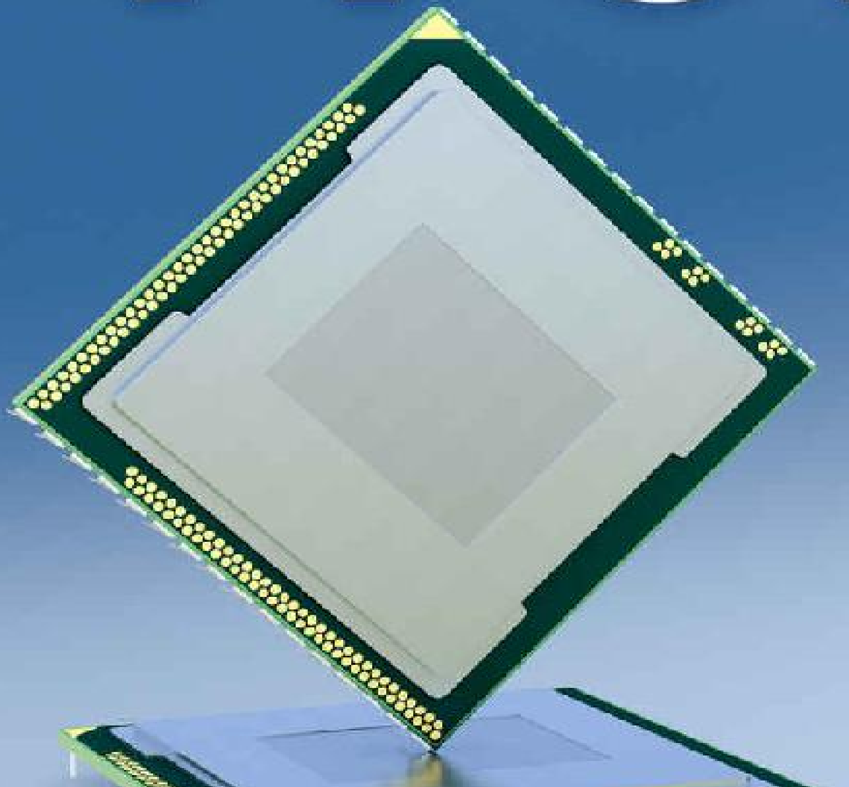
HOW COMPUTERS WORK



THE BASIC PRINCIPLES OF COMPUTERS FOR EVERYONE

JAMES WELLS

How Computers Work

The Basic Principles of Computers For Everyone

By

James Wells

Table of Contents

Introduction

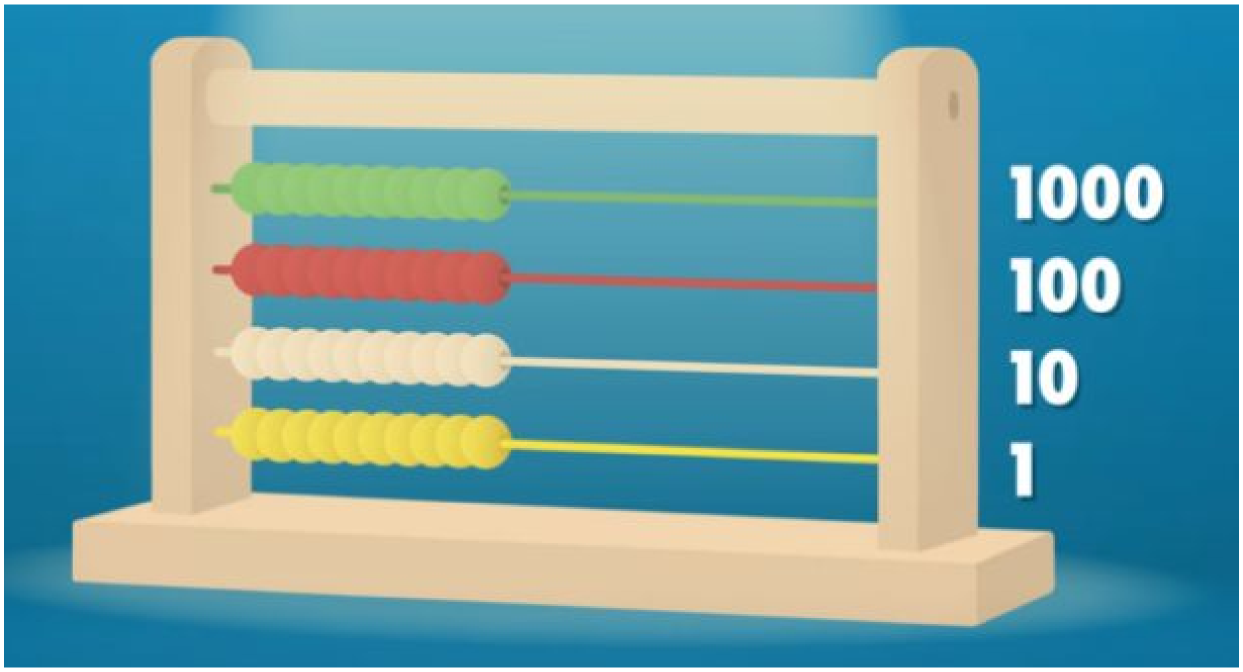
Computers really have allowed us to do some pretty amazing things: global telecommunications, international commerce, global transportation, breakthrough in medicine, distributed education, online shopping, online dating, just the Internet in general. Computers are allowing us to explore our own world and other worlds, and of course some seemingly mundane things like permitting us to spy on our pets from work or communicate with our friends in a nearly indecipherable stream of emoji! But don't call computers magical. They ARE NOT magical. So before we get into what this book is about, it might be useful you to know what this book is not going to teach you. This book is not about how to program. Programming is a really crucial aspect of computer software, and we will get to the rules that guide the logic of hardware and software design. But this book is not about how to program an Arduino Uno to water your plant or how to change the CSS on your grandma's sewing blog so visitors' cursors turn into kittens. This book also is not a computing book. Or at least how computing is thought of in the U.S Computing here is a goal – it is what computers do. Our goal for this book is much broader. But computing means other things in other countries. It is all pretty confusing. But we are going to look at the history of computers… even before humans had electricity. The book is going retrace the design decisions that have given us our president-day components. Book is about how Operating Systems work… or do not work… how smartphones and other smart devices are… well getting smarter. Computers are these amazing devices that are always making our lives easier and they inarguably have become pivotal in our society. From our cars and thermostats to pacemakers and cellphones, computers are everywhere, and by the end of this book you will have better understanding and appreciation for how far we have come and how far they may take us.

1.Early Computing

So in this chapter, we’re going to take a look at computing origins, because even though our digital computers are relatively new, the need for computation is not. Since the start of civilization itself, humans have had an increasing need for special devices to help manage laborious tasks, and as the scale of society continued to grow, these computational devices began to play a crucial role in amplifying our mental abilities. From the abacus and astrolabe to the difference engine and tabulating machine, we’ve come a long way to satisfying this increasing need, and in the process completely transformed commerce, government, and daily life.

Computers are the lifeblood of today’s world. If they were to suddenly turn off, all at once, the power grid would shut down, cars would crash, planes would fall, water treatment plants would stop, stock markets would freeze, trucks with food wouldn’t know where to deliver, and employees wouldn’t get paid. Even many non-computer objects – like DFTBA shirts and the chair you are probably sitting on – are made in factories run by computers. Computing really transformed nearly every aspect of our lives. Advances in manufacturing during the Industrial Revolution brought a new scale to human civilization – in agriculture, industry and domestic life. Mechanization meant superior harvests and more food, mass produced goods, cheaper and faster travel and communication, and usually a better quality of life. And computing technology is doing the same right now – from automated farming and medical equipment, to global telecommunications and educational opportunities, and new frontiers like Virtual Reality and Self Driving Cars. We are living in a time likely to be remembered as the Electronic Age. With billions of transistors in just your smartphones, computers can seem pretty complicated, but really, they are just simple machines that perform complex actions through many layers of abstraction. So in this book, we are going break down those layers, and build up from simple 1’s and 0’s, to logic units, CPUs, operating systems, the entire internet and beyond. And do not worry, in the same way someone buying t-shirts on a webpage doesn’t need to know how that webpage was programmed, or the web designer doesn’t need to know how all packets are routed, or router engineers do not need to know about transistor logic.

The earliest recognized device for computing was the abacus, invented in Mesopotamia around 2500 BCE. It is essentially a hand operated calculator, that helps add and subtract many numbers. It also stores the current state of the computation, much like your hard drive does today. The abacus was created because, the scale of society had become grater that what a single person could keep and manipulate in their mind. There might be thousands of people in a village or tens of thousands of cattle. There are many variants of the abacus, but let's look at a really basic version with each row representing a different power of ten. So each based on the bottom row represents a single unit, in the next row they represent 10, the row above 100, and so on. You can see simple abacus machine at the image.



