to be transmitted to my friend on the other end of the line.



That's correct! Most modern mobile phones have DSPs. They're also being used more often for digital filtering in audio devices and for the operations required to improve acoustic effects.



Wow, so they're all around us, these DSPs!



Currently, development for a one-chip DSP with large RAM at the level of a microcontroller is underway.



I see. So they're good at doing multiplication and addition at the same time, which is useful for digital signal processing.

While CPUs are really important, I get the sense that DSPs are, too. I'll make sure to remember them!

DSPs AND MULTIPLIER-ACCUMULATE OPERATIONS

During the development of the CPU, there was a growing need for increasing the processing speed of calculations—in particular, division and multiplication were a lot slower than desired.

As we've seen, the CPU's ALU was mostly geared toward performing addition and subtraction at this time. Using these older ALUs, you had to repeatedly perform addition to multiply two numbers and repeatedly perform subtraction to divide two numbers. At the time, computers were becoming more popular in scientific applications, which meant that demand for high-performance multiplication was very high. This is when development of the circuits that were capable of floating-point multiplication really took off, and the result was the digital signal processor, or DSP.

To process digital signals, DSPs perform fast Fourier transforms (FFTs) on them. This requires a lot of simultaneous additions and multiplications. To perform these multiplications and additions efficiently, DSP ICs have a multiplier-accumulator circuit.

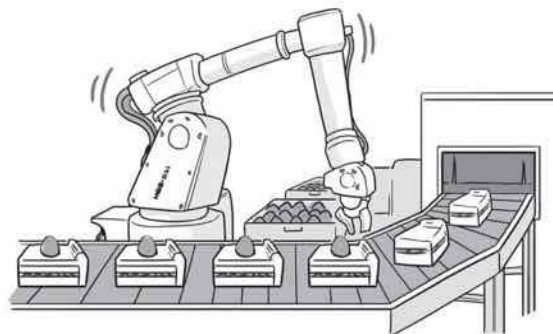
Shortly after DSPs were developed, mobile phones started to use digital transmissions, and digital voice signal processing and filtering became more common. The transmission stream could also be compressed, and the receivers started using systems with DSPs at their core to convert the raw voice data using vocoders (a voice encoder/decoder).

Later, microcontroller-like DSPs with larger RAM molded into the chip started to show up, making voice data processing even faster.

MICROCONTROLLERS IN INDUSTRIAL MACHINES

CPUs, microcontrollers, and DSPs are in many of the devices we use in our daily lives. Some examples include today's wall clocks, alarm clocks, and even wristwatches, all of which are likely to contain a one-chip microcontroller. Other household devices like refrigerators, air conditioners, and washing machines are likely to contain more than one. And the remote controls used to send commands to these devices also contain a microcontroller IC.

Automated robots and conveyor belts in large-scale industry also have to be controlled in some fashion and therefore require either a CPU or DSP.



MANY DEVICES IN OUR HOMES HAVE MICROCONTROLLERS,
AND THEY EVEN HAVE INDUSTRIAL USES...

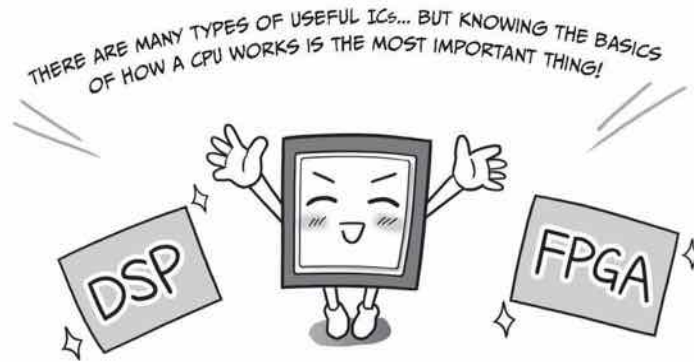
Up until now, we've had single-chip CPUs as well as microcontrollers that contain a CPU, memory, and I/O controllers all in one. The amount we can pack into a single chip is determined by tradeoffs among our capacity to produce semiconductors, their production cost, and the market's demand and margins.

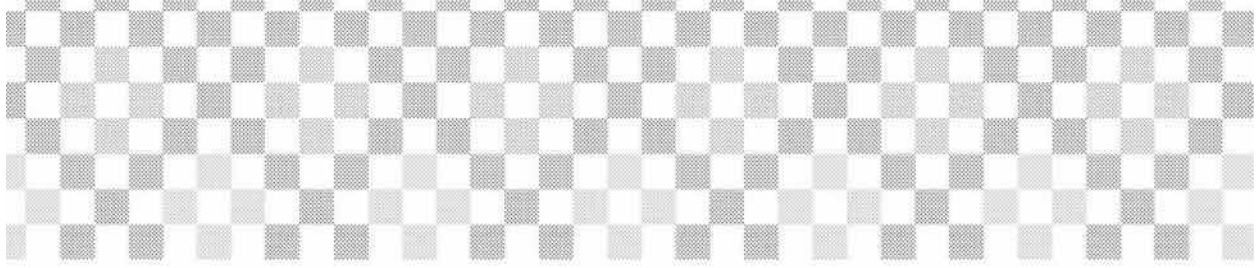
Advancements in IC production technology have led to the development of FPGAs (field-programmable gate arrays). Using an FPGA, you can create any logical circuit you desire and bring it into reality with hardware. The basic structure can contain lookup tables of anywhere from several thousand to several million units in a single IC. These tables can be prepared beforehand by the IC maker and provided as is.

The initial state of the IC consists of a lookup table memory section and logic blocks that can be wired together in different ways, resulting in many possible applications. The raw IC is then configured by the user with specialized tools to write the modeled design into the circuit, creating the desired IC. Development is normally done on a computer, but the specialized tools needed can also be stored on a USB, making the creation of even large-scale logic circuits an easy task.

