NATIONAL BESTSELLER

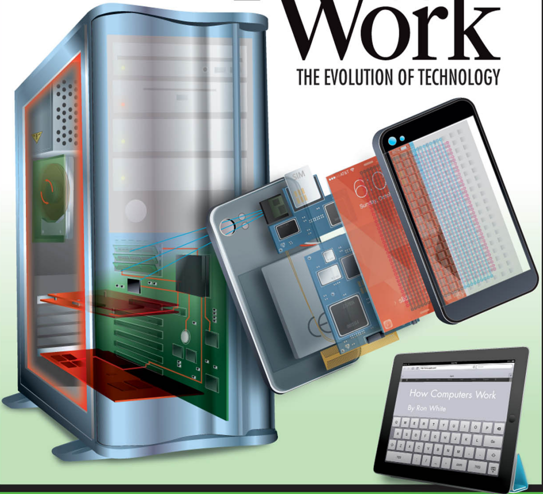
MORE THAN 2,000,000 How It Works SOLD

BY RON WHITE

ILLUSTRATED BY TIM DOWNS

How Computers Work

THE EVOLUTION OF TECHNOLOGY



TENTH EDITION

que

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

SPECIAL OFFER ONLY $4.99!

How Computers Work

Interactive Web Version

The *How Computers Work, Interactive Web Version* is an enhanced digital copy of the *How Computers Work* print book, which can be accessed via a web browser. It includes many new and exciting features that take the one-of-a-kind *How Computers Work* print book experience to the next level. Here's a look at just a few of the new features you'll find in the *Interactive Web Version:*

Designed for the Web—This new digital edition of *How Computers Work* is built from the ground up to work in today's most popular web browsers. Whether you use Windows, Mac, or a tablet, you'll find a viewing experience that works for you.

Animations—See first-hand how morphing software morphs a photo or how video compression and resolution changes the quality of a digital video. Watch as an interactive firewall blocks dangerous data, while letting safe packets through. Have you ever wondered just how a transistor physically controls the flow of electrons to turn itself on and off? You'll see it in action here!

Interactive Elements—Directly interact with more than a dozen of the topics covered in this book. Drag data into a processor to see how it divides up a work load and computes an answer. Or use a virtual laser to photoetch a micochip. You'll even control the flow of light through an LCD monitor as you learn how it generates an image!

How to Purchase the Interactive Web Version:

Register this eBook to unlock a single-use discount, which will allow you to purchase a copy of the Interactive Web Version of *How Computers Work* (ISBN: 9780133096781) for only $4.99, a savings of $25!*

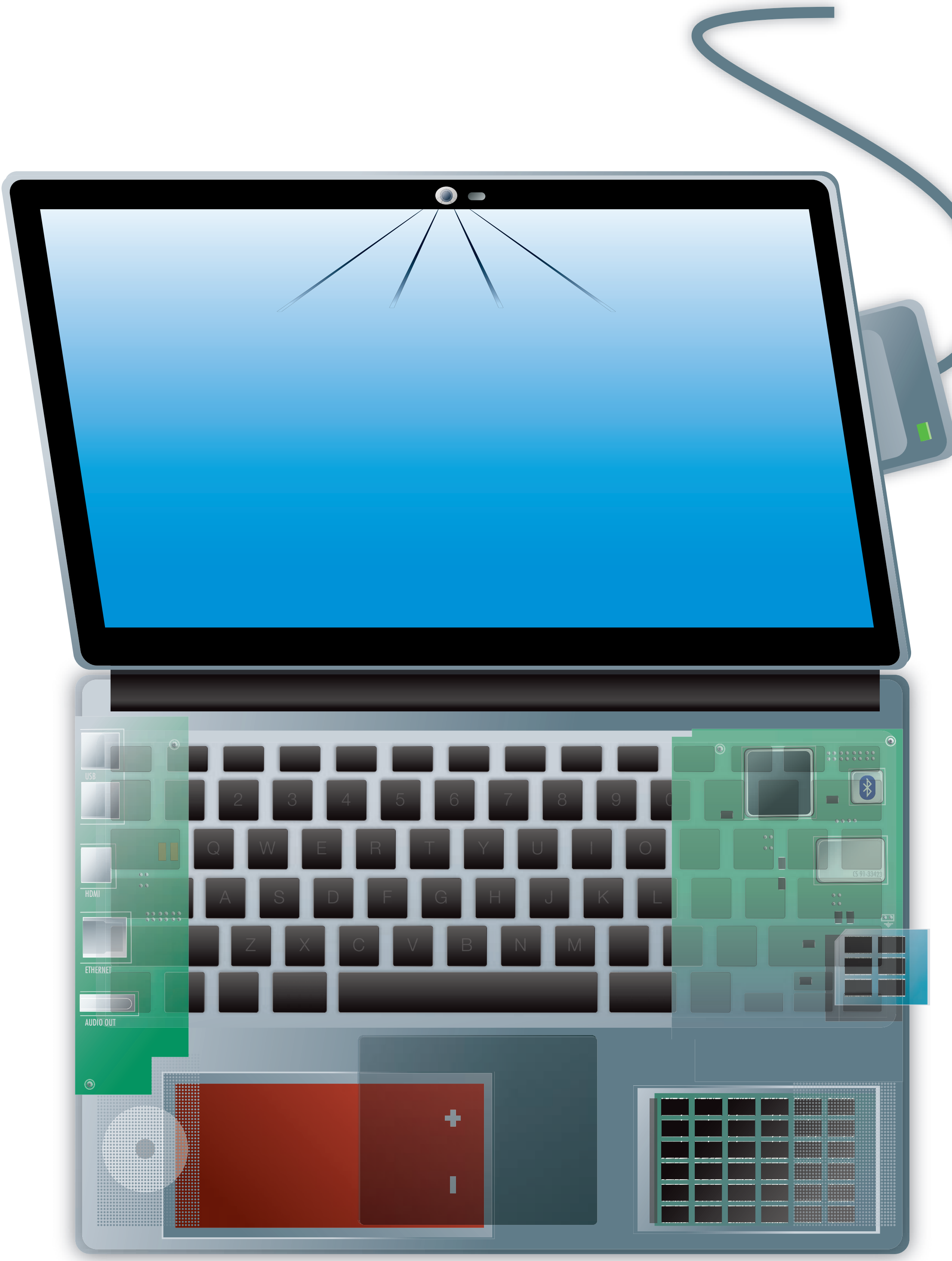
Follow the steps below:

1. Go to quepublishing.com/register and log in or create a new account.
2. Enter the ISBN: 9780789749840.
   (NOTE: please enter the print book ISBN provided to register the eBook you purchased)
3. Answer the challenge question as proof of purchase.
4. Click on the "Access Bonus Content" link in the Registered Products section of your account page, to be taken to the page where your discount code is available.
5. Go to www.quepublishing.com/title/9780133096781.
6. Add the product to your Cart.
7. Click on "Proceed To Checkout".
8. Enter your discount code in the "Enter a New Discount Code" box.
9. Complete payment information.

Discount may not be combined with any other offer and is not redeemable for cash. Offer subject to change.

How Computers Work

Tenth Edition


USB
HDMI
ETHERNET
AUDIO OUT

How Computers Work

The Evolution of Technology

Tenth Edition

Ron White

Illustrated by Timothy Edward Downs

que®

800 East 96th Street
Indianapolis, IN 46240

How Computers Work, Tenth Edition

ISBN-13: 978-0-7897-4984-0

ISBN-10: 0-7897-4984-X

Library of Congress Control Number: 2014951587

Printed in the United States of America

First Printing: December 2014

Trademarks

Warning and Disclaimer

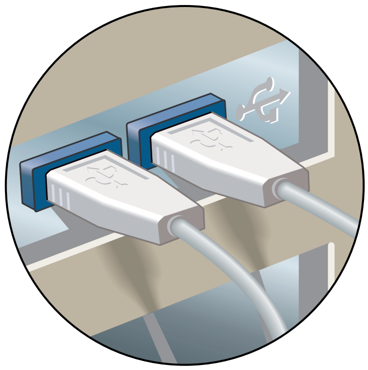
Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com

Editor-in-Chief	Greg Wiegand
Executive Editor	Rick Kughen
Development Editor	Todd Brakke
Managing Editor	Sandra Schroeder
Project Editor	Mandie Frank
Copy Editor	Charlotte Kughen
Indexer	Lisa Stumpf
Proofreader	The Wordsmithery LLC
Editorial Assistant	Kristen Watterson
Designer	Mark Shirar
Compositor	Mary Sudul
Graphics	Timothy Edward Downs



For Shannon and Michael, who always asked, “Why?”
And Sue, who asked “Why Not?”
—Ron

For my mother, Lillian, whose tireless work ethic and uncompromising dedication to quality has inspired me all of my life. I owe so much of who I am to you.
—Tim

Table of Contents

Part 4: How Computers Expand Our Senses 200

Part 5: The Little Net That Grew 248

About the Author

RON WHITE has been writing *How Computers Work* for 20 years, during which time he's also been executive editor at *PC Computing* magazine, BYTE.com, and groovyPost.com. He's been a computer columnist for Windows Sources and 80 Micro. *How Computers Work* was named best nonfiction computer book, and his writing has been honored by the Maggie Awards, the Robert F. Kennedy Journalism Awards, and The National Endowment for the Humanities. He and his wife, Sue, have bounced back and forth between Boston and San Francisco before finally ending up in San Antonio.

About the Illustrator

TIMOTHY EDWARD DOWNS is the national award-winning illustrator of *How Computers Work* and *How Digital Photography Works*. Tim has been involved in all facets of graphic design in his illustrious career. From illustrator to creative director, Tim has led teams of artists and designers in advertising agencies, marketing communications firms, and consumer magazines to better tell their stories through illustration, photography, typography, and design. "Our job doesn't start when the writer hits Save. In order to effectively communicate the tone or the concept of the piece, we need to know and understand the story from the original brainstorm all the way through final execution," reminds Tim.

Examples of Tim's design, illustration, and photographic work can be seen at http://timothyedwarddowns.com.