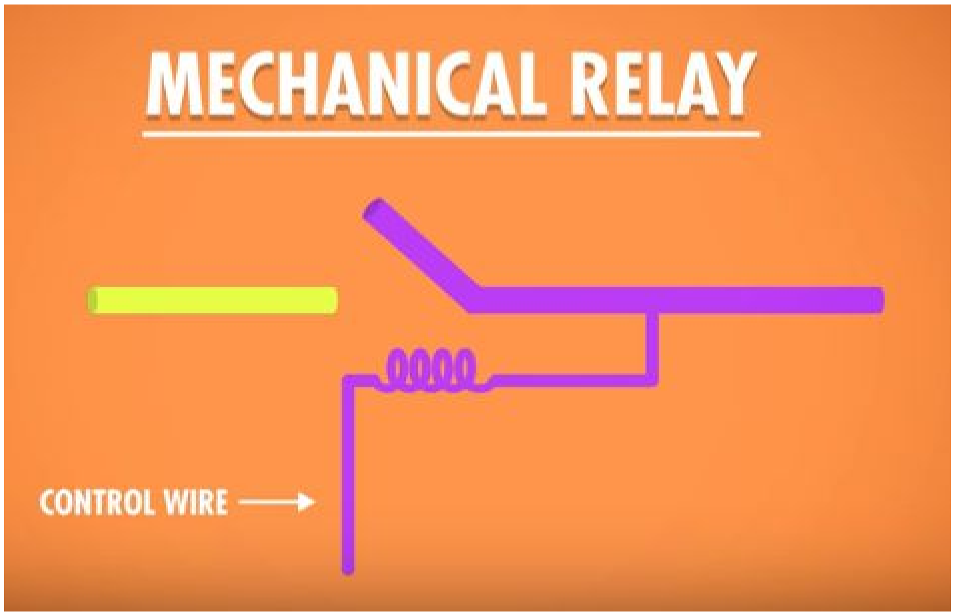
Over the next 4000 years, humans developed all sorts of clever computing devices, like the astrolabe, which enabled ships to calculate their latitude at sea. Or the slide rule, for assisting with manipulation and division. And there are literally hundreds of types of clocks created that could be used to calculate sunrise, tides, positions of celestial bodies, and even just time. Each one of these devices made something that was previously laborious to calculate much faster, and often more accurate – it lowered the barrier to entry, and at the same time amplified our mental abilities. As early computer pioneer Charles Babbage said: "At each increase of knowledge, as well as on the contrivance of every new tool, human labor becomes abridged." However, none of these devices were called "Computers". The earliest documented use of the word "computer" is from 1613, in a book by Richard Braithwait. And it was not a machine at all – it was a job title. Braithwait said, "I have read the truest computer of times, and the best arithmetician that ever breathed, and he reduced thy days into a short number". In those days, computer was a person who did calculations, sometimes with the help of machines, but often not. This job title persisted until the late 1800's, when the meaning of computer started shifting to refer to devices. Notable among these devices was the Step Reckoner, built by German polymath Gottfried Leibniz in 1694. Leibniz said "… it is beneath the dignity of excellent men to waste their time in calculation when any peasant could do the work just as accurately with the aid of machine." It worked kind of like the odometer in your car, which is really just a machine for adding up the number of miles your car has driven. The device had a series of gears that turned; each gear had ten teeth, to represent the digits from 0 to 9. Whenever a gear bypassed nine, it rotated back to 0 and advanced the adjacent gear by one tooth. Kind of when hitting 10 on that basic abacus. This worked in reverse when doing subtraction too. When some clever mechanical tricks, the Step Reckoner was also able to multiply and divide numbers. Multiplications and divisions are really just many additions and subtractions. For example, if we want to divide 17 by 5, we just subtract 5, then 5, then 5 again, and then we can't subtract any more 5's… so we know 5 goes into 17 three times, with 2 left over. The Step Reckoner was able to do this in an automated way, and was the first machine that could do all four of these operations. And this design was so successful it was used for the next three centuries of calculator design. Unfortunately, even the mechanical calculators, most real world problems required many steps of computation before an answer was determined. It could take hours or days to generate a single result. Also these hand-crafted machines were expensive, and not accessible to most of population. So before 20th century, most people experienced computing through pre-computed tables assembled by those amazing "human computers" we talked about. So if we needed to know the square root of 8 million 6 hundred and 75 thousand 3 hundred and 9, instead of spending all day hand-cranking your reckoner, you could look it up in a huge book full of square root tables in a minute or so. Speed and accuracy is particularly important on the battlefield, and so militaries were among the first to apply computing to complex problems. A particularly difficult problem is accurately firing artillery shells, which by the 1800s could travel well over a kilometer (or a bit more than a mile). Add this wind conditions, temperature and atmospheric pressure, and even hitting something as large as a ship was difficult. Range Tables were created that allowed gunners to look up environmental conditions and the distance they wanted to fire and the table would tell them the angle to set the canon. These Range Tables worked so well, they were used well into World War Two. The problem was, if you changed the design of the canon or of the shell, a whole new table had to be computed, which was massively time consuming and inevitably led to errors. Charles Babbage acknowledged this problem in 1822 in a paper to Royal Astronomical society entitled: "Note on the application of machinery to the computation of astronomical and mathematical tables". Let's go to the through bubble. Charles Babbage proposed a new mechanical device called Difference Engine, a much more complex machine that could approximate polynomials. Polynomials describe the relationship between several variables – like the range and air pressure, or amount of pizza you eat and happiness. Polynomials could also be used to approximate logarithmic and trigonometric functions, which are a real hassle to calculate by hand. Babbage started construction in 1823, and over the next two decades, tried to fabricate and assemble 25,000 components, collectively weighing around 15 tons. Unfortunately, the project was ultimately abandoned. But in 1991, historians finished constructing a Difference Engine based on Babbage's drawings and writing – and it worked! But more importantly, during construction of the Difference Engine, Babbage imagined an even more complex machine – the Analytical Engine. Unlike the Difference Engine, Step Reckoner and all other computational devices before it – the Analytical Engine was a "general purpose computer". And it could be used for many things, not just one particular computation; it could be given data and run operations in sequence; it had memory and even a primitive printer. Like the Difference Engine it was ahead of its time, and was never fully constructed. However, the idea of an "automatic computer" – one that could guide itself through a series of operations automatically, was a huge deal, and would foreshadow computer programs. English mathematician Ada Lovelace wrote hypothetical programs for the Analytical Engine, saying, “A new, a vast, and a powerful language is developed for the future use of analysis.” For her work, Ada is often considered the world’s first programmer. The analytical Engine would inspire, arguable, the first generation of computer scientists, who incorporated many of Babbage’s ideas in their machines. This is why Babbage is often considered the “father of computing”. Thanks Thought Bubble! So by the end of the 19th century, computing devices were used for special purpose tasks in the sciences and engineering, but rarely seen in business, government or domestic life. However, the US government forced a serious problem for its 1890 census that demanded the kind of efficiency that only computers could provide. The US Constitution requires that a census be conducted every ten years, for the purposes of distributing federal funds, representation in congress and good stuff like that. And by 1880, the US population was booming mostly due to immigration. That census seven years to manually compile and by the time it was completed, it was already out of date – and it was predicted that the 1890 census would take 13 years to compute. That is a little problematic when it’s required every decade! The census bureau turned to Herman Hollerith, who had built a tabulation machine. His machine was “electro-mechanical” – it used traditional mechanical system for keeping count, like Leibniz’s Step Reckoner—but coupled them with electrically-powered components. Hollerith’s machine used punch cards which were paper cards with a grid of locations that can be punched out to represent data. For example, there was series of holes for marital status. If you were married, you would punch out the married spot, then when the card was inserted into Hallerith’s machine, little metal pins would come down over the card – if a spot was punched out, the pin would pass through the hole in the paper and into a little vial of mercury, which completed the circuit. This now completed circuit powered an electric motor, which turned a gear to add one, in this case, to the “married” total. Hollerith’s machine was roughly 10x faster than manual tabulations, and the Census was completed in just two and half years – saving the census office millions of dollars. Businesses began recognizing the value of computing, and saw its potential to boost profits by improving labor – and data-intensive tasks, like accounting, insurance appraisals, and inventory management. To meet this demand, Hollerith founded The

Tabulating Machine Company, which later merged with other machine makers in 1924 to become The International Buisness Machines Corporation or IBM – which you've probably heard of. These electro-mechanical "business machines" were a huge success, transforming commerce and government, and by the mid-1900s, the explosion in world and the rise of globalized trade demanded even faster and more flexible tools for processing data, setting the stage for digital computers, which we will look at in the next chapter.

2.Electronic Computing

Our last chapter brought us to the start of the 20th century, where early, special purpose computing devices, like tabulating machines, were a huge boon to governments and business – aiding, and sometimes replacing, rote manual tasks, but the scale of human systems continued to increase at an unprecedented rate. The first half of the 20th century saw the world's population almost double. World War 1 mobilized 70 million people, and World War 2 involved more than 100 million. Global trade and transit networks became interconnected like never before, and the sophistication of our engineering and scientific endeavors reached new heights – we even started to seriously consider visiting other planets. And it was this explosion of complexity, bureaucracy, and unlimited data that drove an increasing need for automation and computation. Soon those cabinet-sized electro-mechanical computers grew into room-sized behemoths that were expensive to maintain and prone to errors. And it was these machines that would set the stage for future innovation.

One of the largest electro-mechanical computers built was the Harvard Mark I, completed in 1944 by IBM for Allies during World War 2. It contained 765,000 components, three million connections and five hundred miles of wire. To keep its internal mechanics synchronized, it used a 50-foot shaft running right through the machine driven by a five horsepower motor. One of the earliest uses for this technology was running simulations for the Manhattan Project. The brains of these huge electro-mechanical beasts were relays: electrically-controlled mechanical switches. In a relay, there is a control wire that determines whether a circuit is opened or closed. The control wire connects to a coil of wire inside the relay. When current flows through the coil, an electromagnetic field is created, which in turn, attracts a metal arm inside the relay, snapping it shut and completing the circuit. You can think of a relay like a water faucet. The control wire is like the faucet handle. Open and water flows through the pipe. Close the faucet, and the flow of water stops. Relays are doing the same thing, just with electrons instead of water. You can see the simple construction of relay in the image.



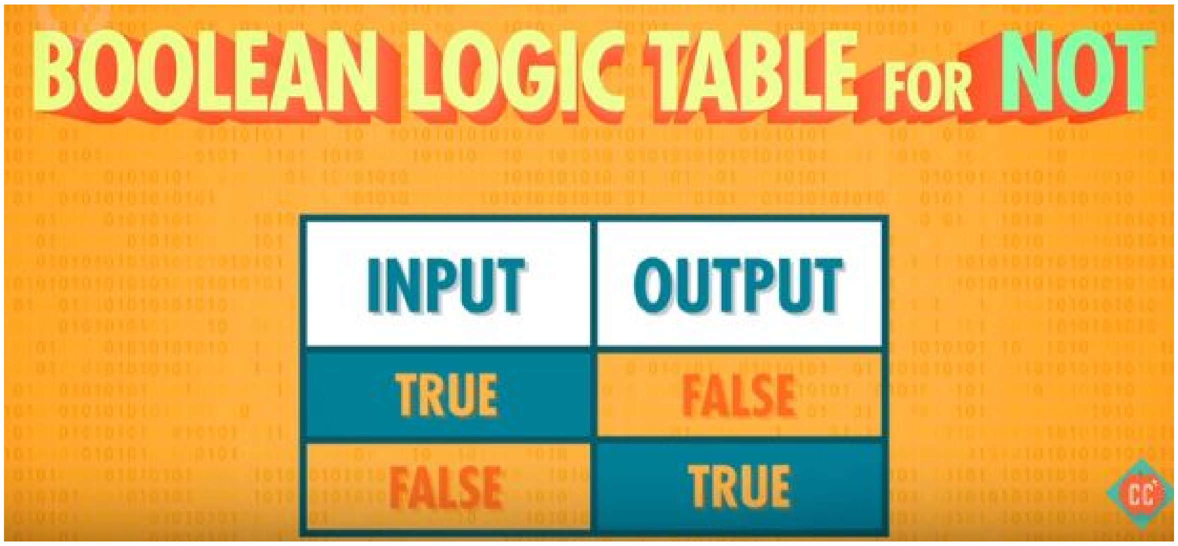
The controlled circuit can then connect other circuits, or to something like a motor, which might increment a count on a gear, like in Hollerith's tabulating machine we talked about last episode. Unfortunately, the mechanical arm inside of a relay *has mass*, and therefore can't move instantly between opened and closed stated. A good relay in the 1940's might be able to flick back and forth fifty times in a second. That might seem pretty fast, but it's not fast enough to be useful at solving large, complex problems. The Harvard Mark I could do 3 additions or subtractions per second; multiplications took 6 seconds, and divisions took 15. And more complex operations, like trigonometric function, could take over a minute. In addition to slow switching speed, another limitation was wear and tear. Anything mechanical that moves will wear over time, some things break entirely and other things start getting sticky, slow, and just plain unreliable. And as the number of relays increases, the probability of a failure increases too. The Harvard Mark I had roughly 3500 relays. Even if you assume a relay has an operational life of 10 years, this would mean you'd have to replace, on average, one faulty relay every day! That's a big problem when you are in the middle of running some important, multi-day calculation. And that's not all engineers had to contend with these huge, dark, and warm machines also attracted insects. In September 1947, operators on Harvard Mark II pulled a dead moth from a malfunctioning repay. Grace Hopper who we will look at in the next chapter noted, “from then on, when anything went wrong with a computer, we said it had bugs in it.” And that’s where we get the term computer bug. It was clear that a faster, more reliable alternative to electro-mechanical relays was needed if computing was going to advance further, and fortunately that alternative already existed! In 1904, English physicist John Ambrose Fleming developed a new electrical component called thermionic valve, which housed two electrodes inside an airtight glass bulb – this way the first vacuum tube. One of the electrodes could be heated, which would cause it to emit electrons – a process called thermionic emission. The other electrode could then attract these electrons to create the flow of our electric faucet, but only if it was positively charged – if it had a negative or neutral charge, the electrons would no longer be attracted across the vacuum so no current would flow. An electronic component that permits the one-way flow of current is called a diode, but what was really needed was a switch to help turn this flow on and off. Luckily, shortly after, in 1906, American inventor Lee de Forest added a third “control wire” electrode that sits between the two electrodes in Fleming’s design. By applying a positive charge to the control electrode, it would permit the flow of electrons as before. But if the control electrode was given a negative, it would prevent the flow to electrons. So by manipulating the control wire, one could open or close the circuit. It is pretty much the same thing as a relay – but importantly, they could switch thousands of times per second. These triode vacuum tubes would become the basics of radio, long distance telephone, and many other electronic devices for nearly a half century. I should note here that vacuum tubes were not perfect – they are kind of fragile, and can burn out like light bulbs, they were a big improvement over mechanical relays. Also, initially vacuum tubes were expensive – a radio set often used one, but a computer might require hundreds or thousands of electrical switches. But by the 1940s, their cost and reliability had improved to the point where they became feasible for use in computer… at least by people with deep pockets, like governments. This marked the shift from electro-mechanical computing to electronic computing. Let’s go to the Thought Bubble. The first large-scale use of vacuum tubes for computing was the Colossus MK 1 designed by engineer Tommy Flowers and completed in December of 1943. The Colossus was installed at Bletchley Park, in the UK, and helped to decrypt Nazi communications. This may sound familiar because two years prior Alan