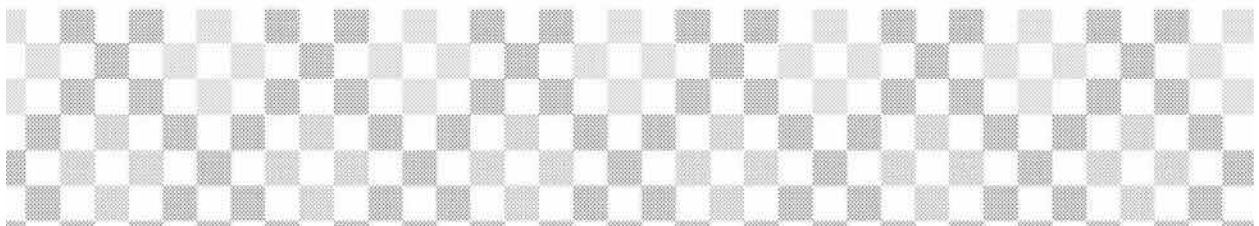
In the past, CPUs were different from FPGAs, but in recent years, FPGAs containing CPU functionality have started to appear. There are two ways to make an FPGA into a CPU. The first is to simply create an existing CPU design in the wiring of your logic circuit using the development tools provided, and the other is to embed a CPU in one part of the gate array as a separate IC.

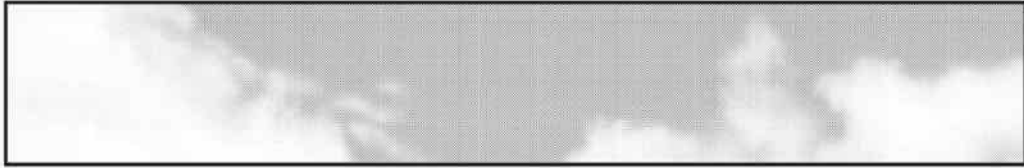
In either case, the CPU as a single IC is becoming gradually less common over time. But even though we have DSPs, there is still a need to control tasks other than multiply-accumulate operations, and as such, the principles underlying the CPU will remain important concepts no matter how ICs continue to develop.





EPILOGUE





I GUESS THIS MARKS THE END OF OUR LESSONS.

I FEEL CONGRATULATIONS ARE IN ORDER SINCE YOU MADE IT ALL THIS WAY.

OH, THAT REMINDS ME... I PROMISED TO RETURN THIS AFTER WE WERE DONE, DIDN'T I?

HERE YOU GO, ONE HOSTAGE LAPTOP.

THE ENTIRE REASON I STARTED LEARNING ABOUT CPUs WAS BECAUSE I LOST TO THIS THING.

IT IS.

BUT THE SHOOTING STAR ISN'T JUST STRONG.

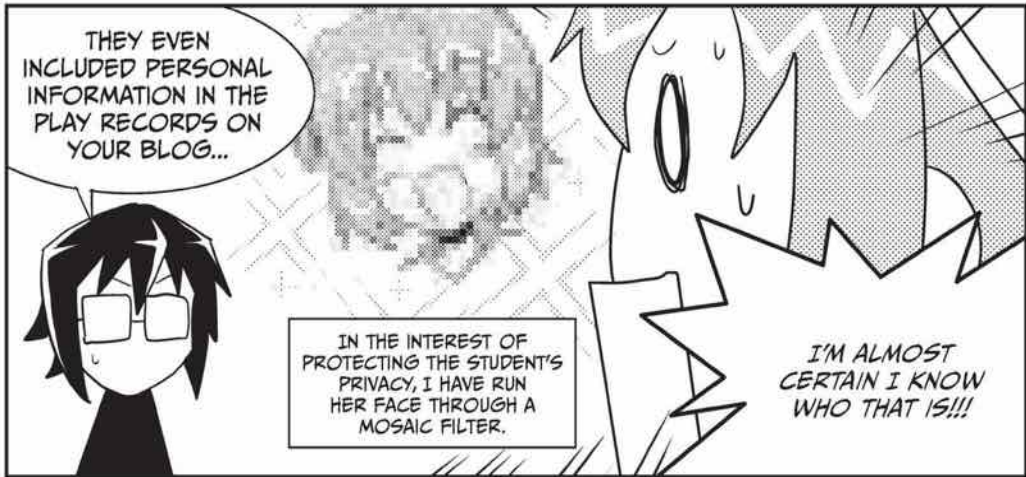
IT'S ESPECIALLY STRONG AGAINST YOU.

I HATE TO ADMIT IT, BUT THIS LITTLE GUY IS REALLY STRONG!



UH, WHAT DO YOU MEAN?

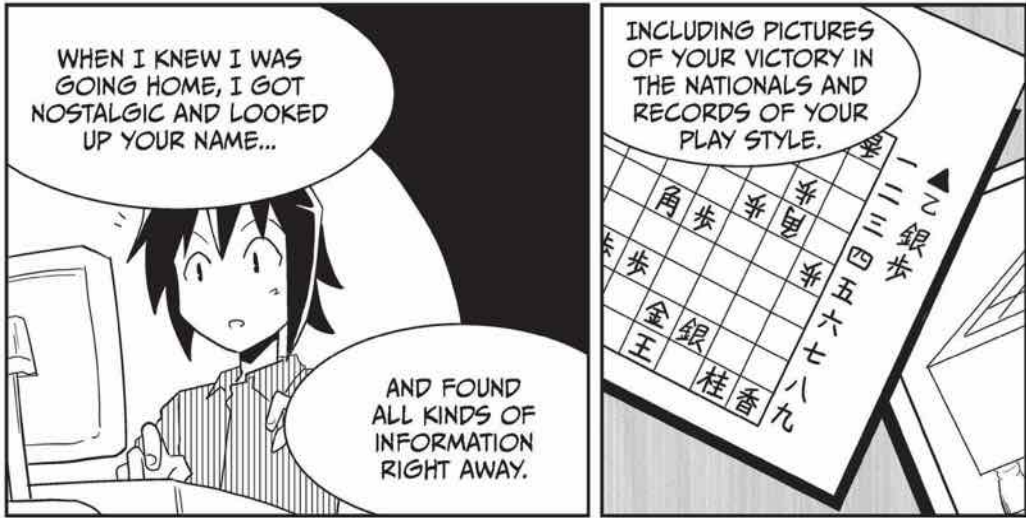
WELL, YOU SEE... YOU KNOW HOW ONE OF THE UPPERCLASSMEN IN YOUR CLUB LIKES TO POST YOUR PLAY DATA ON THE WEB?



THEY EVEN INCLUDED PERSONAL INFORMATION IN THE PLAY RECORDS ON YOUR BLOG...

IN THE INTEREST OF PROTECTING THE STUDENT'S PRIVACY, I HAVE RUN HER FACE THROUGH A MOSAIC FILTER.