Acknowledgments

YOU CAN'T WRITE 10 versions of the same book over a span of 20 years without a lot of help from a lot of people. This book certainly couldn't have been done without Tim Downs. His illustrations have set a new standard for technical art. And *art* is the word. Not only is he always careful to make sure his illustrations are correct technically, he has imbued them with his talent to make them works of art that could stand alone outside of this book's technical explanations. We've worked together for 20 years, and I'm happy that unlike some other partnerships I've entered into, I continue to think of Tim as not just an illustrator, but a dear friend.

This particular edition could not have been done without the aid of some others. My wife, Sue, has served as editor, critic, coach, and personal trainer. I know that the time I've spent in front of a keyboard was not always pleasant for either of us. But when I felt like chucking it all, Sue was there, helping me in the ways I needed most. Our son, Michael White, also spent hours helping with tedious research and checking for errors, despite running his own business and returning to the classroom. Thank you, both.

It's not fashionable among many writers to heap praise on their editors and publishers. But I can't heap enough on my editor, Todd Brakke, and Que Publisher Greg Wiegand. In this and other editions, Todd has been the editor all writers want, ever ready with unerringly good suggestions, encouragement, and lending a hand with the words when time was pressing. Greg has always seemed more like a friend than someone who hired me to write still another edition. When I was ill during the course of writing this, his concern was for my health, not the deadlines I missed. Thank you, guys.

I'm indebted to other editors who gave me the foundations to write. Roddy Stinson taught me the importance of listening to my own voice. Jim Dolan showed me how every detail matters. The late Jim Seymour, who was the editor-in-chief of *PC Computing* when it was launched in 1988, and his wife Nora, who was editor, gave me the opportunity to jump from newspaper to magazines, and were exceptional, warm friends. Preston Gralla guided me in setting up a How It Works section when I joined *PCC*. This book wouldn't be here without him, and I can never thank him enough for a friendship and too many acts of kindness to list.

I owe a huge debt to Mike Edelhart, who was editor-in-chief and publisher of *PC Computing* and who convinced a then dubious book publisher to take a chance on a kind of computer book that had never been done before. Deep thanks go to Randy Ross, Margaret Ficklen, Herb Brody, Brett L. Glass, Marty Jerome, Raymond Jones, Matthew Lake, Jack Nimersheim, Stephen Sagman, Jan Smith, Dylan Tweney, Doug van Kirk, Mark L. Van Name, Bill Catchings, Christine Grech Wendin, and Kenan Woods, all of whom helped in earlier editions.

Thanks to editors now and over the course of nine editions: Rick Kughen, Stephanie McComb, Tonya Simpson, Nick Goetz, Angelina Ward, Sarah Robbins, Cindy Hudson, Melinda Levine, Lysa Lewallen, Renee Wilmeth, and Leah Kirkpatrick for pulling everything together. Juliet Langley pushed the first edition to book sellers with such fervor I doubt it could ever have been a success without her.

I'm also grateful to the dozens of people in the PC industry who've shared their knowledge, schematics, and white papers to infuse *How Computers Work* with detail and accuracy. I'm particularly grateful to Ray Soneira, author of Sonera Technologies' DisplayMate and the person who knows more about display technology than anyone else in the world; Griffin Lemaster of Creative Electron; Chris Thornton of ClipMate; Karen Thomas at Olympus; Joe Vanderwater, Bryan Langerdoff, Dan Francisco, Tami Casey, John Hyde, Bill Kircos, Susan Shaw, and Seth Walker at Intel; Russell Sanchez and Adam Kahn of Microsoft; Jim Bartlett, Ellen Reid Smith, Tim Kearns, and Desire Russell at IBM; Barrett Anderson and Todd Hollinshead at id Software; Marcie Pedrazzi of Adaptec; Eileen Algaze at Rockwell; A.J. Rodgers, Jennifer Jones, and Susan Bierma at Tektronix; Ben Yoder and Lisa Santa Anna at Epson; Dewey Hou at TechSmithCorporation; Kathryn Brandberry and Doyle Nicholas at Thrustmaster; Dani Johnston at Microtek Lab; Lisa Tillman of The Benjamin Group; Dvorah Samansky at Iomega; Brandon Talaich at SanDisk; Chris Walker with Pioneer USA; Carrie Royce at the Cirque Corporation; Andy Marken at Marken Communications; and Tracy Laidlaw at BeyondWords. If there are any mistakes in this book, I can't lay the blame on them.

As this *may* be the last edition of the book I work on, I want to thank all the readers for their kind comments, spotting errors, and, of course, shelling out money to buy it. This odyssey would not be possible without you.

We Want to Hear From You!

AS the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@quepublishing.com

Mail: Que Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at quepublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

About This Book

"Any sufficiently advanced technology is indistinguishable from magic."

—Arthur C. Clarke

SORCERERS have their magic wands—powerful, potentially dangerous tools with lives of their own. Witches have their familiars—creatures disguised as household beasts that could, if they choose, wreak the witches' havoc. Mystics have their golems—beings built of wood and tin brought to life to do their masters' bidding.

We have our personal computers.

PCs, too, are powerful creations that often seem to have a life of their own. Usually, they respond to a wave of a mouse or a spoken incantation by performing tasks we couldn't imagine doing ourselves without some sort of preternatural help. But even as computers successfully carry out our commands, it's often difficult to quell the feeling that there's some wizardry at work here.

And then there are the times when our PCs, like malevolent spirits, rebel and open the gates of chaos onto our neatly ordered columns of numbers, our carefully wrought sentences, and our beautifully crafted graphics. When that happens, we're often convinced that we are, indeed, playing with power not entirely under our control. We become sorcerers' apprentices, whose every attempt to right things leads to deeper trouble.

Whether our personal computers are faithful servants or imps, most of us soon realize there's much more going on inside those silent boxes than we really understand. PCs are secretive. Open their tightly sealed cases and you're confronted with poker-faced components. Few give any clues as to what they're about. Most of them consist of sphinx-like microchips that offer no more information about themselves than some obscure code printed on their impenetrable surfaces. The maze of circuit tracings etched on the boards is fascinating, but meaningless, hieroglyphics. Some crucial parts, such as the hard drive and power supply, are sealed with printed omens about the dangers of peeking inside—omens that put to shame the warnings on a pharaoh's tomb.

This book is based on two ideas. One is that the magic we understand is safer and more powerful than the magic we don't. This is not a hands-on how-to book. Don't look for any instructions for taking a screwdriver to this part or the other. But perhaps your knowing more about what's going on inside all those stoic components makes them a little less formidable when something does go awry. The second idea behind this book is that knowledge, in itself, is a worthwhile and enjoyable goal. This book is written to respond to your random musings about the goings-on inside that box you sit in front of several hours a day. If this book puts your questions to rest—or raises new ones—it will have done its job.

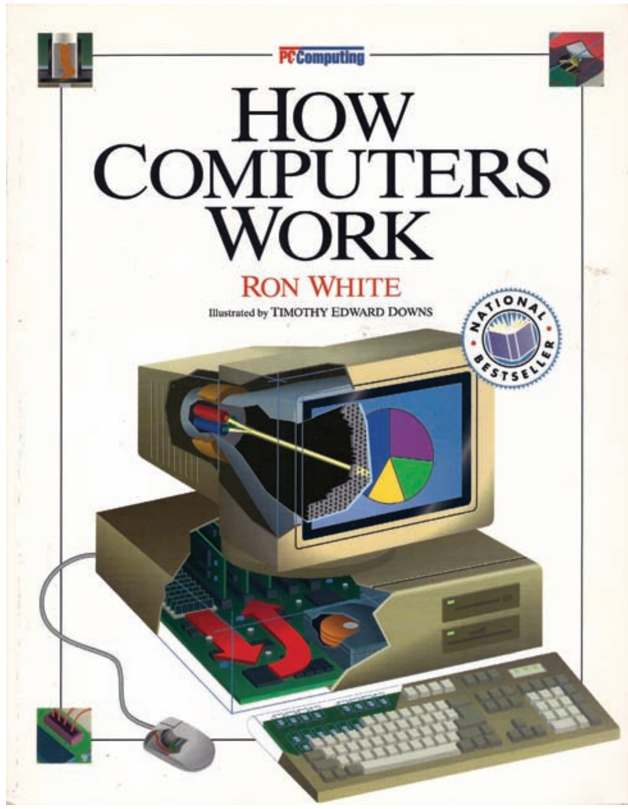
At the same time, however, I'm trusting that knowing the secrets behind the magician's legerdemain won't spoil the show. This is a real danger. Mystery is often as compelling as knowledge. I'd hate to think that anything you read in this book takes away that sense of wonder you have when you manage to make your PC do some grand, new trick. I hope that, instead, this book makes you a more confident sorcerer.

Introduction to the Tenth Edition

"If automobiles had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside."

—Robert Cringely

THIS BOOK was so much easier to write 20 years ago. Computers were relatively new, and they were all pretty much the same. There were only two important differences—whether a computer was an Apple or a Wintel, that is, running Windows on an Intel processor. There were some variations in size, portability, and the customization enthusiasts could do by installing different drives and expansion boards inside PCs. Stuffing the newest-fangled components into my computer at *PC Computing*, the magazine where I worked, was a wonderful pastime and the source of many stories in the magazine. If I made a mistake and the computer groaned its last breath, that often made for an even better story.



One reason PC disasters made for good reading is that every computer user had a similar story of computer disaster. We were all in the same boat because even the editors of computer magazines—if the truth be told—didn't know what they were doing a lot of the time. Back then if you took a class on computers in college, you waited your turn to run a program on some monster of a computer that would return a solution in the form of punch cards. There was no formal place to go to learn about personal computers. Manuals were useless beyond finding out how to plug a PC into the wall and where the switch was located. Programs that worked one way had no resemblance to how other programs worked. You learned about computers by trial and error, by going to user groups made up of equally befuddled PC owners, and by reading magazines.

Before I went to work for *PC Computing* in 1989, my teachers were *PC Magazine*, *PC World*, *Byte*, *Popular Computing*, and other magazines covering this new technical phenomenon. You found a morsel of information in one magazine, a nugget in another, and eventually you pieced together what was going on inside that steel box on your desktop.