Turing, often called the father of computer science, had created an electromechanical device, also at Bletchley Park, called the Bombe. It was electromechanical machine designed to break Nazi Enigma codes, but the Bombe was not technically a computer. Anyway, the first version of Colossus contained 1600 vacuum tubes, and in total, ten Colossi were built to help with code-breaking. Colossus is regarded as the first programmable, electronic computer. Programming was done by plugging hundreds of wires into plugboards, sort of like old school telephone switchboards, in order to set up the computer to perform the right operations. So while “Programmable”, it still had to be configured to perform a specific computation. Enter The Electronic Integrator and Calculator – or EMIAC – completed a few years later in 1946 at University of Pennsylvania. Designed by Mauchly and J. Presper Eckert, this was the world’s first truly general purpose, programmable, electronic computer. ENIAC could perform 5000 ten-digit additions or subtractions per second, many, many times faster then any machine that came before it. It was operational for ten years, and it estimated to have done more arithmetic than the entire human race up to that point. But with that many vacuum tubes failures were common, and ENIAC was generally only operational for about half a day at a time before breaking down. Thanks Thought Bubble. By the 1950’s, even vacuum-tube-based computing was reaching its limits. The US Air Force’s AN/FSQ-7 computer, which was completed in 1955, was part of the “SAGE” air defense computer system. To reduce cost and size, as well as improve reliability and speed, a radical new electronic switch would be needed. In 1947, Bell Laboratory scientists John Bardeen, Walter Brattain, and William Shockley invented the transistor, and with it, a whole new era of computing was born! The physics behind transistors is pretty complex, relying on quantum mechanics, so we are going to stick to the basics. A transistor is just like relay or vacuum tube – it’s a switch that can be opened or closed by applying electrical power via a control wire. Typically, transistors have two electrodes separated by a material that sometimes can conduct electricity, and other times resist it – a semiconductor. In this case, the control wire attaches to a “gate” electrode. By charging the electrical charge of the gate, the conductivity of the semiconducting material can be manipulated, allowing current to flow or be stopped – like the water faucet analogy we discussed earlier. Even the very first transistor at Bell Labs showed tremendous promise – it could switch between on and off states 10,000 times per second. Further, unlike vacuum tubes made of glass and with carefully suspended, fragile components, transistors were solid material known as solid state component. Almost immediately , transistors could be made smaller then the smaller possible relays or vacuum tubes. This led to dramatically smaller and cheaper computers, like the IBM 608, released in 1957 – the first fully transistor-powered, commercially-available computer. It contained 3000 transistors and could perform 4,500 additions, or roughly 80 multiplications or divisions, every second. IBM soon transitioned all of its computing products to transistors, bringing transistor-based computers into offices, and eventually, homes. Today, computers use transistors that are smaller than 50 nanometers in size – for reference, a sheet of paper is roughly 100,000 nanometers thick. And they are not only incredibly small, they're super fast – they can switch stated millions of times per second, and can run for decades. A lot of this transistor and semiconductor development happened in Santa Clara Valley, between San Francisco and San Jose, California. As the most common material used to create semiconductors is silicon, this region soon became known as Silicon Valley. Even William Shockley moved there, founding Shockley Semiconductor, whose employees later founded Intel – the world's largest computer chip maker today. Ok, so we have gone from relays to vacuum tubes to transistors. We can turn electricity on and off really, really, really, fast. But how do we get from transistors to actually computing something, especially if we do not have motors and gears? That's what you will learn in the next few chapters.

3.Boolean Logic & Logic Gates

In this chapter we start our journey up the ladder of abstraction, where we leave behind the simplicity of being able to see every switch and gear, but gain the abillity to assemble increasingly complex systems.

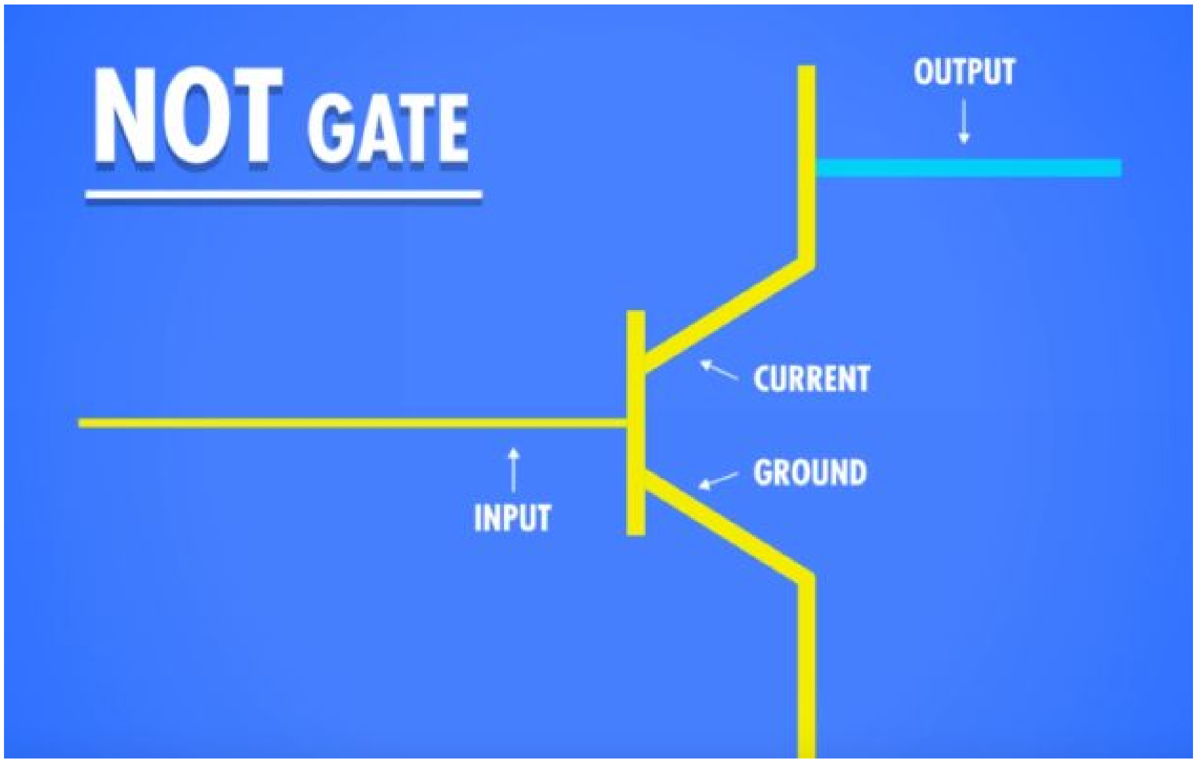
Last chapter was about how computers evolved from electromechanical devices, that often had decimal representations of numbers – like those represented by teeth on a gear – to electronic computers with transistors that can turn the flow of electricity on or off. And fortunately, even with just two states of electricity, we can represent important information. We call this representation Binary – which literally means “of two states”, in the same way a bicycle has two wheels or a biped has two legs. You might think two states is not a log to work with, and you’d be right! But it is exactly you need for representing the values “true” and “false”. In computer, an “on” state, when electricity is flowing, represents true. The off state, no electricity flowing, represents false. We can also write binary as 1’s and 0’s instead of trues and falsie’s – they are just different expressions of the same signal – but we’ll talk more about that in the next episode. Now it is actually possible to use transistors for more than just turning electrical current on and off, and to allow for different levels of current. Some early electronic computers were ternary, that’s three states, and even quandary, using 5 states. The problem is, the more intermediate states there are, the harder it is to keep them all separate – if your smartphone battary starts running low or there’s electrical noise because someone’s running a microwave nearby, the signals can get mixed up… and this problem only gets worse with transistors changing states millions of times per second! So, placing two signals as far apart as possible – using just ‘on and off’ – gives us the most distinct signal to minimize these issues. Another reason computers use binary is that an entire branch of mathematics already existed that dealt exclusively with true and false values. And it had figured out all of the necessary rules and operations for manipulating them. It’s called Boolean Algebra! George Boole, from which Boolean Algebra later got its name, was a self-taught English mathematician in the 1800s. He was interested in representing logical statements that went “0under, over, and beyond” Aristotle’s approach to logic, which was, unsurprisingly, grounded in philosophy. Boole’s approach allowed truth to be systematically and formally proven, through logic equations which he introduced in his first book, “The Mathematical Analysis of Logic” in 1847. In “regular” algebra – the type you probably learned in high school – the values of variables are numbers, and operations on those numbers are things like addition and multiplication. But in Boolean Algebra, the values of variables are true and false, and the operations are logical. There are three fundamental operations in Boolean Algebra: a NOT, an AND, and an OR operation. And these operations turn out to be really useful so we’re going to look at them individually. A NOT takes a single Boolean value, either true or false, and negates it. It flips true to false, and false to true. We can write out a little logic table that shows the original value under Input, and the outcome after applying the operation under Output.