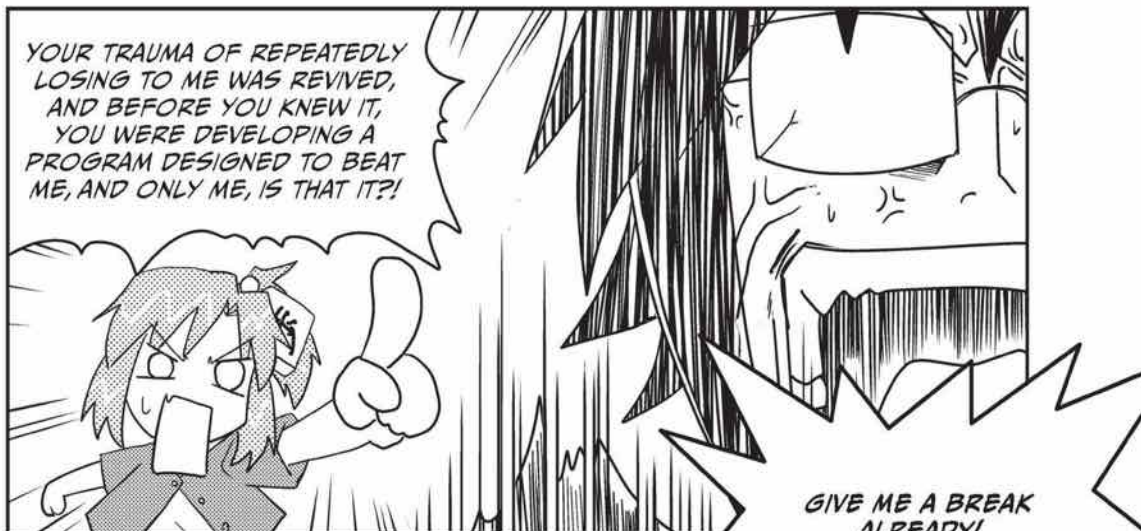
I'M ALMOST CERTAIN I KNOW WHO THAT IS!!!

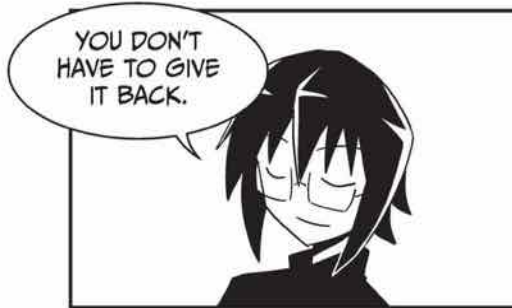


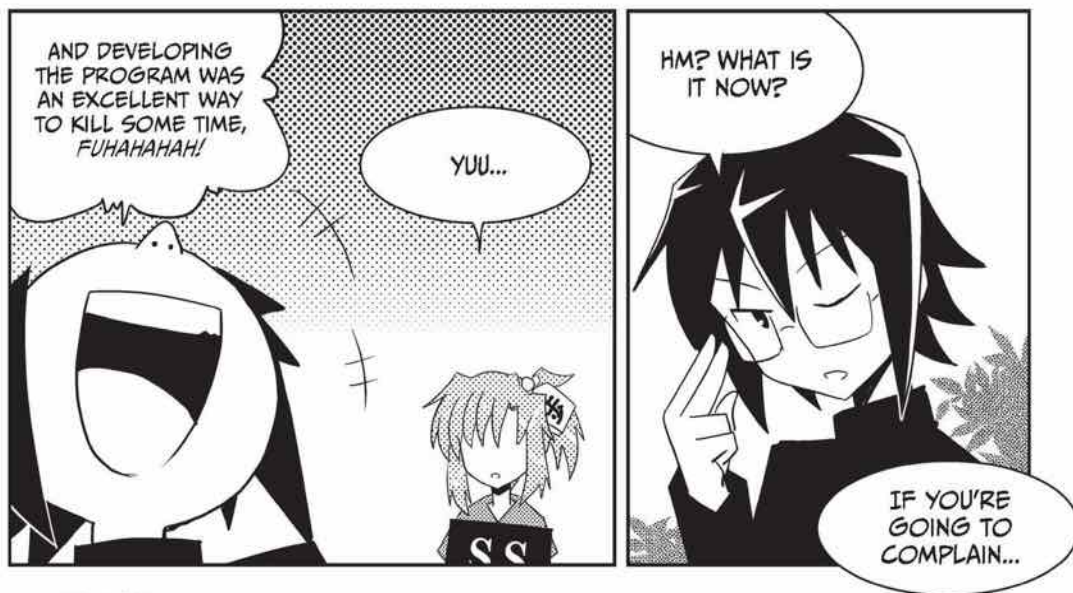
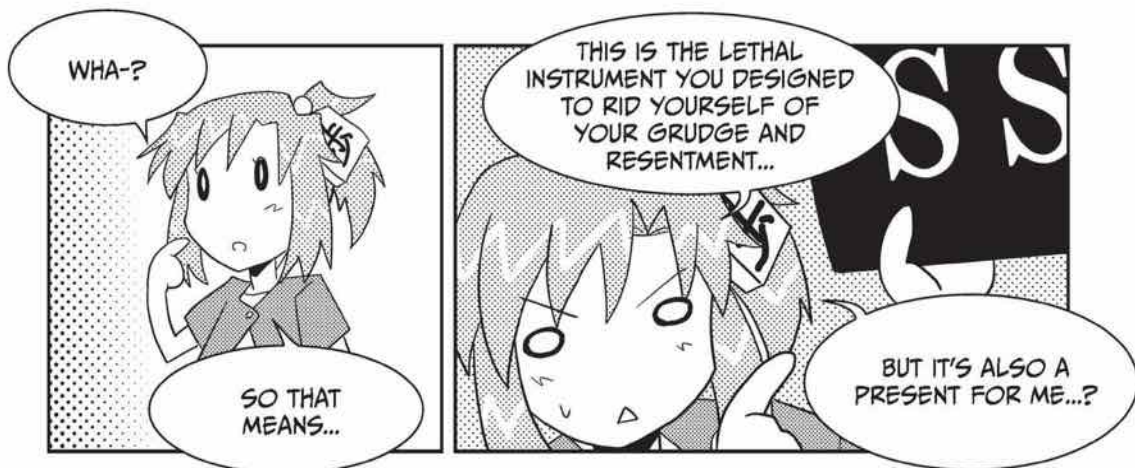
WHEN I KNEW I WAS GOING HOME, I GOT NOSTALGIC AND LOOKED UP YOUR NAME...