But I was going to tell why it was easier to write this book 20 years ago. Back then, you could say all someone needed to know about a floppy disk in a couple of two-page spreads. Something as mind-busting and intricate as a monitor you could do with one illustration. The technology was, in retrospect, elementary. How computers and their components worked was actually simple when you sketched it all out and added some callouts and arrows. No wonder I figured I could, with the help of a great illustrator like Timothy Edward Downs, write a book that laid out for all to see the mysteries of what really is a fascinating technology.

But today? Lord love a duck! Today there's not even a simple explanation of what a computer is. Simply having a microchip doesn't cut it. Microchips are everywhere, in watches, cars, thermostats, ovens, refrigerators, flashlights, dog collars, coffee pots—just about anything that uses electricity. But these aren't computers. They're just electronic doo-dads that use the abilities of microprocessors to perform routine chores.

To be a computer, something must be programmable. It must be capable of doing different things based on the instructions you give it. Your yard's sprinkler system is a computer. True, it doesn't do a lot of different things, but you can program it to water different parts of your yard for different lengths of time, and on different days, or not at all if it's raining. Microwave ovens are computers you use by programming it on the fly by setting how long it will cook at different temperatures, or you can push a Popcorn button to tell the oven to use a preprogrammed sequence.

These are all computers, but the truth is they're not very interesting. How deeply do you care how your sprinkler works? But a computer that can be programed to do so… many… different… things—now, that is a computer worthy of the name, and we see them everywhere today. Phones, cameras, tablets, music players, and TVs have all become computers that are indispensable parts of everyday life. Our lives are richer for the knowledge, problem solving, communication, entertainment, health, and income they provide.

They certainly deserve to be included in a book that calls itself *How Computers Work*. The problem—my problem anyway—is that they make organizing a book like this an exercise in controlled chaos. In previous editions, I simply had a section on storage, another section on displays, one on networking…. I admit that in this edition I've fallen back on that system in some chapters out of pure convenience. But generally the workings of today's computers are not so compartmentalized. The more I looked at computing over the last 20 years, the more it struck me that the real story of how computers work is how they are like living creatures, organisms that have evolved and continue to evolve. These metal and silicon and plastic and glass animals have followed the same patterns of mutation, natural selection and survival of the fittest that Darwin found in plants and animals.

I've tried to explain how technological developments follow the same rise and fall that carry organic species to new shores. Today's achievements in computing are possible only because they rise out of the fossil beds of earlier technology. There is no way for today's technology to have been invented from scratch. I recall that some 30 years ago, I was working for a computing service company that relied on mainframes, an obsolete term to describe computers the size of a couple of high-end refrigerators slammed together. They used two kinds of storage: hard drives and magnetic tape. The 5 megabyte hard drives were the size of large garbage cans, and the platters where information was stored could have doubled as hub caps. You had to have seen the magnetic tape at work to understand one of computer's most common terms: RAM—random access memory. It sounds as if data is retrieved willy-nilly from all different parts of memory chips with no order system to it. Memory chips got their RAM moniker because retrieving tape memory was not random. If you wanted data stored on a reel at the innermost 10 inches of tape, the computer had to wind through the first 3,590 feet to get to that data.

Sure, it would have been better to skip tape altogether and go straight to hard disks. But to paraphrase a certain infamous quote, sometimes you don't build computers with the technology you wish you had; you build them with the technology you have.

Life in the primordial seas had to get the hang of existence as one-celled organisms before the cells could master hooking together and dividing up the jobs needed for mutual survival. Some of those bigger organisms learned to move by wiggling. They developed a way to filter oxygen from water, and later that system developed into lungs that could suck oxygen out of water. It took millions of years for these changes to evolve. You couldn't create magnetic disks until you'd learned how to record data to magnetic tape. Then it took years of engineers trying first this and then that to make the disks smaller and at the same time more capacious. Current solid state drives come from the fossils of technological forays, such as bubble memory, that didn't survive.

The most amazing aspect of this whirlwind of technical progress is that it's only taken 20 years to go from floppy to solid state, from displays that amounted to stick figures to animated graphics that are almost indistinguishable from film images. That smartphone in your pocket has more computing power than those refrigerator-sized mainframes. The wires that have been needed for communications for a century are likely to disappear completely in the next 10 years, along with the need for local data storage. It's possible the keyboard and mouse will become extinct. In fact much of the hardware we're used to today, as its gets smaller and more efficient, may migrate from our desks and our pockets to become parts of our eyes, ears, and brain.

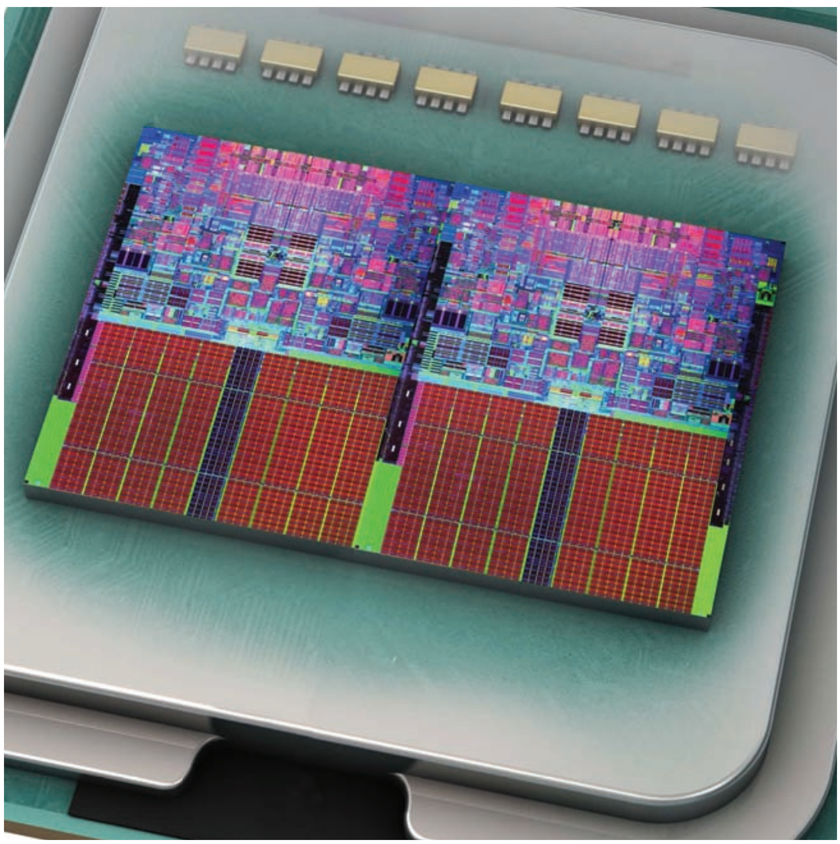
And that's just in the next 20 years. Beyond that we can't imagine what will become of technology any more than one-celled organisms could have imaged a dinosaur. Let's check back with each other after a couple of decades and then see what may be possible.

Ron White
San Antonio, TX

CHAPTER

3

How a Little Microprocessor Does Big Things



IF YOU WANT to feel what it's like to be a microprocessor, go to a toy store for tots and get one of those baby toys that has a bunch of different-sized rings the kid is supposed to put on a stick to make a kind of round pyramid thing. If you don't feel like blowing money on a kid's toy, pull some change out of your pocket. A dime, penny, nickel, quarter, half-dollar (if you can find one), and a dollar bill have a couple of things in common with the toy rings: None is the same size as any other, and they can be stacked. That's all you need for this passage into the brain-numbing world of the microprocessor.

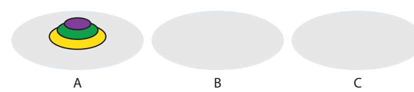
This is called the Towers of Hanoi Puzzle. It's simple but tedious. All you have to do is move the complete stack of objects—rings, coins, plates, whatever—from their initial position, which I'll call "A," to one of two other spots, "B" and "C." There are two rules:

1) You can move only one object at a time.

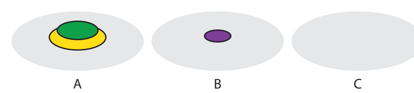
2) You cannot place one object on an object smaller than the object you're moving. For example, you can't put a quarter on top of a dime.

In the original, legendary version of the game, you must move 64 rings while following those rules. It doesn't sound that complicated. And it's not, really. Mathematicians have pointed out the similarities between the Towers of Hanoi and binary computations, geometry, and the exact duplication of a function called the binary carry sequence.

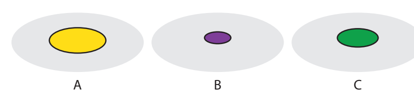
But never mind all that. Let's play the game using only three rings, or disks, as we've drawn them here:



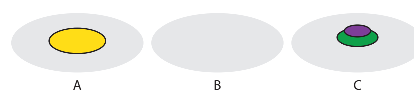
1. Take the top, purple disk and place in the empty B spot.



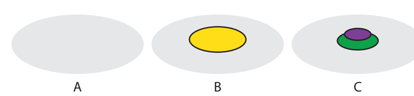
2. Place the green disk on C.



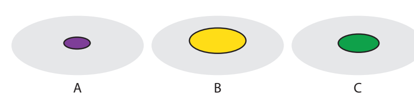
3. Put the purple disk from position B and put it on top of the green disk.



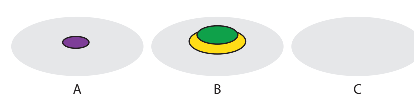
4. Move the yellow disk from A to the B position.



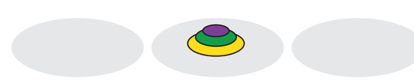
5. Put the purple disk on the now empty position A.



6. Move the green disk on top of the yellow disk at B.



7. Move the purple disk to sit on the green disk.



There. In seven moves you've relocated three disks to a different spot. As you might have guessed, it gets more intricate as you move down through the stack—not more difficult, just more, and more, of the same. Add one more disk, and it takes 15 moves. Ten disks add 1,023 moves.