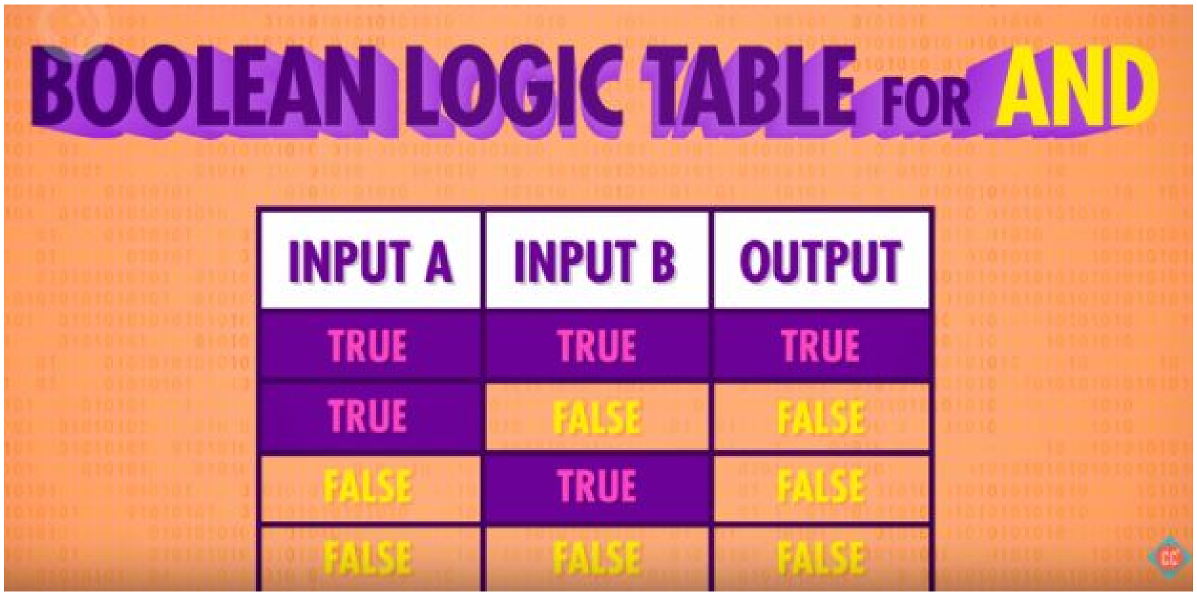
BOOLEAN LOGIC TABLE FOR NOT

INPUT	OUTPUT
TRUE	FALSE
FALSE	TRUE

Now here’s the cool part – we can easily build Boolean logic out of transistors. As we discussed last chapter, transistors are just little electrically controlled switches. They have three wires: two electrodes and one control wire. When you apply electricity to the control wire, it lets current flow through from one electrode, through the transistor, to the other electrode. This is a lot like a spigot on a pipe – open the tap, water flows, close the tap, water shuts off. You can think of the control wire as an input, and the wire coming from the bottom electrode as the output. So with a single transistor, we have one input and one output. If we turn the input on, the output is also on because the current can flow through it. If we turn the input off, the output is also off and the current can no longer pass through. Or in Boolean terms, when the input is true, the output is true. And when the input is false, the output is also false. This is not a very exciting circuit though because it is not doing anything – the input and output are the same. But, we can modify this circuit just a little bit to create a NOT. Instead of having the output wire at the end of the transistor, we can move it before. If we turn the input on, the transistor allows current to pass through it to the "ground", and the output wire will not receive that current – so it will be off.



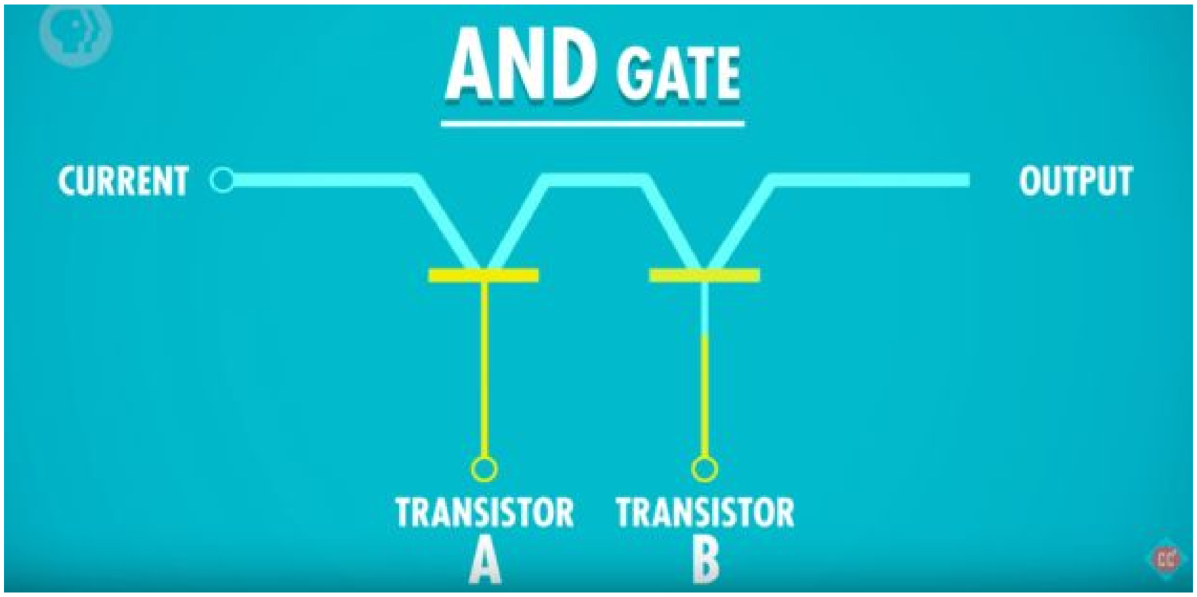
In our water metaphor grounding would be like if all the water in your house was flowing out of a huge hose so there wasn't any water pressure left for your shower. So in this case if the input is on, output is off. When we turn off the transistor, though, current is prevented from flowing down it to the ground, so instead, current flows through the output wire. So the input will be off and the output will be on. And this matches our logic table for NOT so congrats, we just built a circuit that computes NOT! We call them NOT gates – we call them gates because they're controlling the path of our current. The AND Boolean operation takes two inputs, but still has a single output. In this case the output is only true if both inputs are true. Here is Boolean Logic Table for AND.



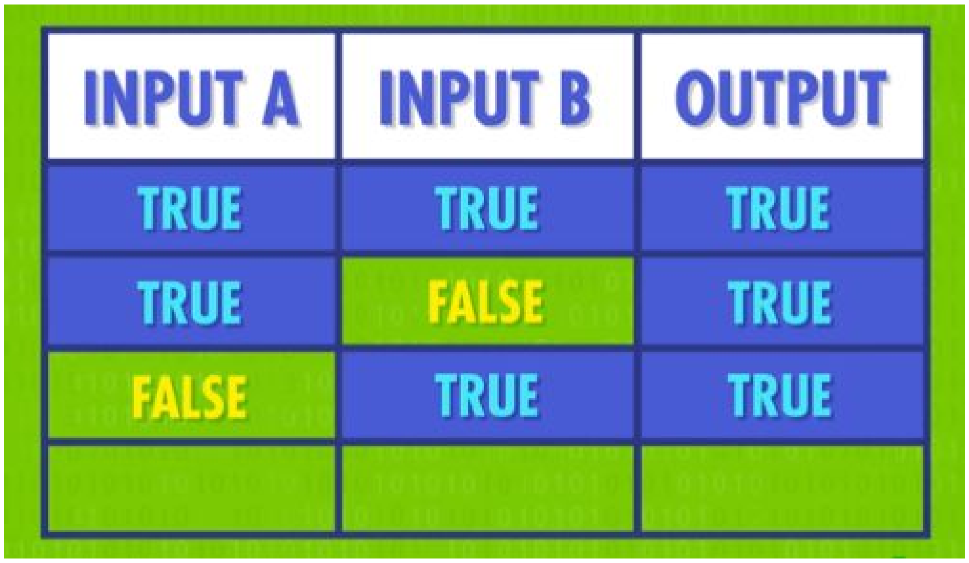
BOOLEAN LOGIC TABLE FOR AND

INPUT A	INPUT B	OUTPUT
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

To build an AND gate, we need two transistors connected together so we have our two inputs and one output. If we turn on just transistor A, current will not flow because the current is stopped by transistor B. Alternatively, if transistor B is on, but the transistor A is off, the same thing, the current can’t get through. Only if transistor A AND transistor B are on does the output wire have current.

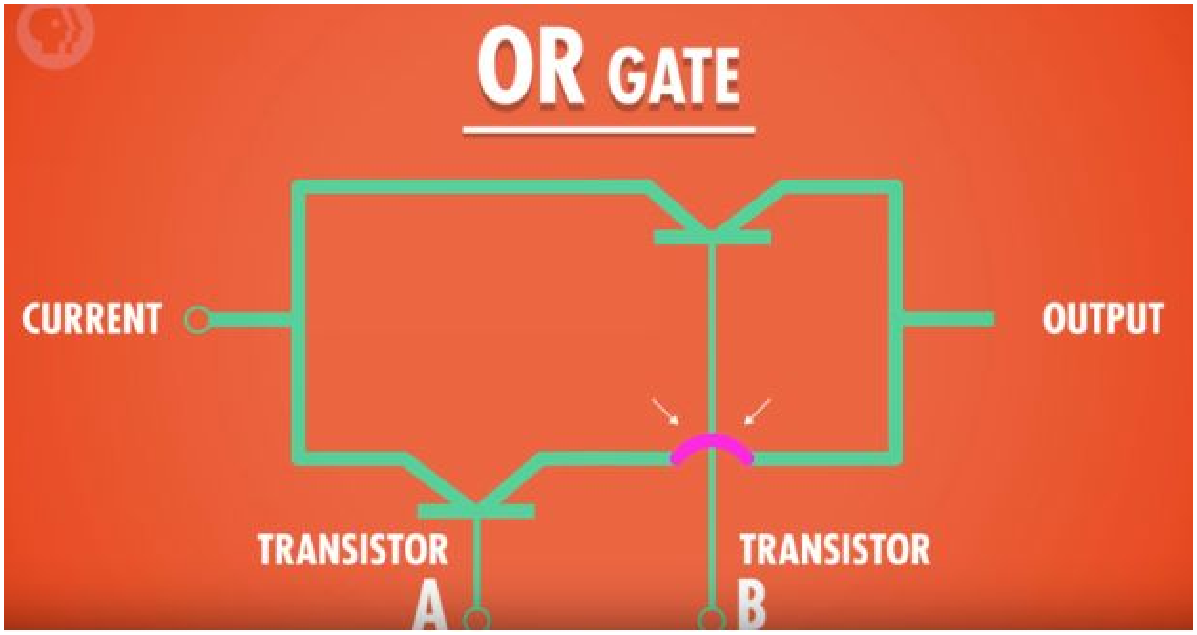


The last Boolean operation is OR – where only one input has to be true for the output to be true. An OR statement is also true if both facts are true. The only time an OR statement is false is if both inputs are false.

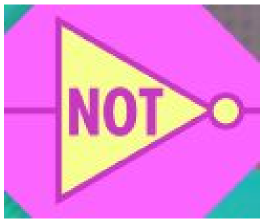


INPUT A	INPUT B	OUTPUT
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE

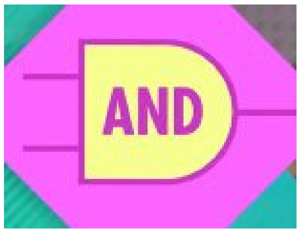
Building an OR gate from transistors needs a few extra wires. Instead of having two transistors in series – one after the other – we have them in parallel. We run wires from the current source to both transistors. We use little arc to note that the wires jump over one another and aren't connected, even though they look like they cross.