AND FOUND ALL KINDS OF INFORMATION RIGHT AWAY.

INCLUDING PICTURES OF YOUR VICTORY IN THE NATIONALS AND RECORDS OF YOUR PLAY STYLE.

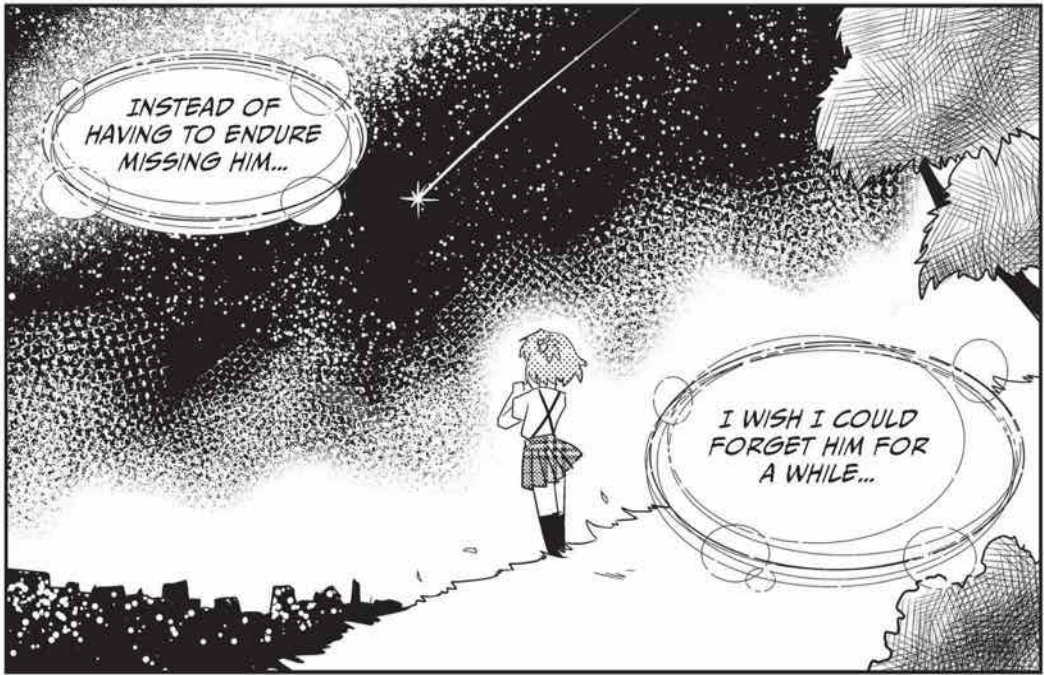














I REMEMBER...

GAAH!

EVERYTHING!!!



I-I-IT DOESN'T
MATTER ONE BIT!!

...!!

A-ANYWAY
WHAT DID YOU
WISH FOR?!

WELL,
YOU SEE...



I WISHED THAT I
WOULD RETURN
HERE SOMEDAY...

TO TALK A LOT
MORE WITH
AYUMI.



THAT I'D GET A LOT
BETTER AT SHOGI,
OF COURSE... YEAH!



IF YOU KEEP
PRAYING TO THE
STARS FOR THAT,
YOU'RE ONLY GOING
TO GET WORSE,
YOU KNOW.

AND BESIDES,
IT'S NOT LIKE I
LOST TO YOU—
I LOST TO
THE CPU!!

WE'RE BACK TO
WHERE WE STARTED!!

WELL, IF YOU'RE NOT
SATISFIED WITH THAT, THEN
YOU'LL JUST HAVE TO BEAT
ME ONE ON ONE.

IF YOU CAN...
IDIOT!

THERE YOU GO AGAIN!

YOU JUST DON'T KNOW
WHEN TO GIVE UP,
DO YOU, AYUMI?!!

THE END



I GOT A PROTECTIVE CASE FOR THE SHOOTING STAR!

NOW IT WON'T BREAK IF I FLIP OUT AND HIT IT WHEN I LOSE!

I CAN'T DECIDE IF YOU'RE TAKING GOOD OR BAD CARE OF IT!!

AFTERWORD

Up to this point, we have only talked about very old and primitive CPUs. The ones we have shown could only really be used for things like toys or simple AC remote controls. They are too simple to be used in most modern microcomputers or CPUs today. Currently, the speed of progress is so fast that everything I say here will quickly become obsolete. But even so, I wrote this book in an attempt to help anyone who might want to learn some lasting basics—even in these fast-changing times.

In other words, I would like to emphasize that this book has concentrated on the very basic principles governing CPUs, forgoing any information relating to general computer architecture. But even so, I would like to leave you with a small impression of the current state of progress.

It is unfortunately quite hard to illustrate modern complex circuits in the type of diagrams we used at the start of the book to show the different parts of the CPU, so I'm going to have to talk in very superficial terms here.

There are many techniques used to make modern CPUs execute programs more quickly. An older one of these techniques is the *prefetch instruction*. Instead of trying to get the next instruction after the current one has finished processing, prefetching tries to extract the next instruction from memory before the current one has completed to shorten any wait times that might otherwise occur.

Since the CPU is a lot faster than main memory, it makes a lot of sense to let prefetch decode the next instructions and store them in a cache in preparation for the next calculation. Repeating this prefetching process of reading and decoding instructions ahead of time can lead to continuous execution speed increases across the board.

There is another instruction called *pipelining* in which the instruction cycle is broken into several dependent steps. These steps are usually fetch, decode, execute, access memory, and finally write back to register/memory. The goal here is to keep all parts of the CPU busy at all times by executing these steps in parallel. So while one instruction might be running its execute step, another instruction would be running its decode step, and yet another instruction would be fetching.

CPU researchers found some tendencies toward inefficiencies when using certain instruction and operand combinations. Attempts to remove these inefficiencies by simplifying the instruction set led to the development of the *RISC (reduced instruction set computer) architecture*. Processors that use this instruction set are called RISC processors.

Many worried that reducing the instruction set would make complex calculations require more instructions and therefore slow down execution. In reality, however, many applications saw a performance boost on the RISC architecture. It turns out that reducing the instruction set's complexity leads to simplifications in the hardware design that allow for higher single instruction execution clock speeds.