The Towers of Hanoi Puzzle was discovered by French mathematician Edouard Lucas in 1883. A legend, possibly created by Lucas, has it that there is a temple with a room that contains three poles and 64 gold disks with holes in the centers. The priests at the temple are on a mission from God to move all the discs from one pole to another, Towers-of-Hanoi style. When they complete the task, the world will come to an end. Not to worry. The number of moves to complete the puzzle increases exponentially as the number of discs increases. If the priests could move one disc each second, it would take them 2^{64}[ms]–1 seconds, or roughly 585 billion years, which is 127 times the age of the sun, to complete all the 18,446,744,073,709,551,615 moves required to finish the mission.

This sort of mind-numbing task was made for computing. The **CPU**, or **central processing unit**, whether in a computer, a watch, a smartphone, hearing aid, smart TV, or even a toaster oven plays its own version of the Towers of Hanoi. Like the moves in the puzzle, the methods used by processors work by doing the same thing many times. Multiplying 100 by 3,000 is accomplishing by adding 3,000 a hundred times. Division consists of subtracting one number from another continuously until the result is 0 or the final number is not large enough to subtract the divisor from it.

In the most powerful processors for PCs, the CPU can work with 64 bits at a time, but this doesn't mean that in the monks' full 65 ring game, a 64-bit computer could grab the first 64 disks and move them all at once. It still has to move only one tower ring with each tick of its internal clock, but the computer could simultaneously move one ring on 63 other towers. (A processor such as the Intel Core i7-965 Extreme Edition, with a clock speed of 3.20 gigahertz—3,200,000,000 ticks of the clock every second—could complete the monks' original tower exercise in 13 day and four hours.)

A processor is limited in the number of places it can put the binary numbers representing the bits while still carrying out its software's instructions. After adding two numbers, the CPU might need to put the result in a part of the chip that acts as a temporary holding pen. After adding two more numbers, it retrieves the stored number to add it to the sum of the second addition. It's kind of like how we humans scribble the result of some math operation on a slip of paper, so we can use it a minute later when we solve a related problem.

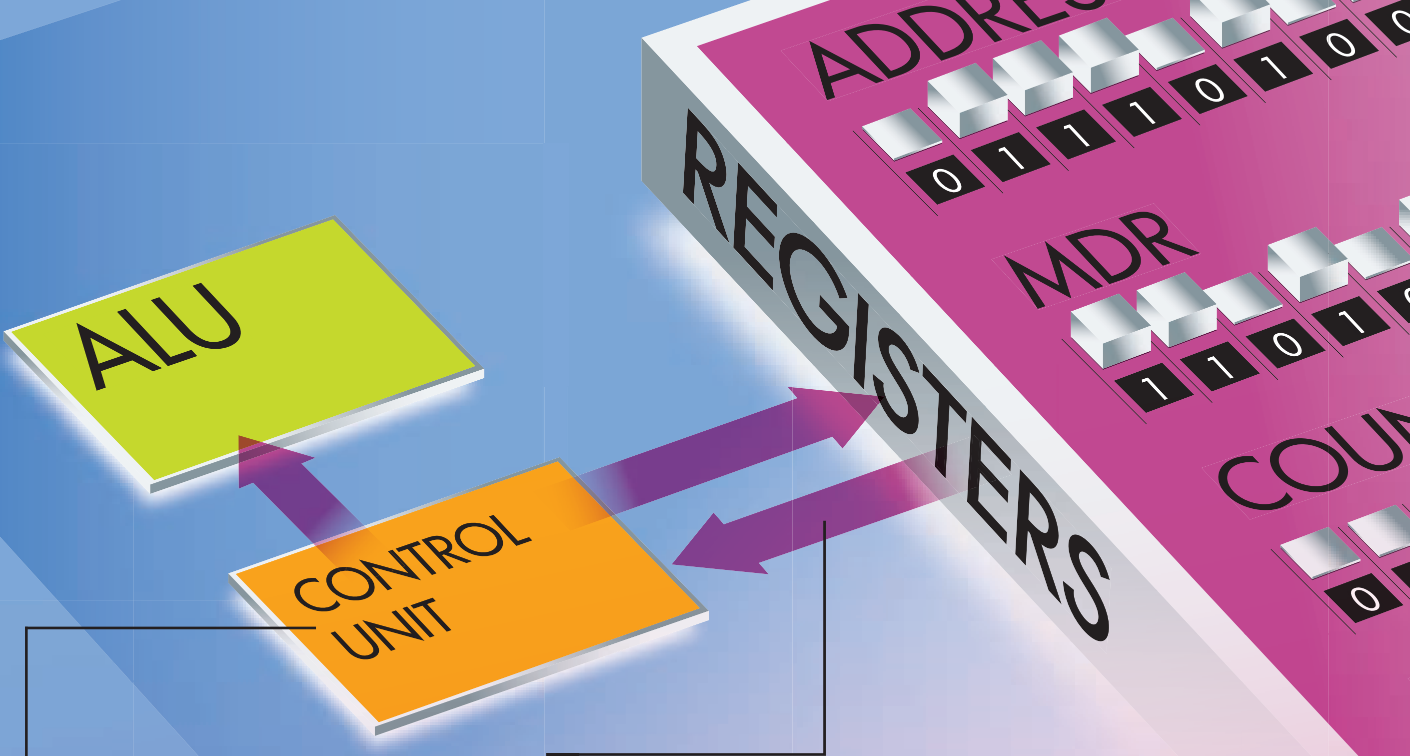
Much is made of all the amazing things computing can do, but what it can't do, so far, requires intuition, the ability to find a link among seemingly dissimilar ideas. When, in 1997, IBM's Big Blue computer was the first machine to win a tournament with a reigning chess champion—Garry Kasparov—it did so by playing "what if." What if it moved this piece, and then Kasparov moved that piece, and so on through six to eight moves starting from 20 or more contingencies. Big Blue was capable of 200 million positions a second. It's much like doing the Towers of Hanoi—very fast. Human chess champions, on the other hand, say they uniformly avoid the what-if approach in favor of patterns. They will look at the board, and the years of experience buried in their brains see patterns of pieces that reveal a strategy to a win. Or, in Kasparov's case, a loss.

Humans are very good at insight and hunches. Computing is very good at doing the same thing over and over again. Remember that when your computer seems to be taking too long to do something. It's just plodding along in its own manner. It doesn't have your skills.

How a Processor Tracks Numbers

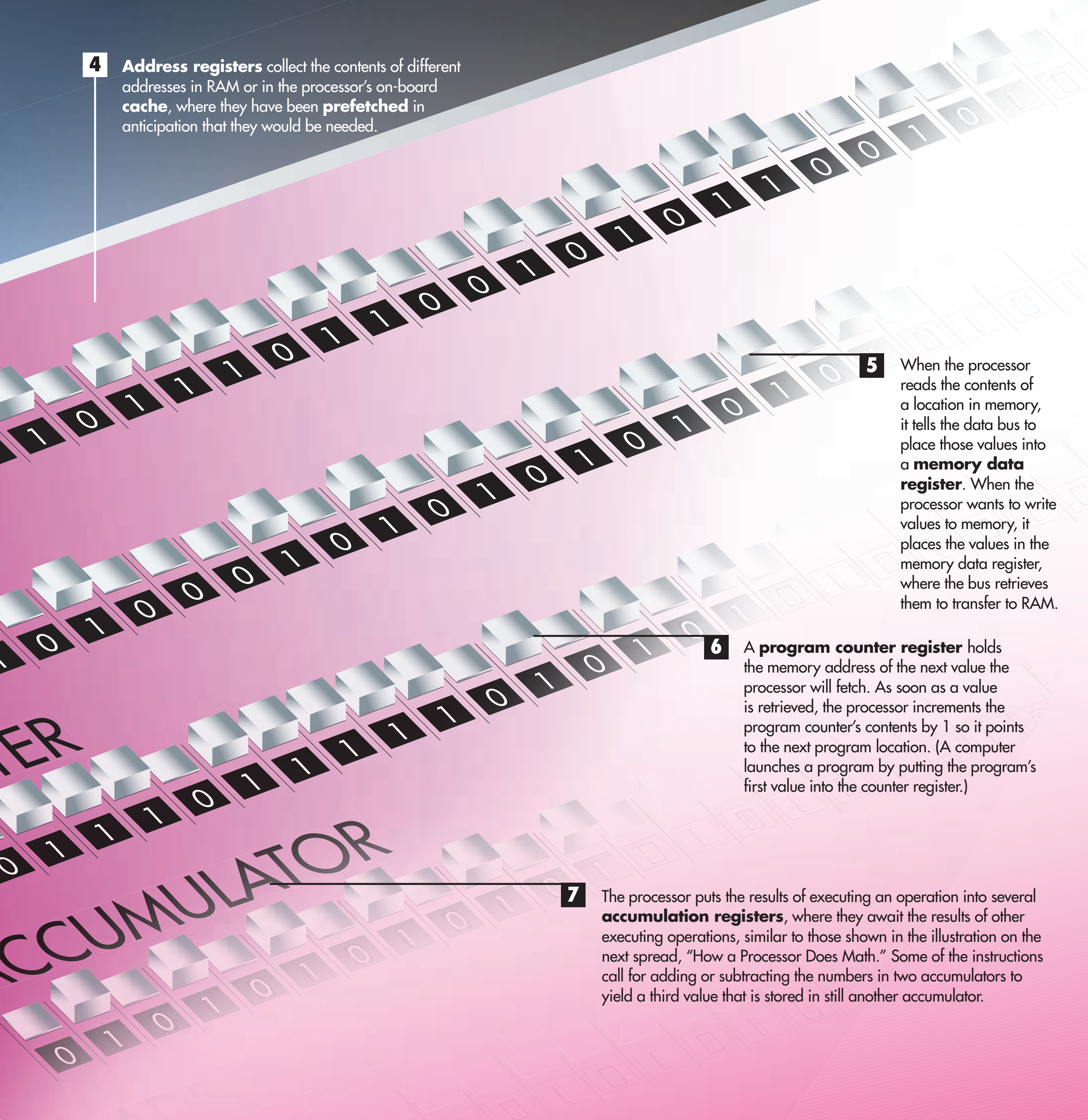
Few of us can do complex math in our heads. Even for something as simple as adding several rows of numbers, we need a pencil and paper to keep track of which numbers go where. Microprocessors are not all that different in this regard. Although they are capable of performing intricate math involving thousands of numbers, they, too, need notepads to keep track of their calculations. Their notepads are called **registers**, and their pencils are pulses of electricity.