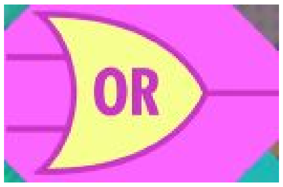
If both transistors are turned off, the current is prevented from flowing to the output, so the output is also off. Now, if we turn on just Transistor A, current can flow to the output. Same thing if transistor A is off, but Transistor B is on. Basically if A OR B is on, the output is also on. Also, if both transistors are on, the output is still on. Ok, now that we've got NOT, AND, and OR gates, and we can leave behind the constituent transistors and move up a layer of abstraction. The standard engineers use for these gates are a triangle with a dot for a NOT



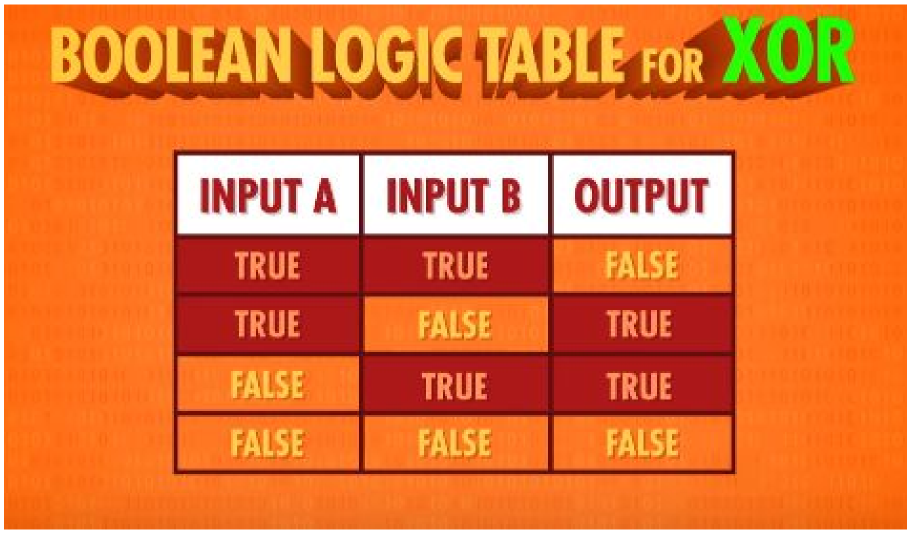
a D for AND



and a spaceship for the OR.



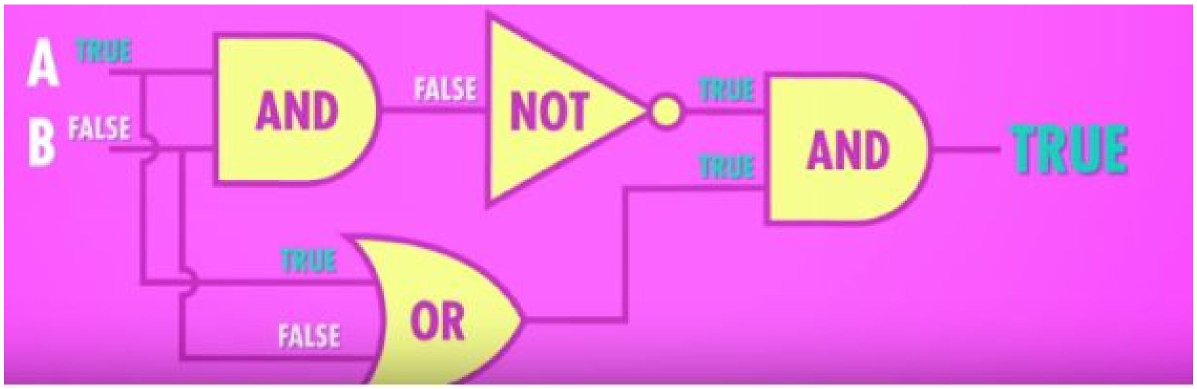
Representing them and thinking about them this way allows us to build even bigger components while keeping the overall complexity relatively the same – just remember that that mess of transistors and wires is still there. For example, another useful Boolean operation in computation is called an exclusive OR – or XOR for short.



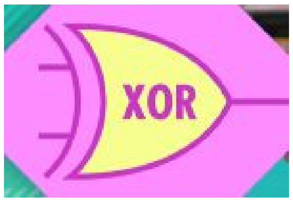
BOOLEAN LOGIC TABLE FOR XOR

INPUT A	INPUT B	OUTPUT
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

XOR is like a regular OR, but with one difference: if both inputs are true, the XOR is false. The only time an XOR is true is when one input is true and the other input is false. And building this from transistors is pretty confusing, but we can show how an XOR is created from our three basics Boolean gates. We know we have two inputs again – A and B – and one output. Let's start with an OR gate, since the logic table looks almost identical to an OR. There's only one problem – when A and B are true, the logic is different from OR, and we need to output "false". To do this we need to add some additional gates. If we add an AND gate, and the input is true and true, the output will be true. This is not what we want. But if we add a NOT immediately after this will flip it to false. Okay, now if we add a final AND gate and send it that value along with the output of our original OR gate, the AND will take in "false" and "true", and since AND needs both values to be true, its output is false. That's the first row of our logic table. If we work through the remaining input combinations, we can see this Boolean logic circuit does implement an Exclusive OR.



And XOR turns out to be a very useful component, and we'll get to it in another chapter, so useful in fact engineers gave it its own symbol too – and OR gate with a smile :)

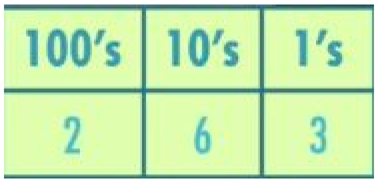


But most importantly, we can now put XOR into our metaphorical toolbox and not have to worry about the individual logic gates that make it up, or the transistors that make up those gates, or how electrons are flowing through a semiconductor. Moving up another layer of abstraction. When computer engineers are designing processors, they rarely work at the transistor level, and instead work with much larger blocks, like logic gates, and even larger components made up of logic gates, which we will look at in future chapters. And even if you are a professional computer programmer, it is not often that you think about how that you are programming is actually implemented in the physical world by these teeny tiny components. We have also moved from thinking about raw electrical signals to our first representation of data – true and false – and we have even gotten a little taste of computation. With just the logic gates in this chapter, we could build a machine that evaluates complex logic statements, like if "Name is John Green AND after 5pm OR is Weekend AND near Pizza Hut", then "John will want pizza" equals true. And with that, I'm starving.

4.Representing Numbers and Letters with Binary

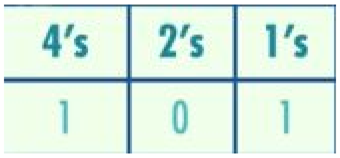
In this chapter we are going to look at how computers store and represent numerical data. This means we have got to look at Math! But don't worry. Every single one of you already knows exactly what you need to know to follow along. So, last chapter was about how transistors can be used to build logic gates, which can evaluate Boolean statements. And in Boolean algebra, there are only two binary values: true and false. But if we only have two values, how in the world do we represent information beyond just these two values? That's where the Math comes in.

So, as we mentioned last chapter, a single binary value can be used to represent a number. Instead of true and false, we can call these two states 1 and 0 which is actually incredibly useful. And if we want to represent larger things we just need to add more binary digits. This works exactly the same way as the decimal numbers that we are all familiar with. With decimal numbers there are "only" 10 possible values a single digit can be; 0 through 9, and to get numbers larger than 9 we just start adding more digits to the front. We can do the same with binary. For example, let's take the number two hundred and sixty three. What does this number actually represent? Well it means we've got 2 one-hundreds, 6 tens, and 3 ones. If you add those all together, we've got 263.



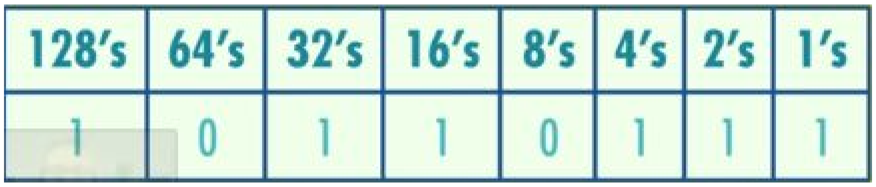
100's	10's	1's
2	6	3

Each multiplier is ten times larger than the one to the right. That's because each column has ten possible digits to work with, 0 through 9, after which you have to carry one to the next column. For this reason, it's called base-ten notation, also called decimal since deci means ten. AND Binary works exactly the same way, it's just base-two. That's because there are only two possible digits in binary – 1 and 0. This means that each multiplier has to be two times larger than the column to its right. Instead of hundreds, tens, and ones, we now have fours, twos and ones. Take for example the binary number: 101. This means we have 1 four, 0 twos, and 1 one.



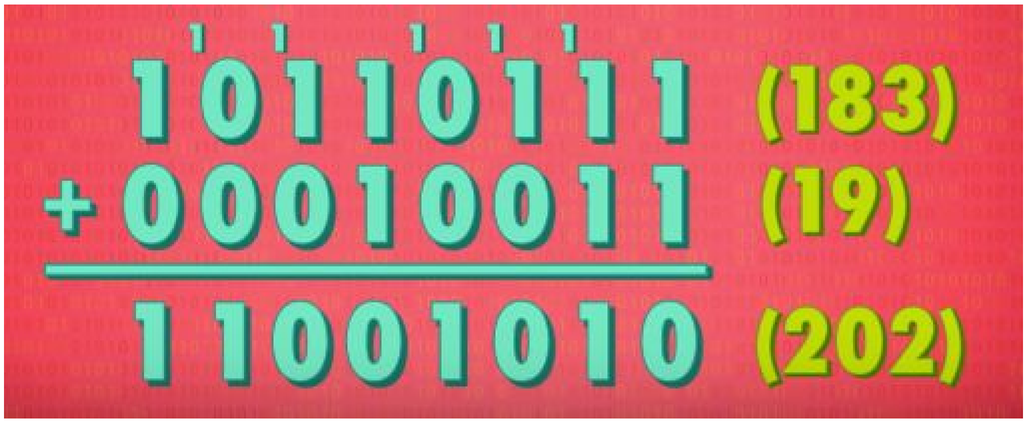
4's	2's	1's
1	0	1

Add those all together and we've got the number 5 in base ten. But to represent larger numbers, binary needs a lot more digits. Take this number in binary 10110111. We can convert it to decimal in the same way. We have 1 x 128, 0 x 64, 1 x 32, 1 x 16, 0 x 8, 1 x 4, 1 x 2, and 1 x 1.



128's	64's	32's	16's	8's	4's	2's	1's
1	0	1	1	0	1	1	1

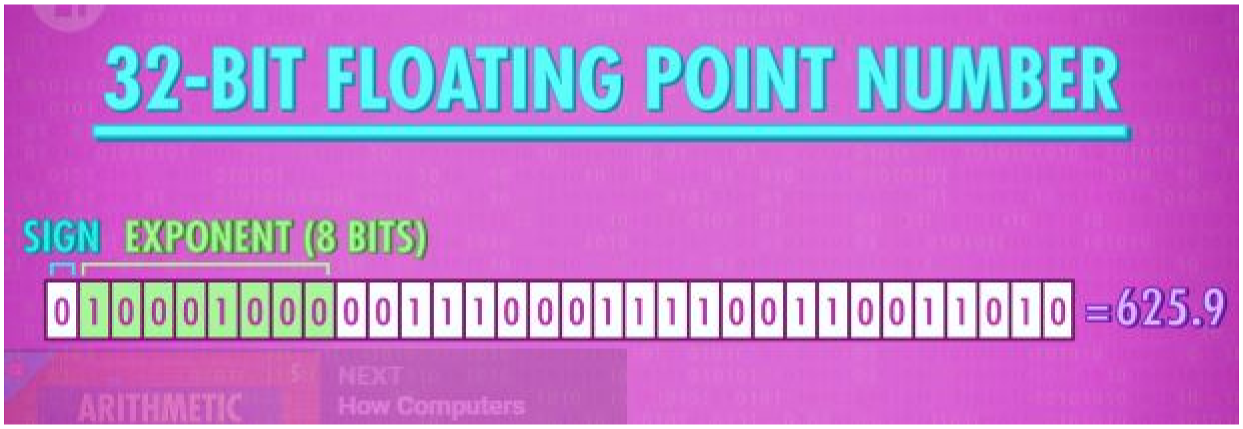
Which all adds up to 183 (128+32+16+4+2+1=183). Math with binary numbers is not hard either. Take for example decimal addition of 183 plus 19. First we add 3 + 9, that's 12, so we put 2 as the sum and carry 1 to the ten's column. Now we add 8 plus 1 plus the 1 we carried that's 10, so the sum is 0 carry 1. Finally, we add 1 plus the 1 we carried, which equals 2. So the total sum is 202. Here's the same sum but in binary. Just as before, we start with the ones column. Adding 1+1 results in 2, even in binary. But, there is no symbol "2" so we use 10 and put 0 as our sum and carry the 1. Just like in our decimal example. 1 plus the 1 carried, equals 3 or 11 in binary, so we put the sum as 1 and we carry 1 again, and so on. We end up with 11001010, which is the same as the number 202 in base ten.



```
  1 1   1 1 1
  10110111  (183)
+ 00010011  (19)
----------
  11001010  (202)
```