Because of this, RISC processors have started to be used in many different areas. Processors that don't use RISC principles have been dubbed *CISC* (*complex instruction set computer*), riffing off the RISC name. This acronym was created purely as an antonym of RISCs, and there is no particular architecture associated with the CISC name.

Recent Intel and other CPU chips contain not just one but many cores, which are distributed between different processes on the system. This is something that falls in the domain of computer architecture, so as I mentioned at the start of the book, this is not something I will explain in much detail.

However, there is no requirement that all complex calculations must be performed in order. It is fine for the CPU to split up different parts of a task and run the individual parts on separate cores simultaneously, exchanging data between cores only when absolutely necessary. Letting the CPU multitask like this can improve execution speed a great deal. Using the CPU in such a way, however, poses problems not only for the hardware but also for the OS, memory access, and code execution scheduling.

INDEX

A

- absolute (direct) addressing, 169, 172
- accumulators, 104–105, 110
 - bit shifts, 143
 - defined, 186
- addition circuits, 62
 - carry look-ahead adder circuits, 68–69
 - full adder circuits, 66–67
 - half adder circuits, 63–65
 - ripple carry adder circuits, 67–68
- ADD mnemonic, 163, 166, 192
- address assignment, 89–91
- address bus, 92, 96–97, 99
- addressing modes, 165, 168
 - absolute addressing, 169, 172
 - effective addressing, 169
 - indirect addressing, 170–171, 174
 - relative addressing, 173
- address modification, 174–175
- address pointers, 91, 92
- address references, 167
- address registers, 108
- address space (memory space)
 - control of, 90, 119–120
 - external devices, 121
 - size of, 96–97
- ALUs (arithmetic logic units), 22–24
 - 74S181 example, 177
 - and binary arithmetic, 47
 - and bus width, 95
- AND gate (logic intersection gate), 51–55
- applications, 198
- arithmetic operations, 15,
 - in binary, 44–47
 - as instructions, 142–144, 179–180

- arithmetic shifts, 149–151
- arithmetic unit, 16–19, 22–24
- assembly languages, 193, 198
 - characteristics of, 194, 196–197
 - smaller-scale software development, 198
- asynchronous counters, 82
- ATMs, 25–26, 113–114

B

- base 2 system. *See* binary number (base 2) system
- base 10 (decimal number) system, 38–41
- base registers, 175, 186
- Basic Input/Output System (BIOS), 120, 208
- billion floating-point operations per second (GFLOPS), 138
- binary number (base 2) system
 - addition and subtraction in, 44–47
 - vs. decimal, 38–41
 - sign bits, 147–148
- BIOS (Basic Input/Output System), 120, 208
- bits, 39, 97
- bit shifts, 143
 - arithmetic shifts, 149–151
 - circular shifts, 152
 - left shifting, 146
 - logical shifts, 145–146, 149
 - overflow, 150–151
 - right shifting, 145–146
- block, 133
- borrow flag, 187
- branches, 113
- branch instructions, 155–157
 - conditional jumps, 161
 - in programs, 200–203
- bundled signal pathways, 94–95

buses

- address bus, 92, 96–97, 99
- bundled signal pathways, 94–95
- bus width, 95–97
- control bus, 99
- data bus, 92, 95, 99
- external bus, 92–93, 96
- internal data bus, 92–93

bytes, 97

C

- CAD (computer-aided design) programs, 85
- carry flag (C-flag), 160, 187
- carry input and output, 67
- carry look-ahead adder circuits, 68–69
- central processing units. *See* CPUs (central processing units)
- C-flag (carry flag), 160, 187
- circular shifts (rotating shifts), 152
- CISC (complex instruction set computer) architecture, 239
- clock frequency, 133–134
- clock generators, 134–135, 208
- clocks, 78–80, 133
 - degree of accuracy, 134
 - frequency of, 133–134
- clock speed, 133–134
- command input, 24
- compiling, 197, 198
- complements
 - in binary arithmetic, 44–47, 147–148
 - in logic operations, 60–61
 - and relative addressing, 173
- complex instruction set computer (CISC) architecture, 239
- compression, 32–33

- computer-aided design (CAD)
 - programs, 85
 - computers
 - components of, 16
 - information processing, 11-13
 - operations of, 14-15
 - condensers, 134
 - conditional branches, 161
 - conditional jumps, 161, 202
 - conditional skips, 161
 - condition evaluation, 113
 - branch instructions and, 161, 200-203
 - status flags, 158-160, 187-188
 - continuous output, 31
 - control (instruction) flow, 16, 21
 - control bus, 99
 - control signals
 - I/O signals, 100
 - R/W signals, 98-99
 - control unit, 16, 19-21
 - counters
 - asynchronous counters, 82
 - program counter, 107-108, 112-114, 187
 - synchronous counters, 82
 - C++ programming language, 198
 - C programming language, 197, 198
 - CPUs (central processing units)
 - accumulators, 104-105
 - addresses, 89-91
 - address space, 90, 96-97, 119-121
 - ALUs, 22-24
 - architecture, 106-107
 - arithmetic unit, 16-19
 - buses, 92-97
 - clock, 133-135
 - compression, 33
 - control signals, 98-100
 - control unit, 16, 20-21
 - current technologies, 238-239
 - decision making, 25-27
 - information processing, 11-13
 - instruction processing cycle, 107-114
 - interaction with other components, 16
 - interrupts, 122-129, 135-137
 - I/O ports, 132-133
 - memory classifications, 132
 - memory devices, 115-118
 - memory system. *See* memory system
 - vs. microcontrollers, 216-217
 - operands and opcodes, 102-103
 - operation processing, 14-15, 18-19, 25-27
 - performance of, 138
 - pre-execution process, 208-209
 - program counter, 107-108, 112-114
 - registers, 103-105
 - reset signals, 136-137
 - stack and stack pointer, 126-127
 - crystal oscillators, 134
- D**
- data bus, 92, 95, 99
 - data flow, 16, 21
 - data transfer operations, 153
 - decimal number (base 10)
 - system, 38-41
 - De Morgan's laws, 60-61
 - destination operand, 164
 - D flip-flops, 78-80
 - digital information and
 - operations, 12-13, 204. *See also* addition circuits; logic operations; memory circuits
 - addition and subtraction in binary, 44-47
 - vs. analog, 31-33
 - binary vs. decimal system, 38-41
 - fixed- and floating-point numbers, 42-43
 - mobile phones, 224
 - reciprocal states, 37-38
 - direct (absolute) addressing, 169, 172
 - discrete output, 31
 - DSPs (digital signal processors), 222-224
- E**
- effective addressing, 169
 - electron vacuum tubes, 220
 - embedded controllers, 215. *See also* microcontrollers
 - ENIAC computer, 220
 - exclusive logic union gate (XOR gate), 57, 59
 - exponents, 42
 - external bus, 92-93, 96
 - external devices
 - address space, 121
 - external bus, 93
 - I/O ports and signals, 100, 121, 132-133, 154
 - microcontrollers and, 219
 - synchronization, 124
- F**
- facial-recognition software, 204
 - falling edge, 79
 - fast Fourier transforms (FFTs), 224
 - fetching, 111
 - field-programmable gate arrays (FPGAs), 85, 225
 - fixed-point numbers, 42-43
 - flip-flop circuits, 74-75
 - D flip-flops, 78-80
 - RS flip-flops, 76-77
 - T flip-flops, 81-83
 - floating-point numbers, 42-43, 137-138, 151, 224
 - FLOPS (floating-point operations per second), 137-138
 - FPGAs (field-programmable gate arrays), 85, 225
 - FPU (floating point units), 15
 - frequency dividers, 135
 - full adder circuits, 66-67