1 A microprocessor's registers consist of reserved sections of transistors in the faster memory inside the microprocessor. There the processor's **arithmetic logic unit** (ALU), in charge of carrying out math instructions, and the **control unit**, which herds instructions and data through the processor, have quick access to the registers. The size of the registers determines how much data the processor can work with at one time. Most PCs have registers with 32 or 64 bits for data.



2 The processor's **control unit** directs the fetching and execution of program instructions. (See "How a Microprocessor Moves Data", on pages 36-37.) It uses an electrical signal to fetch each instruction, decodes it, and sends another control signal to the arithmetic logic unit telling the ALU what operation to carry out.

3 With each clock cycle—the thin unit of time during which the different components of a computer can do one thing—the processor writes or reads the values of the bits by sending or withholding a pulse of electricity to each. Each chunk of binary numbers is only that. They have no labels to identify them as instructions, data, values going into a computation, or the product of executing instructions. What the values represent depends on which registers the control unit uses to store them.



4 **Address registers** collect the contents of different addresses in RAM or in the processor's on-board **cache**, where they have been **prefetched** in anticipation that they would be needed.

5 When the processor reads the contents of a location in memory, it tells the data bus to place those values into a **memory data register**. When the processor wants to write values to memory, it places the values in the memory data register, where the bus retrieves them to transfer to RAM.

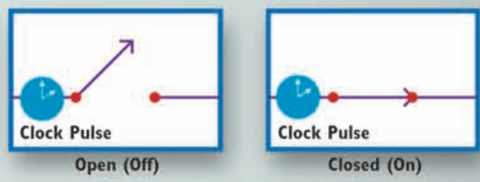
6 A **program counter register** holds the memory address of the next value the processor will fetch. As soon as a value is retrieved, the processor increments the program counter's contents by 1 so it points to the next program location. (A computer launches a program by putting the program's first value into the counter register.)

7 The processor puts the results of executing an operation into several **accumulation registers**, where they await the results of other executing operations, similar to those shown in the illustration on the next spread, "How a Processor Does Math." Some of the instructions call for adding or subtracting the numbers in two accumulators to yield a third value that is stored in still another accumulator.

How a Processor Does Math

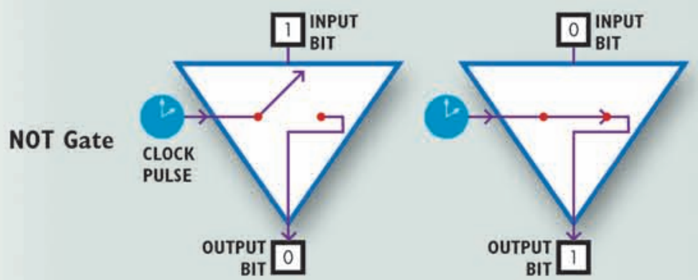
1 All information—words and graphics as well as numbers—is stored in and manipulated by a PC in the form of binary numbers. In the binary numerical system, there are only two digits—0 and 1. All numbers, words, and graphics are formed from different combinations of those digits.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010



2 Transistor switches are used to manipulate binary numbers because there are two possible states of a switch, open (off) or closed (on), which nicely matches the two binary digits. An open transistor, through which no current is flowing, represents a 0. A closed transistor, which allows a pulse of electricity regulated by the PC's clock to pass through, represents a 1. (The computer's clock regulates how fast the computer works. The faster a clock ticks, causing pulses of electricity, the faster the computer works. Clock speeds are measured in **megahertz**, or millions of ticks per second.) Current passing through one transistor can be used to control another transistor, in effect turning the switch on and off to change what the second transistor represents. Such an arrangement is called a **gate** because, like a fence gate, the transistor can be open or closed, allowing or stopping current flowing through it.

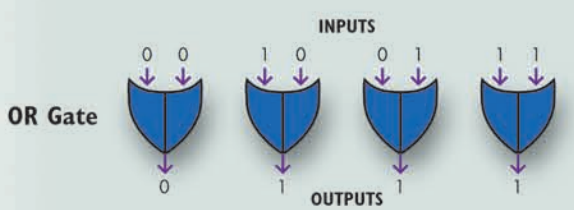
3 The simplest operation that can be performed with a transistor is called a **NOT logic gate**, made up of only a single transistor. This NOT gate is designed to take one *input* from the clock and one from another transistor. The NOT gate produces a single *output*—one that's always the opposite of the input from the transistor. When current from another transistor representing a 1 is sent to a NOT gate, the gate's own transistor switches open so that a pulse, or current, from the clock can't flow through it, which makes the NOT gate's output 0. A 0 input closes the NOT gate's transistor so that the clock pulse passes through it to produce an output of 1.



NOT Gate Operations

INPUT FROM CLOCK	INPUT FROM OTHER TRANSISTOR	OUTPUT
1	1	0
1	0	1

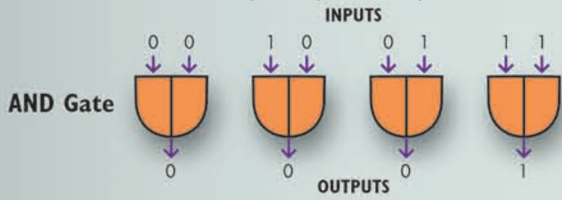
4 NOT gates strung together in different combinations create other logic gates, all of which have a line to receive pulses from the clock and two other input lines for pulses from other logic gates. The **OR gates** create a 1 if either the first or second input is a 1, and put out a 0 only if both inputs are 0.



OR Gate Operations

1ST INPUT	2ND INPUT	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

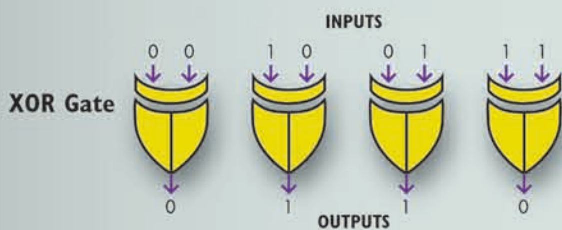
5 An AND gate outputs a 1 only if *both* the first and the second inputs are 1s.



AND Gate Operations

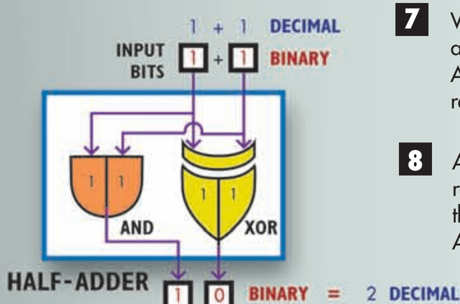
1ST INPUT	2ND INPUT	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

6 An **XOR gate** puts out a 0 if *both* the inputs are 0 or if *both* are 1. It generates a 1 only if *one* of the inputs is 1 and the *other* is 0.



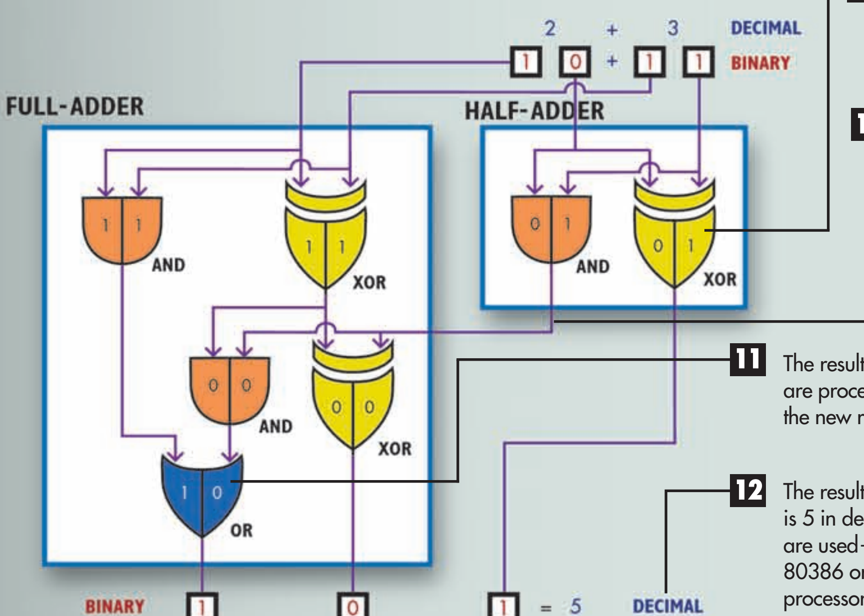
XOR Gate Operations

1ST INPUT	2ND INPUT	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0



7 With different combinations of logic gates, a computer performs the math that is the foundation of all its operations. This is accomplished with gate designs called **half-adders** and **full–adders**. A half-adder consists of an XOR gate and an AND gate, both of which receive the same input representing a one-digit binary number. A full-adder consists of half-adders and other switches.

8 A combination of a half-adder and a full-adder handles larger binary numbers and can generate results that involve carrying over numbers. To add the decimal numbers 2 and 3 (10 and 11 in the binary system), first the half-adder processes the digits on the right side through both XOR and AND gates.