Each of these binary digits, 1 or 0, is called a “bit”. So in these few examples, we were using 8-bit numbers with their lowest value of zero and highest value is 255, which requires all 8 bits to be set to 1. That’s 256 different values, or 2 to the 8th power. You might have heard of 8-bit computers, or 8-bit graphics or audio. These were computers that did most of their operations in chunks of 8 bits. But 256 different values isn’t a lot to work with, so it meant things like 8-bit games were limited to 256 different colors for their graphics. And 8-bit is such a common size in computing, it has a special word: a byte. A byte is 8 bits. If you’ve got 10 bytes, it means you’ve really got 80 bits. You’ve heard of kilobytes, megabytes, gigabytes and so on. These prefixes denote different scales of data. Just like one kilogram is a thousand grams, 1 kilobyte is a thousand bytes… or really 8000 bits. Mega is a million bytes (MB), and giga is a billion bytes (GB). Today you might even have heard a hard drive that has 1 terabyte (TB) of storage. That’s 8 trillion ones and zeros. But hold on! That’s not always true. In binary, a kilobyte has two to the power of 10 bytes, or 1024. 1000 is also right when talking about kilobytes but should acknowledge it is not the only the correct definition. You’ve probably also heard the term 32-bit or 64-bit computers – you’re almost certainly using right now. What this means it that they operate in chunks or 32 or 64 bits. That’s a lot of bits! The largest number you can represent with 32 bits is just under 4.3 billion. Which is thirty-two 1’s in binary. This is why our Instagram photos are so smooth and pretty – they are composed of millions of colors, because computers today use 32-bit color graphics of course, not everything is a positive number. So we need a way to represent positive and negative numbers. Most computers use the first bit for the sign: 1 for negative, 0 for positive numbers, and then use the remaining 31 bits for the number itself. That gives us a range of roughly plus or minus two billion. While this is pretty big range of numbers, it’s not enough for many tasks. There are 7 billion people on the earth, and the US national debt is almost trillion dollars after all. This is why 64-bit numbers are useful. The largest value a 64-bit number can represent is around 9.2 quintillion! That’s a lot of possible numbers and will hopefully stay above the US national debt for a while! Most importantly, in the next chapter, computers must label locations in their memory, known as addresses, in order to store and retrieve values. As computer memory has grown to gigabytes and terabytes – that’s trillions of bytes – it was necessary to have

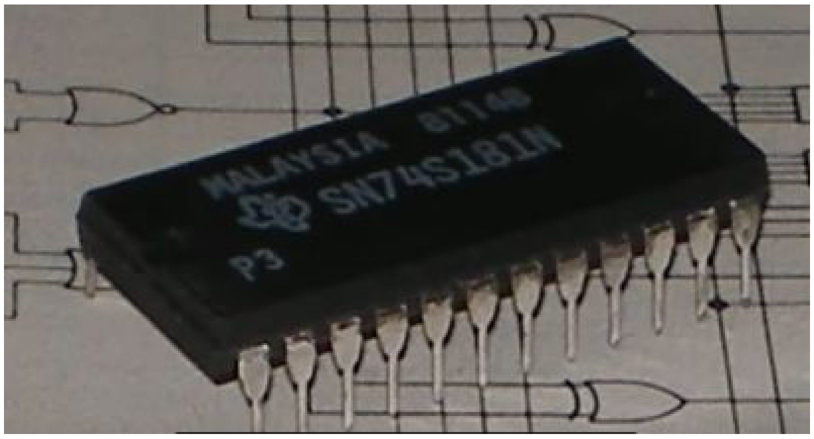
64-bit memory addresses as well. In addition to negative and positive numbers, computers must deal with numbers that are not whole numbers, like 12.7 and 3.14. These are called “floating point” numbers, because the decimal point can float around in the middle of numbers. Several methods have been developed to represent floating point numbers. The most common of which is the IEEE 754 standard. And you thought historians were the only people bad at naming things! In essence, this standard stores decimal values sort of like scientific notation. For example, 625.9 can be written as 0.6259 x 10^3. There are two important numbers here: the .6259 is called the significand. And 3 is the exponent. In a 32-bit floating point number, the first is used for the sign of the number – positive or negative. The next 8-bits are to store the exponent and the remaining 23 bits are used to store the significand.



Ok, this is just a lot about numbers, but your name is probably composed of letters, so it’s really useful for computers to also have a way to represent text. However, rather than have a special form of storage for letters, computers simply use numbers to represent letters. The most straightforward approach might be to simply number the letters of the alphabet: A being 1, B being 2, C 3, and so on. In fact, Francis Bacon, the famous English writer, used five-bit sequences to encode all 26 letters of the English alphabet to send secret messages back in the 1600s. And five bits can store 32 possible values – so that’s enough for the 26 letters, but not enough for punctuation, digits, and upper and lower case letters. Enter ASCII, the American Standard Code for Information Interchange. Invented in 1963, ASCII was a 7-bit code, enough to store 128 different values. With this expanded range, it could encode capital letters, lowercase letters, digits 0 through 9, and symbols like the @ sign and punctuation marks. For example, a lowercase ‘a’ is represented by the number 97, while a capital ‘A’ is 65. A colon is 58 and a closed parenthesis is 41. ASCII even had a selection of special command codes, such as a newline character to tell the computer where to wrap a line to the next row. In older computer systems, the line of text would literally continue off the edge of the screen if you didn’t include a new line character! Because ASCII was such an early standard, it became widely used, and critically, allowed different computers built by different companies to exchange data. This ability to universally exchange information is called “interoperability”. However, it did have major Limitation: it was really only designed for English. Fortunately, there are 8 bits in a byte, not 7, and it soon became popular to use codes 128 though 255, previously unused, for “national” characters. In the US, those extra numbers were largely used to encode additional symbols, like mathematical notation, graphical elements, and common accented characters. On the other hand, while the Latin characters were used universally, Russian computers used the extra codes to encode Cyrillic characters, and Greek computers, Greek letters, and so on. And national character codes worked pretty well for most countries. The problem was, if you opened an email written in Latvian on Turkish computer, the result was completely incomprehensible. And things totally broke with the rise of computing in Asia, as languages like Chinese and Japanese have thousands of characters. There was no way to encode all those characters in 8-bits! In response, each invented multi-byte encoding schemes, all of which were mutually incompatible. The Japanese were so familiar with this encoding problem that they had a special name for it: “mojibake”, which means “scrambled text”. And so it was born – Unicode – one format to rule them all. Devised in 1992 to finally do away with all of the different international schemes it replaced them with one universal encoding scheme. The most common version of Unicode uses 16 bits with space for over a million codes – enough for every single character from every language ever used – more than 120,000 of them in over 100 types of script plus space for mathematical symbols and even graphical characters like Emoji. And in the same way that ASCII defines a scheme for encoding letters as binary numbers, other file formats – like MP3 or GIFs – use binary numbers to encode sounds or colors of pixel in our photos, movies, and music. Most importantly, under the hood it all comes down to long sequences of bits. Text messages, YouTube videos, every webpage on the internet, and even your computer’s operating system, are nothing but long sequences of 1s and 0s. Next chapter, is about how your computer starts manipulating those binary sequences, for our first true taste of computation.

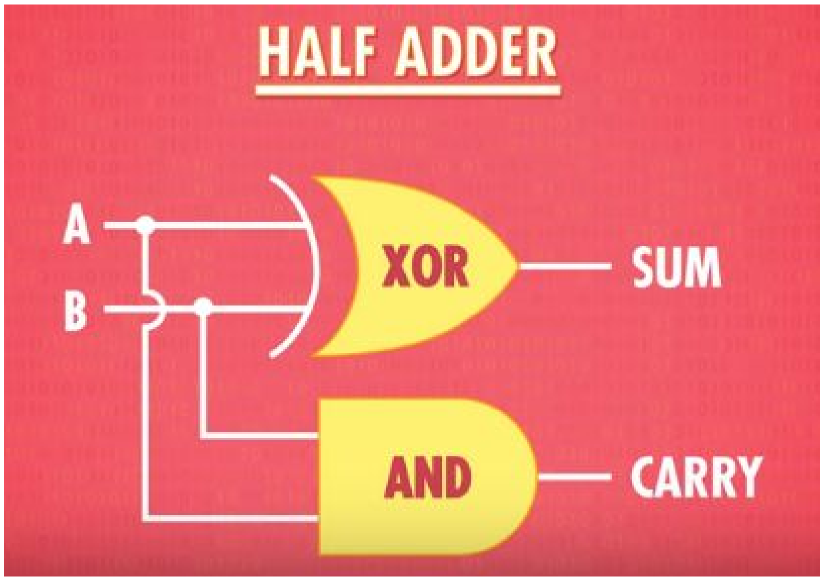
5.How Computers Calculate – the ALU

So in the last chapter, was about how numbers can be represented in binary. Representing like 00101010 is 42 in decimal. Representing and storing numbers is an important function of a computer, but the real goal is computation, or manipulating numbers in a structured and purposeful way, like adding two numbers together. These operations are handled by computer's Arithmetic and Logic Unit, but most people call it by its street name: the ALU. The ALU is the mathematical brain of a computer. When you understand an ALU's design and function, you'll understand a fundamental part of modern computers. It is THE thing that does all of the computation in a computer, so basically everything uses it. First though, look at this beauty.

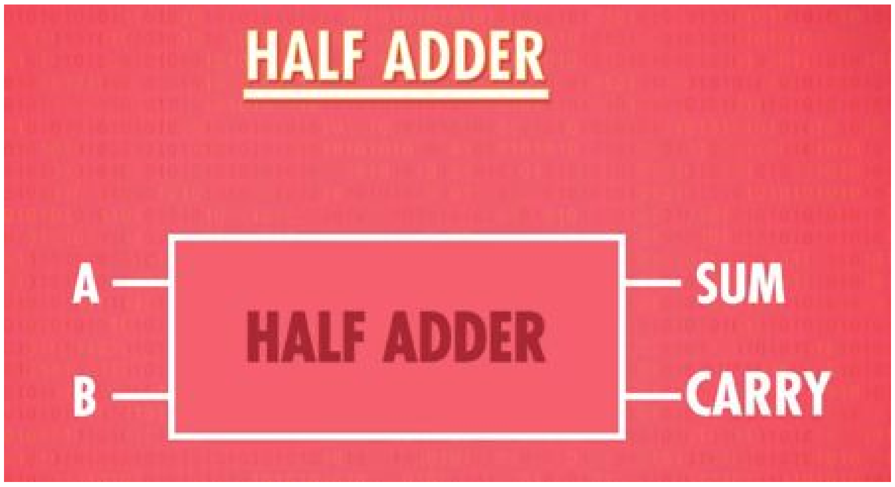


This is perhaps the most famous ALU ever, the Intel 74181. When it was released in 1970, it was the first complete ALU that fit entirely inside of a single chip – which was a huge engineering feat at the time. So in this chapter we're going to take those Boolean logic gates we learned about last chapter to build a simple ALU circuit with much of the same functionality as the 74181. And over the next few chapters we'll use this to construct a computer from scratch. So it's going to get a little bit complicated, but I think you can handle it.