- G**
- GFLOPS (billion floating-point operations per second), 138
 - GPUs (graphics processing units), 133
 - ground, 37
 - GT flag, 187
- H**
- half adder circuits, 63–65
 - hard disk drives (HDDs), 115–118
 - hardware description language (HDL), 85
 - high-level languages, 193
 - characteristics of, 194–197
 - large-scale software development, 198
- I**
- ICs (integrated circuits), 48–50. *See also* microcontrollers
 - addition circuits, 62–69
 - architecture, 178
 - De Morgan’s laws, 60–61
 - DSPs, 222–224
 - function table, 179
 - logic gates, 50–59
 - memory circuits, 70–83
 - modern circuit design, 85
 - pins, 49–50, 177
 - immediate value processing, 166
 - index registers, 175, 186
 - indirect addressing, 170–171, 174
 - information, 30–31. *See also* digital information and operations
 - analog, 31–33
 - compression of, 32–33
 - processing of, 11–13
 - signal-to-noise ratio, 30
 - transmission of, 31, 185
 - information technology (IT), 30
 - input devices, 16–17
 - input/output (I/O) instructions, 154
 - input/output (I/O) ports, 100, 121, 132–133
 - input/output (I/O) signals, 100
 - input/output (I/O) space, 117, 121
 - instruction (control) flow, 16, 21
 - instruction decoders, 109, 186
 - instruction registers, 105, 109, 186
 - instructions. *See* operations and instructions
 - integrated circuits. *See* ICs (integrated circuits); microcontrollers
 - internal data bus, 92–93
 - interrupt flag, 188
 - interrupt masks, 128, 187
 - interrupts, 122–125
 - non-maskable, 129
 - priority of, 128–129
 - resets, 209
 - stack and stack pointer, 126–127
 - timer, 129, 135–136
 - interrupt signals, 136
 - interrupt vector table (IVT), 209
 - I/O (input/output) instructions, 154
 - I/O (input/output) ports, 100, 121, 132–133
 - I/O (input/output) signals, 100
 - I/O (input/output) space, 117, 121
 - IT (information technology), 30
 - IVT (interrupt vector table), 209
- J**
- Java, 198
 - jump instructions, 155–157, 161
- L**
- large-scale software development, 198
 - latching, 74, 77
 - LDA mnemonic, 167, 192
 - left shifting, 146
 - load/store (L/S) signals, 98–99
 - logical shifts, 145–146, 149
 - logic gates, 50–51
 - addition circuits, 62–69
 - AND, 51–55
 - De Morgan’s laws, 60–61
 - NAND, 57–58
 - NOR, 57–59
 - NOT, 51, 53, 56
 - OR, 51–52, 55
 - XOR, 57, 59
 - logic intersection complement gate (NAND gate), 57–58
 - logic intersection gate (AND gate), 51–55
 - logic negation gate (NOT gate), 51, 53, 56
 - logic operations, 15, 33, 179, 181. *See also* logic gates
 - De Morgan’s laws, 60–61
 - instructions for, 143
 - integrated circuits, 48–50
 - logic union complement gate (NOR gate), 57–59
 - logic union gate (OR gate), 51–52, 55
 - loops, 113
 - lossless compression, 33
 - lossy compression, 33
 - L/S (load/store) signals, 98–99
 - LT flag, 187
- M**
- machine code monitors, 208
 - machine language, 142, 194
 - memory circuits
 - flip-flop circuits, 74–83
 - importance of, 71–73
 - registers, 70–71, 103–105
 - memory management units (MMUs), 114
 - memory space. *See* address space
 - memory system
 - addresses, 89–91
 - classifications of memory, 132
 - hard disk drives, 115–118
 - I/O space, 121
 - primary memory, 16, 18–19, 70, 115, 116–118
 - RAM space, 119–121
 - ROM space, 119–121
 - secondary memory, 16, 18, 115
 - solid state drives, 118

MFLOPS (million floating-point operations per second), 138
microcontrollers, 213
 architecture of, 220
 vs. CPUs, 216-217
 DSPs, 222-224
 function of, 214-215
 history of, 220-221
 in industrial machines, 224-225
million floating-point operations per second (MFLOPS), 138
MIPS (million instructions per second), 137
MMUs (memory management units), 114
mnemonics, 163, 192, 196-198
mode pin, 177, 179
modification registers, 175, 186
motherboards, 120
multiplexers (MUX), 93
multiplier-accumulate operation circuits, 222, 224

N

NAND gate (logic intersection complement gate), 57-58
negative flag (N-flag), 187
noise (information), 30, 33
non-maskable interrupts (NMI), 129
non-volatile memory, 132, 208
NOR gate (logic union complement gate), 57-59
NOT gate (logic negation gate), 51, 53, 56
number systems, 38-41