9 The result of the XOR operation (1) becomes the rightmost digit of the result.

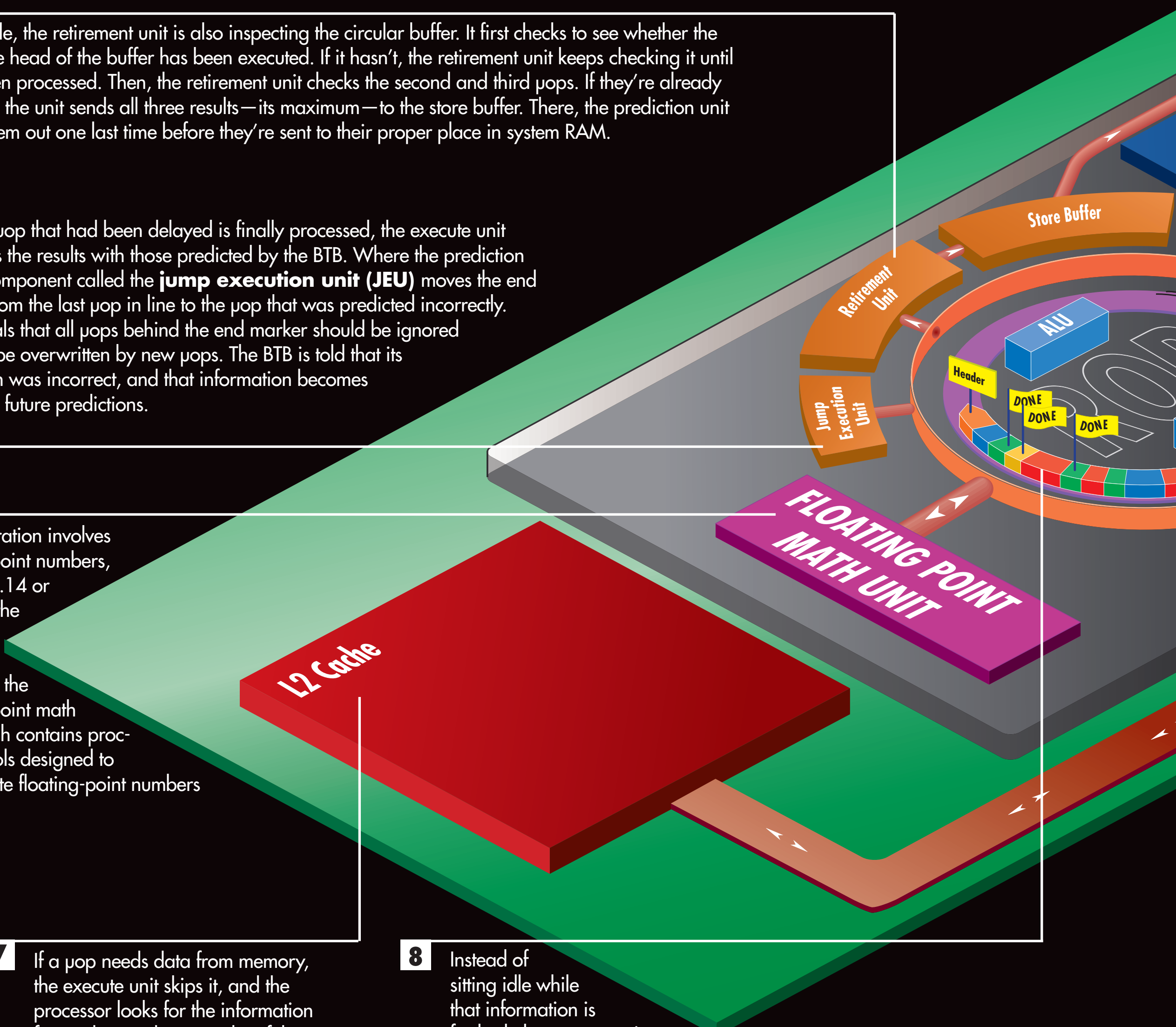
10 The result of the half-adder's AND operation (0) is sent to XOR and AND gates in the full-adder. The full-adder also processes the left-hand digits from 11 and 10, sending the results of both of the operations to another XOR gate and another AND gate.

11 The results from XORing and ANDing the left-hand digits are processed with the results from the half-adder. One of the new results is passed through an OR gate.

12 The result of all the calculations is 101 in binary, which is 5 in decimal. For larger numbers, more full-adders are used—one for each digit in the binary numbers. An 80386 or later processor, including today's Pentium class processors, uses 32 full-adders.

How a Processor Moves Data

TODAY'S MICROPROCESSORS have more than 100 billion transistors. Taking a walk through one of them could get a person hopelessly lost. Old or new, however, how a processor performs its most basic functions hasn't changed. They may have as many as eight execution cores and multiple caches—you can look at those on pages **XXX–xxx**—but, like the old single-core Pentium III processor illustrated here, they all face the same problem of how to move data quickly and with nary a hitch.



11 Meanwhile, the retirement unit is also inspecting the circular buffer. It first checks to see whether the µop at the head of the buffer has been executed. If it hasn't, the retirement unit keeps checking it until it has been processed. Then, the retirement unit checks the second and third µops. If they're already executed, the unit sends all three results—its maximum—to the store buffer. There, the prediction unit checks them out one last time before they're sent to their proper place in system RAM.

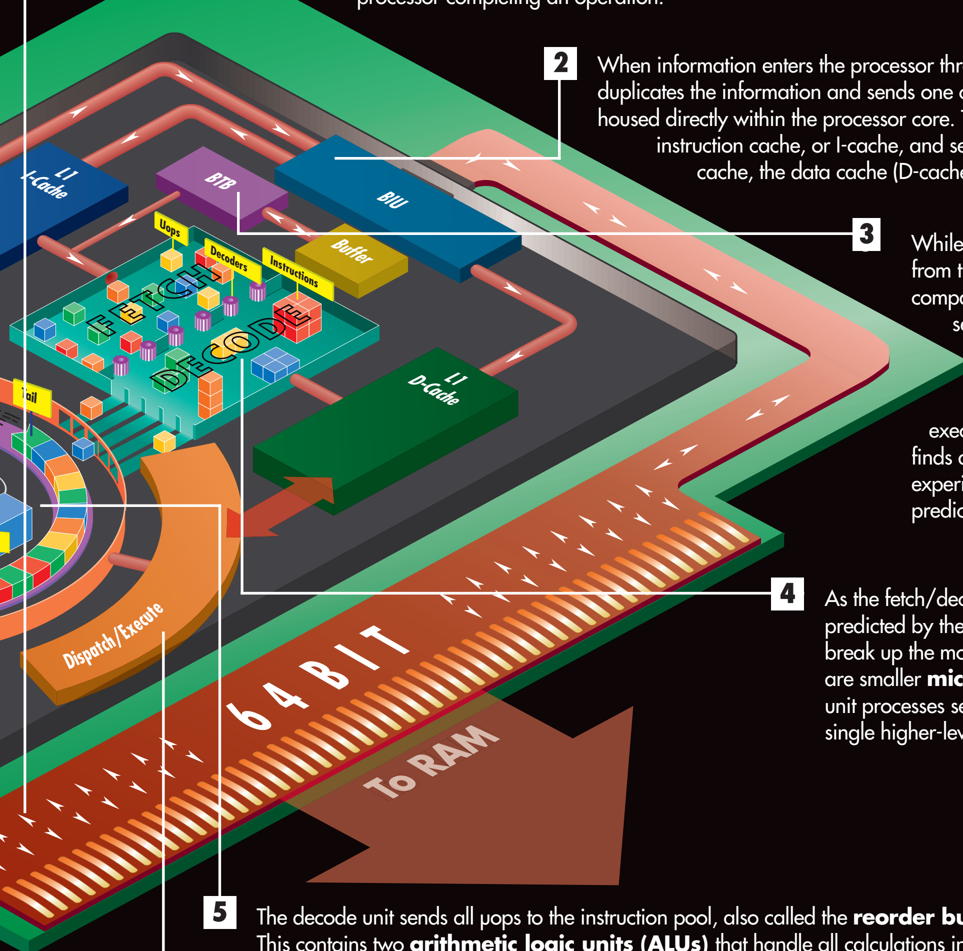
10 When a µop that had been delayed is finally processed, the execute unit compares the results with those predicted by the BTB. Where the prediction fails, a component called the **jump execution unit (JEU)** moves the end marker from the last µop in line to the µop that was predicted incorrectly. This signals that all µops behind the end marker should be ignored and can be overwritten by new µops. The BTB is told that its prediction was incorrect, and that information becomes part of its future predictions.

9 If an operation involves floating-point numbers, such as 3.14 or .33333, the ALUs hand off the job to the floating-point math unit, which contains processing tools designed to manipulate floating-point numbers quickly.

7 If a µop needs data from memory, the execute unit skips it, and the processor looks for the information first in the nearby L1 cache. If the data isn't there, the processor checks the next cache level, L2 in this case. Cache size and organization vary based on the specific processor design, but each level of cache increases in both capacity and time needed to fetch data from it.

8 Instead of sitting idle while that information is fetched, the execute unit continues inspecting each µop in the buffer for those it can execute. This is called **speculative execution** because the order of µops in the circular buffer is based on the BTB's branch predictions. The unit executes up to five µops simultaneously. When the execution unit reaches the end of the buffer, it starts at the head again, rechecking all the µops to see whether any have finally received the data they need to be executed.

1 A processor and its integrated cache share the same interface to the computer's information. Program code or data manipulated by that code move in and out of the chip at the PC's maximum bus speed. Much of a computer's architecture is structured to alleviate the bus bottleneck by minimizing the times a clock cycle—the smallest time in which a computer can do anything—ticks away without the processor completing an operation.



2 When information enters the processor through the **bus interface unit (BIU)**, the BIU duplicates the information and sends one copy to the CPU's closest data caches that are housed directly within the processor core. The BIU sends program code to the Level 1 instruction cache, or I-cache, and sends data to be used by the code to another L1 cache, the data cache (D-cache).

3 While the fetch/decode unit is pulling in instructions from the I-cache, the **branch target buffer (BTB)** compares each instruction with a record in a separate set-aside buffer to see whether any instruction has been used before. The BTB is looking in particular for instructions that involve **branching**, a situation in which the program's execution could follow one of two paths. If the BTB finds a branch instruction, it predicts, based on past experience, which path the program will take. The predictions are better than 90 percent accurate.