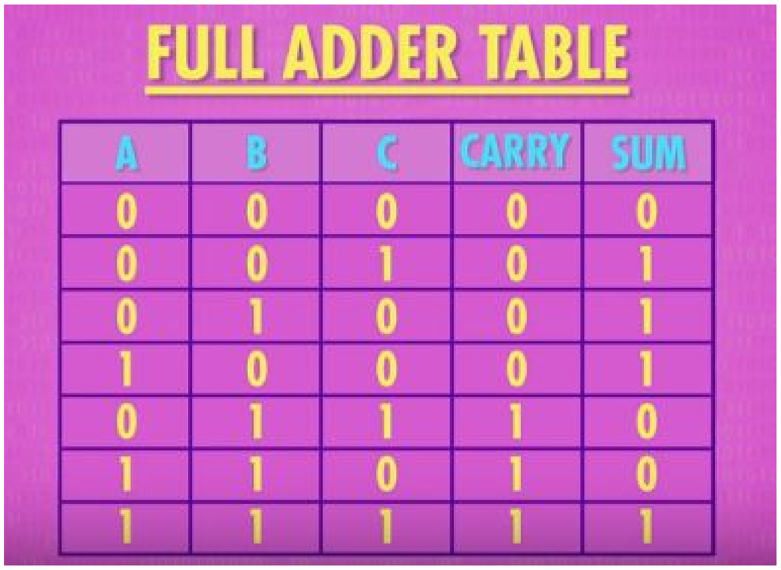
An ALU is really two units in one – there's an arithmetic unit and a logic unit. Let's start with the arithmetic unit, which is responsible for handling all numerical operations in a computer, like addition and subtraction. It also deos a bunch of other simple things like add one to a number, which is called an increment operation, but we'll look at those later. In this chapter, we're going to focus on the adding two numbers together. We could build this circuit entirely out of individual transistors, but that would get confusing really fast. So instead we can use high-level of abstraction and build our components out of logic gates, in this case: AND, OR, NOT and XOR gates. The simplest adding circuit that is possible to build takes two binary digits, and adds them together. So we have two inputs, A and B, and one output, which is the sum of those two digits. Just to clarify: A, B and the output are all single bits. There are only four possible input combinations. The first three are: 0+0 = 0, 1+0 = 1, 0+1 = 1. Remember that in binary, 1 is the same as true, and 0 is the same as false. So this set of input exactly matches the Boolean logic of an XOR gate, and we can use it as our 1-bit adder. But the fourth input combination, 1+1, is a special case. 1 + 1 is 2 obviously but there's no 2 digit in binary, so as we looked at in the last chapter, the result is 0 and the 1 is carried to the next column. So the sum is really 10 in binary. Now, the output of our XOR gate is partially correct – 1 plus 1, outputs 0. But, we need an extra output wire for that carry bit. The carry bit it only "true" when the inputs are 1 AND 1, because that's the only time then the result (two) is bigger than 1 bit can store… and conveniently we have a gate for that! An AND gate, which is only true when both inputs are true, so we'll add that to our circuit too. And that's it. This circuit is called half adder.



It's not that complicated – just two logic gates – but let's abstract away even this level of detail and encapsulate our newly minted half adder as its own component, with two inputs – bits A and B – and two outputs, the sum and the carry bits.

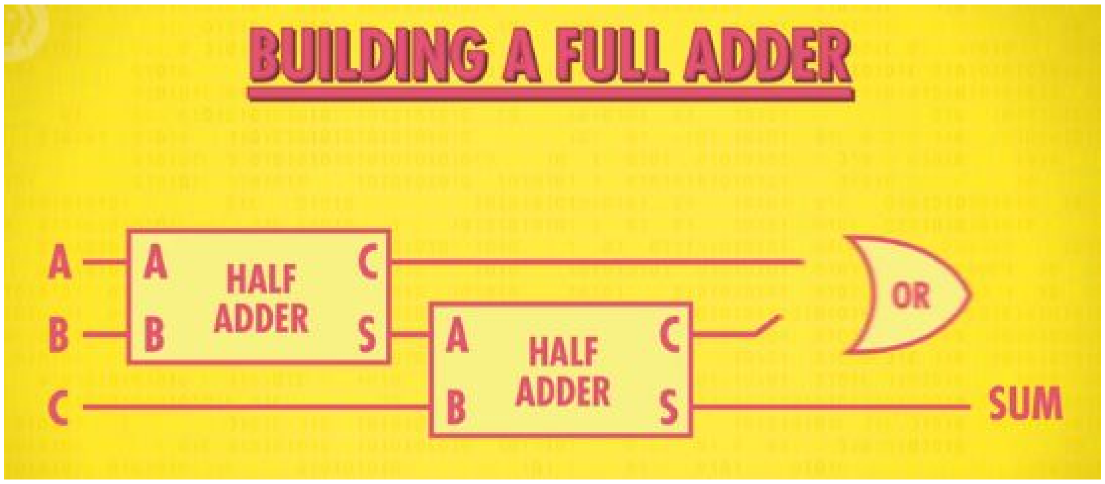


If you want to add more than 1 + 1 we're going to need a "Full Adder." That half-adder left us with a carry bit as output. That means that when we move on to the next column in a multi-column addition, and every column after that, we are going to leave to add three bits together, no two. Here's Full Adder Table.

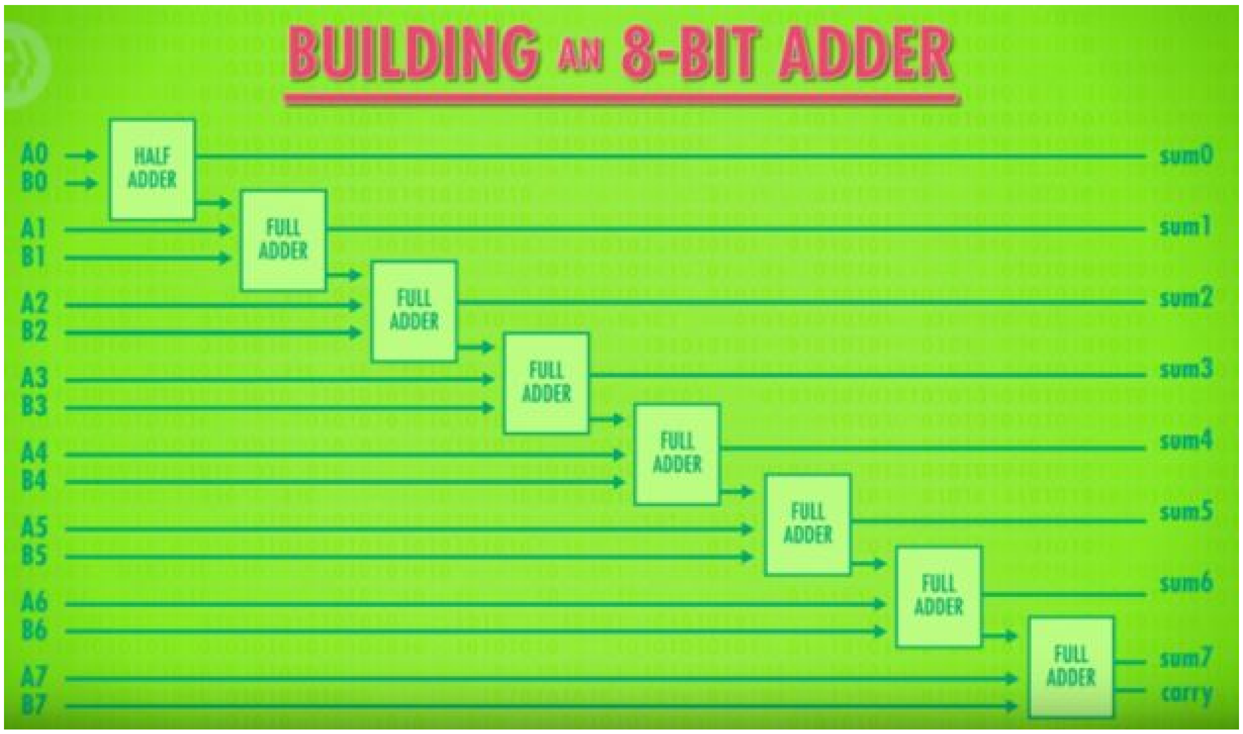

FULL ADDER TABLE

A	B	C	CARRY	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	1	0	1	0
1	1	1	1	1

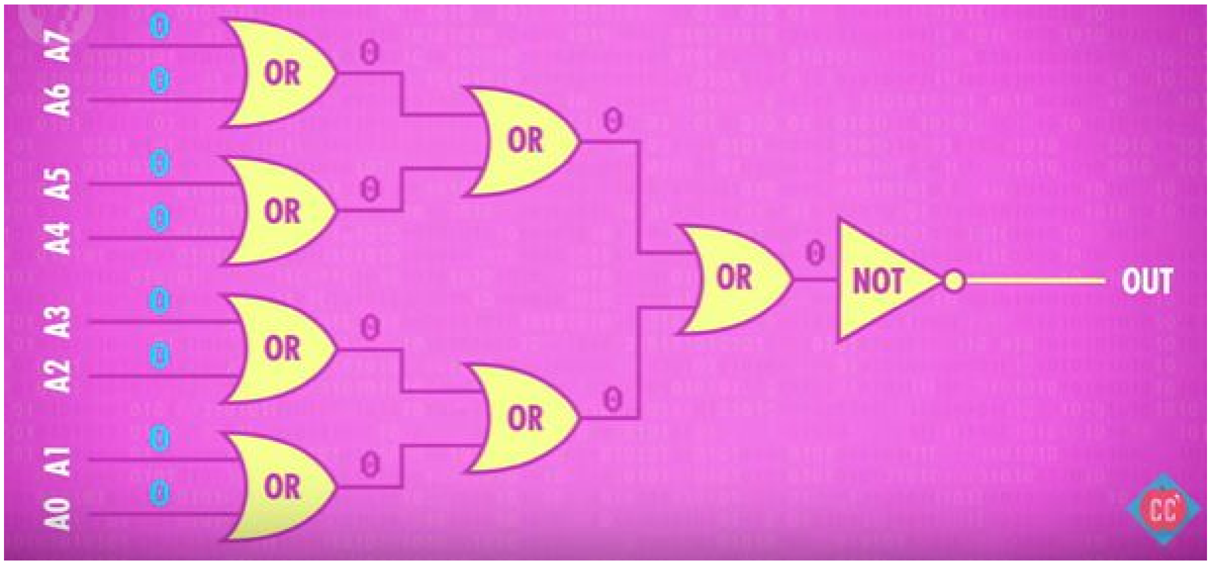
A full adder is a bit more complicated – it takes three bits as inputs: A, B and C. so the maximum possible input is 1 + 1 + 1, which equals 1 carry out 1, so we still only need two output wires: sum and carry. We can build a full adder using half adders. To do this, we use a half adder to add A plus B just like before – but then feed that result and input C into a second half adder. Lastly, we need a OR gate to check if either one of the carry bits was true.