O

object code, 199
ODD flag, 187
on-board programming, 208
opcodes, 102-103, 110, 142, 162-163, 180
operands, 102-103, 110, 142
 addressing modes, 165, 168-174
 address modification, 174-175

 address references, 167
 immediate value processing, 166
 number of, 163-164
 types of, 162-165
operation execution speed, 137
operations and instructions, 14. *See also* arithmetic operations; bit shifts; digital information and operations; logic operations
ALUs and, 22-24
branch instructions, 155-157, 161, 200-203
data transfer operations, 153
I/O instructions, 154
jump instructions, 155-157, 161
memory and, 18-19, 70-71, 89-90, 103-105
processing and decision making, 25-27
programs and, 19
skip instructions, 157
SLEEP instruction, 188
types of, 15
OR gate (logic union gate), 51-52, 55
output devices, 16-17
overflow, 45, 150-151
overflow flag (overflow bit; OV-flag), 151, 187

P

parallel transmission, 185
PC (program counter), 107-108, 112-114, 187
personal computers (PCs), 220
pins, 49-50
pipelining, 238
prefetch instructions, 238
primary memory, 16, 18-19, 70, 115, 116-118
primitives, 32
processing speed, 118
program counter (PC), 107-108, 112-114, 187

programs, 19, 101, 192, 199
 assembly languages, 192-194, 196-197
 with conditions and jumps, 200-203
 control unit and, 20-21
 high-level languages, 194-197
 large-scale software development, 198
 machine language, 194
 pre-execution process, 208-209
 vs. source code, 199
 storage of, 208
propagation delay, 68
Python, 198

R

RAM (random access memory), 119-121, 132, 208
read-only memory (ROM), 119-121, 132, 208
read/write (R/W) signals, 98-99
read-write memory (RWM), 132
reduced instruction set computer (RISC) architecture, 238-239
registers, 70-71, 83, 103-104
 accumulators, 104-105, 110, 143, 186
 address registers, 108
 base registers, 175, 186
 index registers, 175, 186
 instruction decoders, 109, 186
 instruction registers, 105, 109, 186
 program counter, 107-108, 112-114, 187
 shift registers, 185
 stack pointer, 126-127, 187
 status registers, 160, 186
 temp registers, 186
relative addressing, 173
repeating processes, 202
resets, 128-129
reset signals, 136-137

- reset vector, 208–209
- right shifting, 145–146
- ripple carry adder circuits, 67–68
- RISC (reduced instruction set computer) architecture, 238–239
- rising edge, 79
- ROM (read-only memory), 119–121, 132, 208
- rotating shifts (circular shifts), 152
- RS flip-flops, 76–77
- R/W (read/write) signals, 98–99
- RWM (read-write memory), 132

S

- SAM (sequential access memory), 132
- scientific notation (standard form), 42
- SD cards, 220
- secondary memory, 16, 18, 115
- select pins, 177, 179
- semiconductors, 220
- sequential access memory (SAM), 132
- serial transmission, 185
- S-flag (sign flag), 160, 187
- shift registers, 185
- signal (information), 30
- signals (I/O), 56
- signal-to-noise ratio, 30
- sign bits, 147–148
- sign flag (S-flag), 160, 187
- skip instructions, 157
- SLEEP instruction, 188
- solid state drives (SSDs), 118
- source code, 198–199
- source operand, 164
- stack, 126–127
- stack pointer (SP), 126–127, 187
- STA mnemonic, 167, 192
- standard form (scientific notation), 42
- state, 71, 74
- status flags, 159–160, 187–188, 201

- status output, 24–26
- status registers, 160, 186
- synchronization, 124
- synchronous counters, 82

T

- temp registers, 186
- T flip-flops, 81–83
- TFLOPS (trillion floating-point operations per second), 138
- thermometers, 31–32
- timer interrupts, 129, 135–136
- transistors, 220
- trigger conditions, 74
- trillion floating-point operations per second (TFLOPS), 138
- truth tables, 53–56, 58–59
- two's complement
 - in binary arithmetic, 44–47
 - expressing negative numbers in binary, 147–148
 - and relative addressing, 173

U

- underflow, 151

V

- variables, 195
- Venn diagrams, 54–56, 58–59
- virtual memory, 114
- vocoders, 224
- volatile memory, 132
- voltage, 31
 - as binary states, 37–38
 - and reset process, 136–137
 - and turning on CPU, 208

X

- XOR gate (exclusive logic union gate), 57, 59
- xx-register relative addresses, 173

Z

- zero flag (Z-flag), 187

ABOUT THE AUTHOR

Michio Shibuya graduated from the electrical engineering department of Toukai University in 1971. Among other occupations, he has worked as an NMR researcher in a private medical institution, has spent 12 years working as a MOS product designer and developer for a foreign company, and has since pursued a career in IC design at technical departments of both domestic and foreign trading companies. Since May of 2007, Shibuya has worked for the semiconductor trading company Sankyosha, first as a field application engineer and currently as a special advisor. He is also the author of *Learning Signal Analysis and Number Analysis Using Excel*, *Learning Fourier Transforms Using Excel*, *The Manga Guide to Fourier Transforms*, *The Manga Guide to Semiconductors*, and *Learning Electrical Circuits Using the Circuit Simulator LTspice* (all published by Ohmsha).

PRODUCTION TEAM FOR THE JAPANESE EDITION

Production: Office sawa

Office sawa was established in 2006 and specializes in advertisement and educational practical guides in medicine and computers. They also take pride in their sales promotion materials, reference books, illustrations, and manga-themed manuals.

Email: office-sawa@sn.main.jp

Scenario: Sawako Sawada

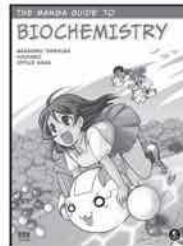
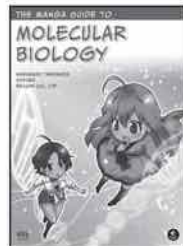
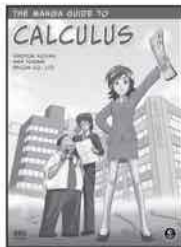
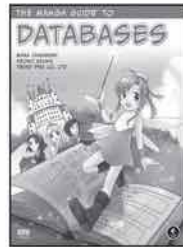
Illustrations: Takashi Tonagi

HOW THIS BOOK WAS MADE

The *Manga Guide* series is a co-publication of No Starch Press and Ohmsha, Ltd. of Tokyo, Japan, one of Japan's oldest and most respected scientific and technical book publishers. Each title in the best-selling *Manga Guide* series is the product of the combined work of a manga illustrator, scenario writer, and expert scientist or mathematician. Once each title is translated into English, we rewrite and edit the translation as necessary and have an expert review each volume. The result is the English version you hold in your hands.