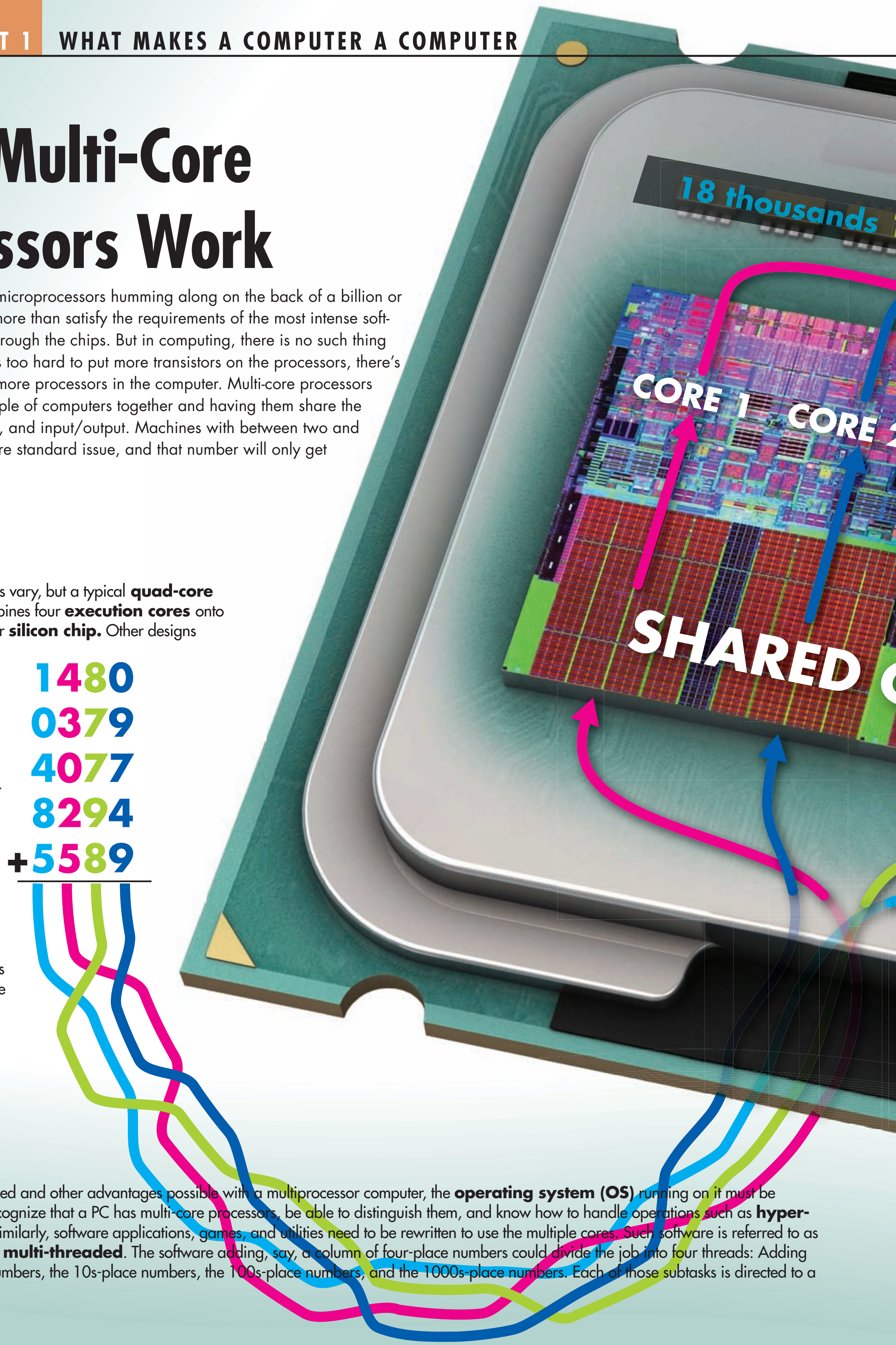
4 As the fetch/decode unit pulls instructions in the order predicted by the BTB, three decoders working in parallel break up the more complex instructions into **µops**, which are smaller **micro-operations**. The dispatch/execution unit processes several µops faster than it processes a single higher-level instruction.

5 The decode unit sends all µops to the instruction pool, also called the **reorder buffer**. This contains two **arithmetic logic units (ALUs)** that handle all calculations involving integers. The ALUs use a circular buffer, with a head and tail, that contains the µops in the order in which the BTB predicted they would be needed.

6 The dispatch/execute unit checks each µop in the buffer to see whether it has all the information needed to process it, and when it finds a µop ready to process, the unit executes it, stores the result in the micro-op itself, and marks it as done.

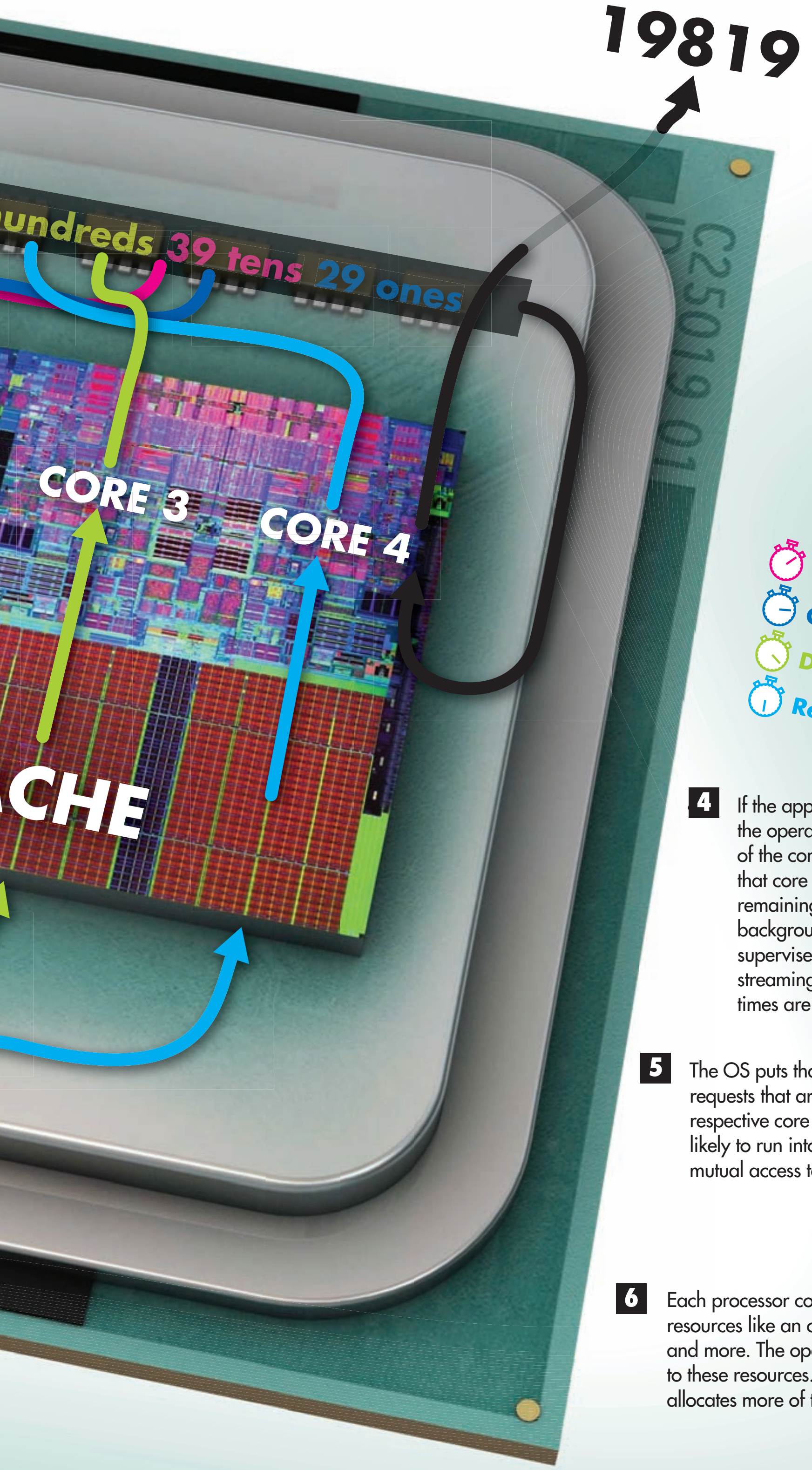
How Multi-Core Processors Work

YOU'D THINK the microprocessors humming along on the back of a billion or so transistors would more than satisfy the requirements of the most intense software you can push through the chips. But in computing, there is no such thing as "enough." So if it's too hard to put more transistors on the processors, there's another solution: Put more processors in the computer. Multi-core processors are like bolting a couple of computers together and having them share the same memory, power, and input/output. Machines with between two and ten processor cores are standard issue, and that number will only get bigger.



1 Specific designs vary, but a typical **quad-core** processor combines four **execution cores** onto a single **die**, or **silicon chip.** Other designs spread their cores across two dies. Regardless, these identical cores are the heart of any microprocessor and the part that does the heavy work of executing instructions from software. The wildly colored areas above the cores in the photo are supporting the circuitry.

2 To gain the speed and other advantages possible with a multiprocessor computer, the **operating system (OS)** running on it must be designed to recognize that a PC has multi-core processors, be able to distinguish them, and know how to handle operations such as **hyper-threading**. Similarly, software applications, games, and utilities need to be rewritten to use the multiple cores. Such software is referred to as **threaded**, or **multi-threaded**. The software adding, say, a column of four-place numbers could divide the job into four threads: Adding the 1s-place numbers, the 10s-place numbers, the 100s-place numbers, and the 1000s-place numbers. Each of those subtasks is directed to a different core.



3 When the subtasks exit the cores, the operating system combines the threads into a single number, and sends that operation to one of the cores for execution.

4 If the application software isn't equipped to work in multiple cores, the operating system can still take advantage of them. It picks one of the cores to run the software and creates an **affinity** between that core and the program. It then creates affinities between the remaining cores and various tasks. A second core may handle background operations, such as disk optimizing; a third core might supervise a download; and the fourth could render a video that's streaming from the Internet. Neither the operations nor their finish times are affected by the processing going on in the other cores.

5 The OS puts that operation into a **time-staggered queue** along with requests that are going to other cores. Each of the operations enters its respective core on different clicks of the computer's clock so they are less likely to run into each other or cause a traffic jam in the areas they have mutual access to.