MORE MANGA GUIDES

Find more *Manga Guides* at your favorite bookstore, and learn more about the series at <https://www.nostarch.com/manga/>.

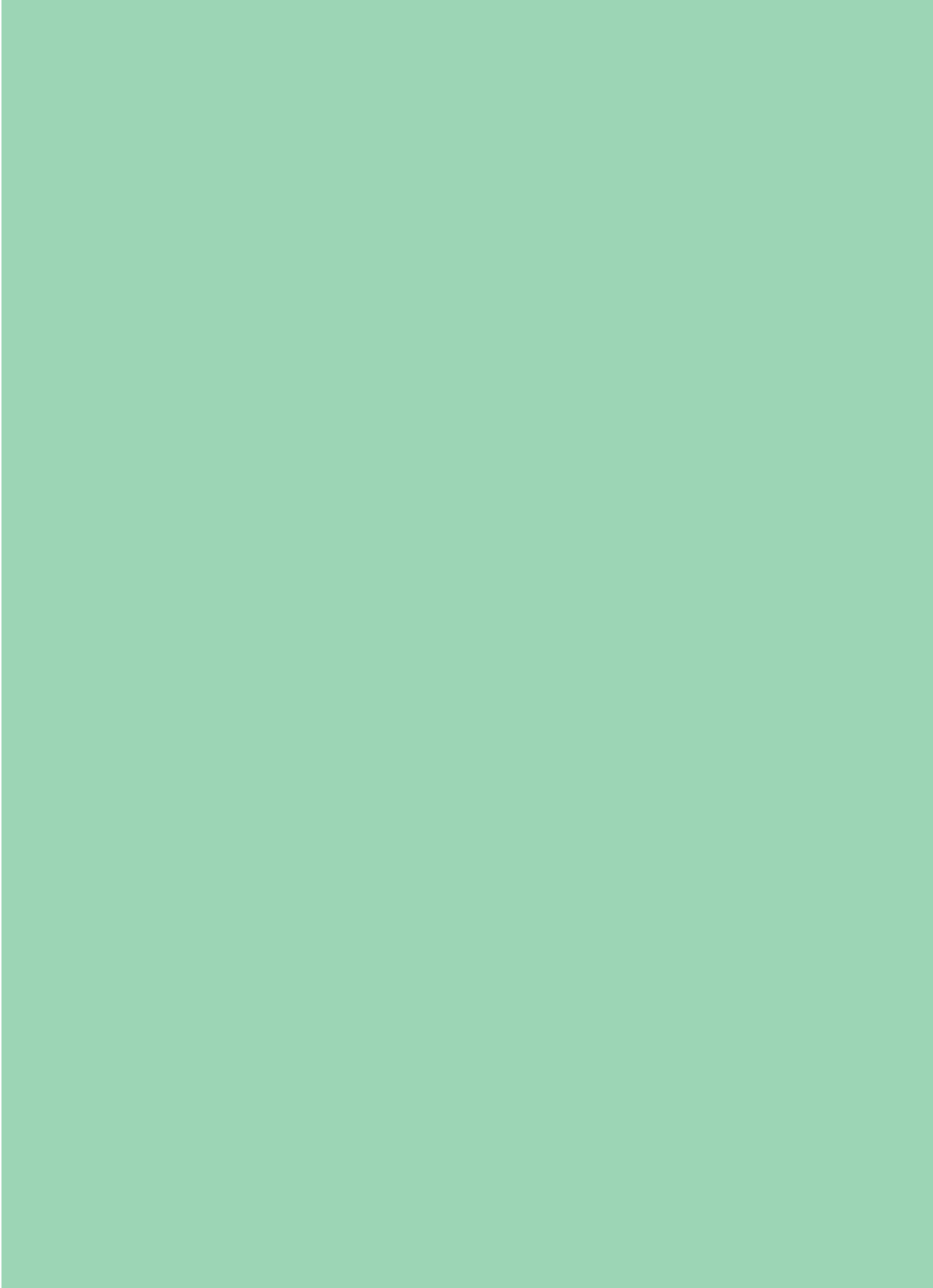


UPDATES

Visit <https://www.nostarch.com/microprocessors/> for updates, errata, and other information.

COLOPHON

The Manga Guide to Microprocessors is set in CCMeanwhile and Chevin.



A CARTOON GUIDE TO MICROPROCESSORS



AYUMI IS A WORLD-CLASS SHOGI (JAPANESE CHESS) PLAYER WHO CAN'T BE BEATEN—THAT IS, UNTIL SHE LOSES TO A POWERFUL COMPUTER CALLED THE SHOOTING STAR. AYUMI VOWS TO FIND OUT EVERYTHING SHE CAN ABOUT HER NEW NEMESIS. LUCKY FOR HER, YUU KANO, THE GENIUS PROGRAMMER BEHIND THE SHOOTING STAR, IS WILLING TO TEACH HER ALL ABOUT THE INNER WORKINGS OF THE MICROPROCESSOR—THE "BRAIN" INSIDE ALL COMPUTERS, PHONES, AND GADGETS.

FOLLOW ALONG WITH AYUMI IN *THE MANGA GUIDE TO MICROPROCESSORS* AND YOU'LL LEARN ABOUT:

- * HOW THE CPU PROCESSES INFORMATION AND MAKES DECISIONS

- * HOW COMPUTERS PERFORM ARITHMETIC OPERATIONS AND STORE INFORMATION
- * LOGIC GATES AND HOW THEY'RE USED IN INTEGRATED CIRCUITS
- * THE KEY COMPONENTS OF MODERN COMPUTERS, INCLUDING REGISTERS, GPUS, AND RAM
- * ASSEMBLY LANGUAGE AND HOW IT DIFFERS FROM HIGH-LEVEL PROGRAMMING LANGUAGES

WHETHER YOU'RE A COMPUTER SCIENCE STUDENT OR JUST WANT TO UNDERSTAND THE POWER OF MICROPROCESSORS, YOU'LL FIND WHAT YOU NEED TO KNOW IN *THE MANGA GUIDE TO MICROPROCESSORS*.

FIND MORE MANGA GUIDES AT WWW.NOSTARCH.COM/MANGA

SHELVES IN: COMPUTERS/HARDWARE



THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com