6 Each processor core isn't completely distinct. They might share access to resources like an on-die graphics processor, memory caches (as shown here), and more. The operating system can determine how each core shares access to these resources. If only one core is active, for example, the OS dynamically allocates more of the shared cache to that core.

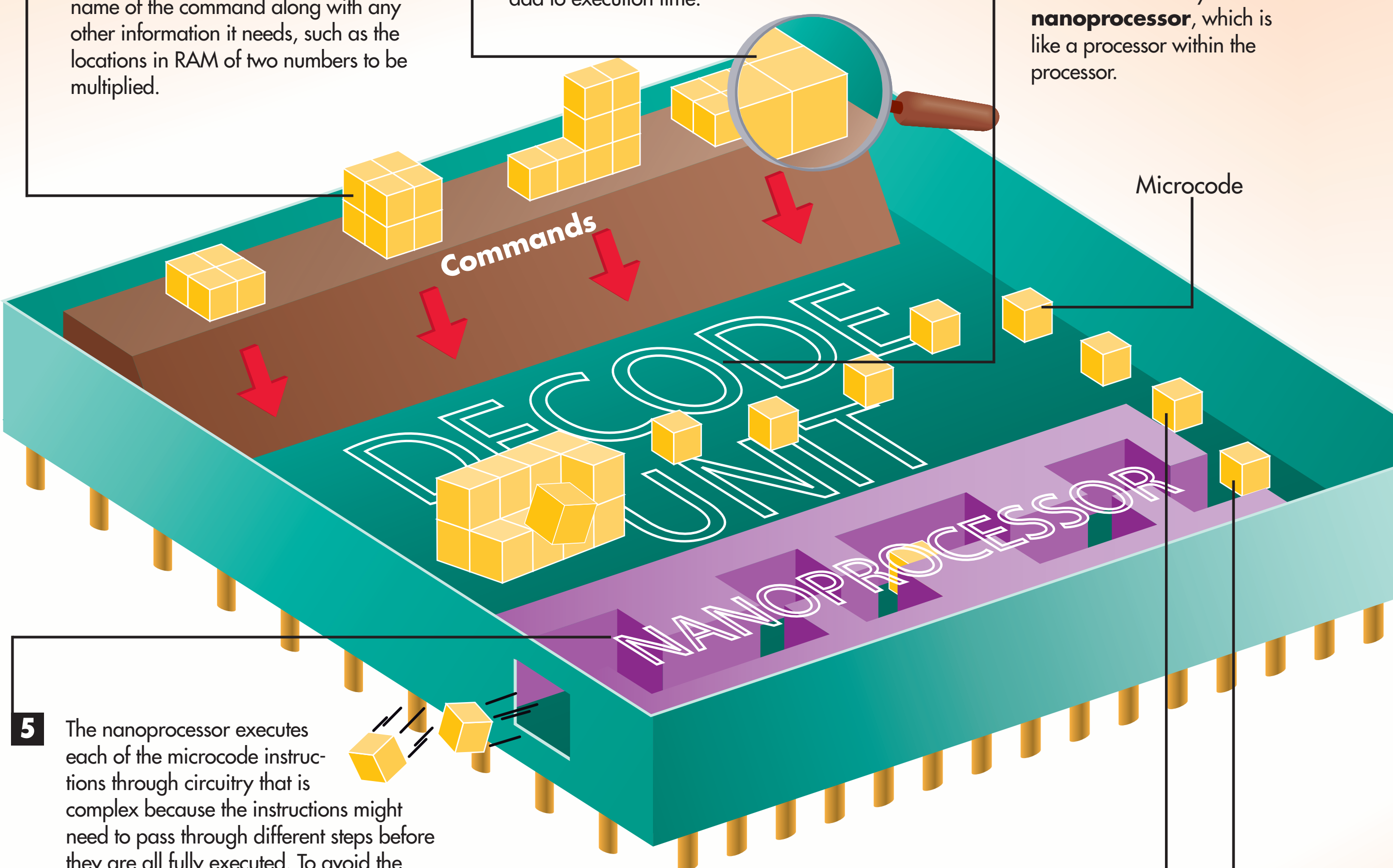
How Desktop CPUs Keep It Complex

THE CHORES that desktop computers face—putting out a company payroll, rendering blueprints for a new factory—require solving a lot of complex computations. That's why the everyday desktop processors, modeled after Intel's ***x*86** line, use **complex instruction set computing (CISC)**. It uses complicated, intertwined instructions to bite off manageable problems that it solves and returns to a still-churning solution.

1 Built into a CISC microprocessor's read-only memory is a large set of commands containing several subcommands that must be carried out to complete a single operation, such as multiplying two numbers or moving text. When software makes a demand, the program sends the processor the name of the command along with any other information it needs, such as the locations in RAM of two numbers to be multiplied.

2 Because the CISC commands are not all the same size, the microprocessor checks how many bytes of processing room the command needs and then sets aside that much internal memory. There are also several different ways the commands can be loaded and stored, and the processor must determine the correct way to load and store each of the commands. Both these preliminary tasks add to execution time.

3 The processor sends the command requested by the software to a **decode unit**, which translates the complex command into **microcode**, a series of smaller instructions that are executed by the **nanoprocessor**, which is like a processor within the processor.



4 With **multicore** processors, more than one instruction can be carried out simultaneously, and the results are united when each processor core completes its instructions.

5 The nanoprocessor executes each of the microcode instructions through circuitry that is complex because the instructions might need to pass through different steps before they are all fully executed. To avoid the added heat and power use typical of CISC processing, some *x*86 processors are implementing RISC methods beneath a transitional CISC layer.

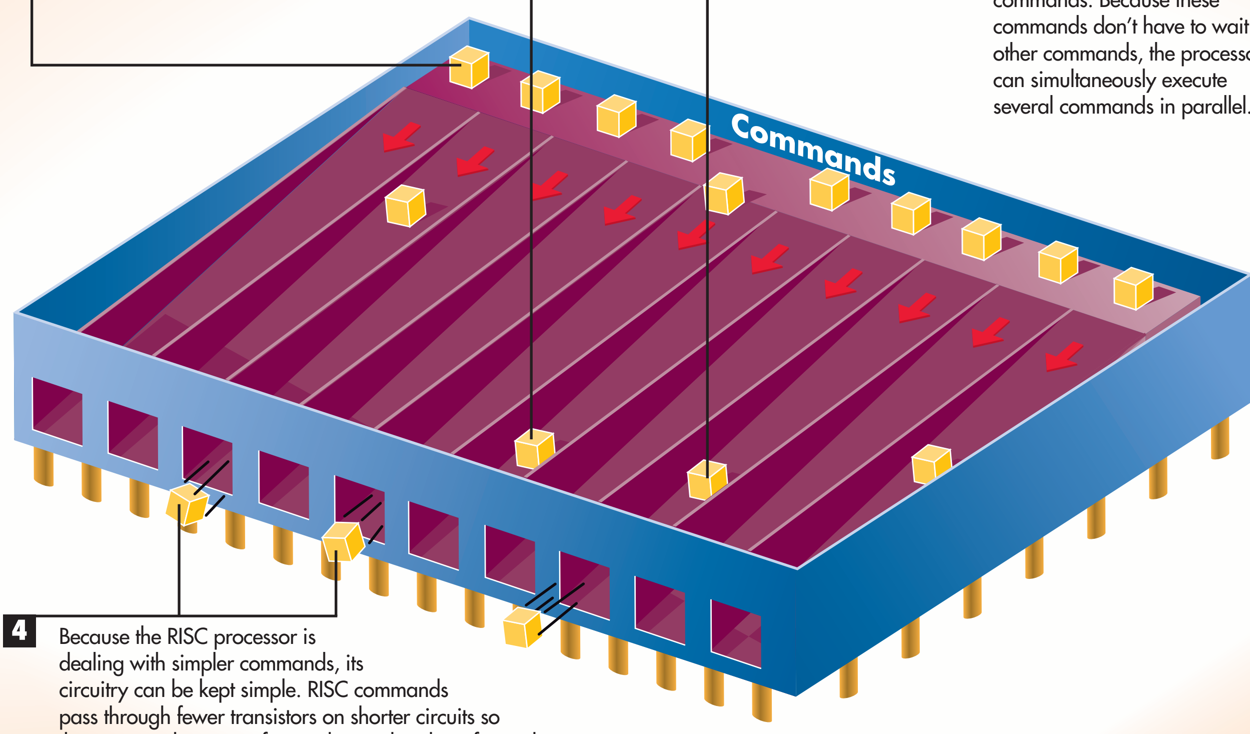
How Mobile CPUs Keep It Simple

THE JOBS that mobile devices take on—dialing a phone number, displaying web pages, playing Angry Birds—are simple as far as computing goes. So they use a simpler system to execute code: **reduced instruction set computing (RISC)**. It deals with instructions that are already broken into chewable pieces, which it quickly gnashes before it spits out the result, using less power and generating less heat in the process. RISC is the basis for the many variants of **ARM architecture** prevalent in 75 percent of mobile products, including everything from tablets to smartphones to digital cameras.

1 Command functions built into an RISC processor consist of several small, discrete instructions that perform only a single job. Apps tell the processor which combination of its smaller commands to execute in order to complete a larger operation.

2 All RISC commands are the same size, and there is only one way they can be loaded and stored. Because each command is already a form of microcode, RISC processors don't require the extra step of passing instructions through a decode unit to translate them into simpler microcode. This lets the processor load commands for execution faster than in a CISC processor.

3 During the compilation of RISC apps, the compiler determines which commands will not depend on the results of other commands. Because these commands don't have to wait on other commands, the processor can simultaneously execute several commands in parallel.



4 Because the RISC processor is dealing with simpler commands, its circuitry can be kept simple. RISC commands pass through fewer transistors on shorter circuits so the commands execute faster. The result is that often only one CPU **clock cycle** is needed for each instruction. The number of cycles needed to complete a full operation depends on the number of small commands that make up that operation, but generally it will be faster than a CISC processor.

Index

Symbols

A

B

D

E

F

G

J

K

L

M

S

T

X-Y

Z