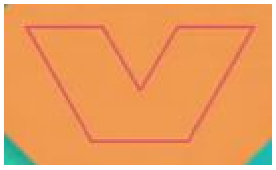
That's it, we just made a full adder! Again, we can go up a level of abstraction and wrap up this full adder as its own component. It takes three inputs, adds them, and outputs the sum and the carry, if there is one. Armed with our new components, we can now build a circuit that takes two, 8-bit numbers – let's call them A and B – and adds them together. Let's start with the very first bit of A and B, which we'll call A0 and B0. At this point, there is no carry bit to deal with, because this is our first addition. So we can use our half adder to add these two bits together. The output is sum0. Now we want to add A1 and B1 together. It's possible there was a carry from the previous addition of A0 and B0, so this time we need to use full adder that also inputs the carry bit. We output this result as sum1. Then, we take any carry from this full adder, and run it into the next full adder that handles A2 and B2. And we just keep doing this in a big chain until all 8 bits have been added.



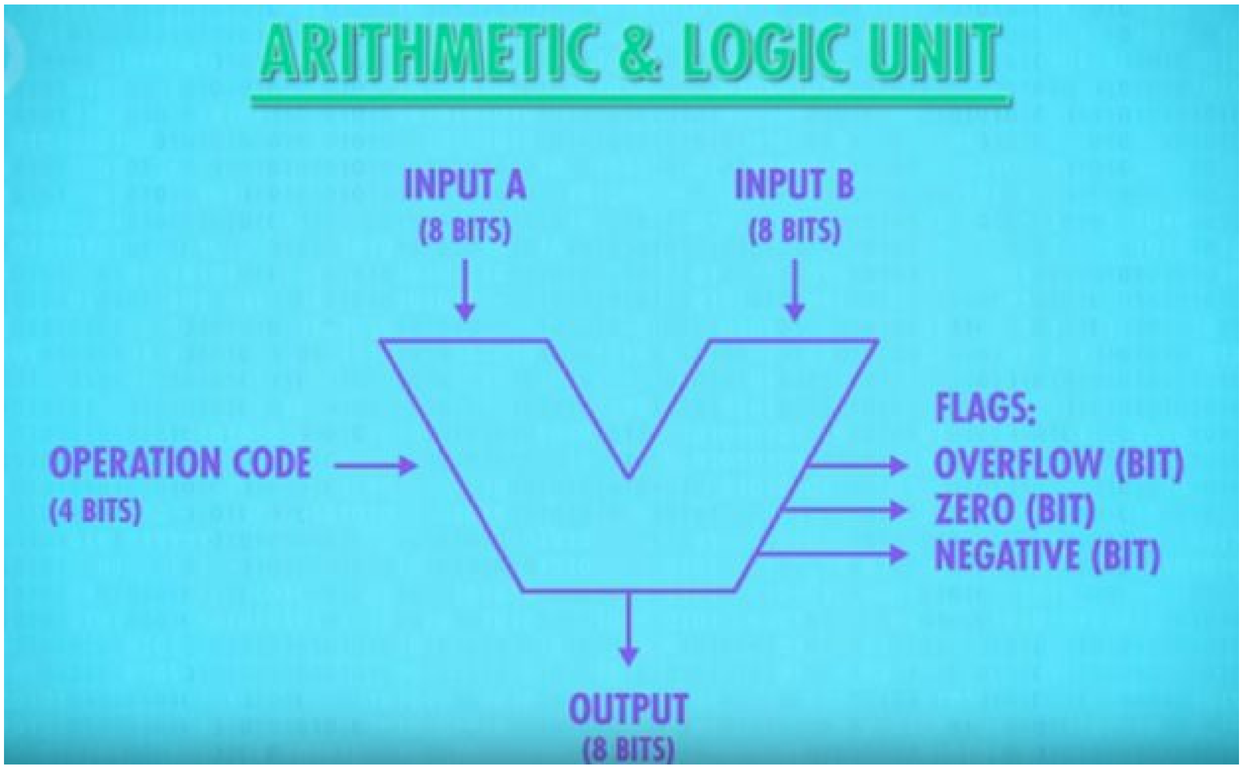
Notice how the carry bits ripple forward to each subsequent adder. For this reason this is called an 8-bit ripple carry adder. Notice how our last full adder has a carry out. If there is a carry into the 9th bit, it means the sum of the two numbers is too large to fit into 8-bits. This is called an overflow. In general, an overflow occurs when the result of an addition is too large to be represented by the number of bits you are using. This can usually cause errors and unexpected behavior. Famously, the original PacMan arcade game used 8bits to keep track of what level you were on. This meant that if you made it past level 255 – the largest number storablein 8-bits – to level 256, the ALU overflowed. This caused a bunch of errors and glitches making the level unbeatable. The bug became a rite of passage for the greatest PacMan players. So if we want to avoid overflows, we can extend our circuit with more full adders, allowing us to add 16 or 32 bit numbers. This makes overflows less likely to happen, but at the expense of more gates. An additional downside is that it takes a little bit of time for each of the carries to ripple forward. Admittedly, not very much time, electrons move pretty fast, so we're talking about billionths of a second, but that's enough to make a difference in today's fast computers. For this reason, modern computers use a slightly different adding circuit called a 'carry-look-ahead' adder which is faster, but ultimately does exactly the same thing – adds binary numbers. The ALU's arithmetic unit also has circuit for other math operations and in general these 8 operations are always supported. And like our adder, these other operations are built from individual logic gates. Interestingly, you may have noticed that there are no multiply and divide operations. That's because simple ALUs don't have circuit for this, and instead just perform a series of additions. Let's say you want to multiply 12 by 5. That's the same thing as adding 12 itself 5 times. So it would take 5 passes through the ALU to do this one multiplication. And this is how many simple processors, like those in your thermostat, TV remote, and microwave, do multiplication. It's slow, but it gets the job done. However, fancier processors, like those in your laptop or smartphone, have arithmetic units with dedicated circuits for multiplication. And as you might expect, the circuit is more complicated than addition – there's no magic, it just takes a lot more logic gates – which is why less expensive processors don't have this feature. Ok, let's move on to the other half of the ALU: the Logic Unit. Instead of arithmetic operations, the Logic Unit performs… well… logical operations, like AND, OR and NOT, which we looked at previously. It also performs simple numerical tests, like checking if a number is negative. For example, here's a circuit that tests if the output of the ALU is zero.



It does this using a bunch of OR gates to see if any of the bits are 1. Even if one single bit is 1, we know the number can't be zero and then we use a final NOT gate to flip this input so the output is 1 only if the input number is 0. So that's a high level overview of what makes up an ALU. We even built several of the main components from scratch, like our ripple adder. As you saw, it's just a big bunch of logic gates connected in clever ways. This brings us back to that ALU you admired so much at the beginning of the chapter. The Intel 74181. Unlike the 8-bit ALU we made today, the 74181 could only handle 4-bit inputs, which means YOU BUILT AN ALU THAT'S LIKE TWICE AS GOOD AS THAT SUPER FAMOUS ONE. WITH YOUR MIND! Well… sort of. We didn't build the whole thing… but you get the idea. The 74181 used about 70 logic gates, and it couldn't multiply or divide. But it was a huge step forward in miniaturization, opening the doors to more capable and less expensive computers. This 4-bit ALU circuit is already a lot to take in, but our 8-bit ALU would require hundreds of logic gates to fully build and engineers don't want to see all that complexity when using an ALU, so they came up with a special symbol to wrap it all up, which looks like a bit "V".



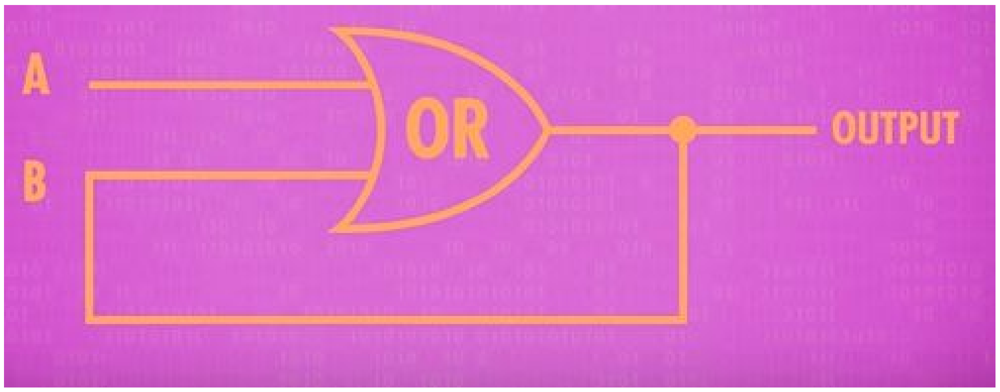
Just another level of abstraction! Our 8-bit ALU has two inputs, A and B, each with 8 bits. We also need a way to specify what operation the ALU should perform, for example, addition or subtraction. For that, we use a4-bit operation code. We'll look at that more in a later episode, but in brief, 1000 might be the command to add, while 1100 is the command for subtract. Basically, the operation code tells the ALU that operation to perform. And the result of that operation on inputs A and B is an 8-bit output. ALU also output a series of Flags, which are 1-bit output for particular states and statuses.



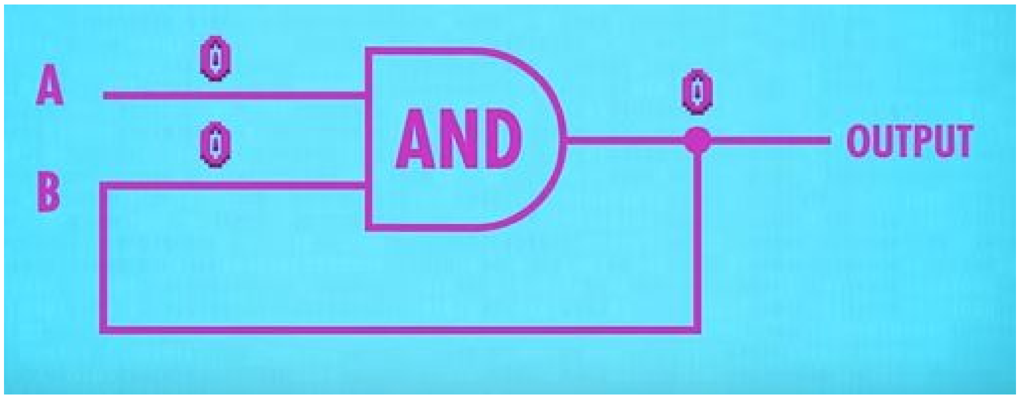
For example, if we subtract two numbers, and the result is 0, our zero-testing circuit, the one we made earlier, sets the Zero Flag to True (1). This is useful if we are trying to determine if two numbers are equal. If we wanted to test if A was less than B, we can use the ALU to calculate A subtract B and look to see if the Negative Flag was set to true. If it was, we know that A was smaller than B. And finally, there's also a wire attached to the carry out on the adder we built, so if there is an overflow, we'll know about it. This is called the Overflow Flag. Fancier ALUs will have more flags, but these three flags are universal and frequently used. In fact, we'll be using them soon in a next chapter. So now you know how computer does all its basic mathematical operations digitally with no gears or levers required. We're going to use this ALU when we construct our CPU two chapters from now. But before that, our computer is going to need some memory! We'll look at that in the next chapter.

6.Registers and RAM

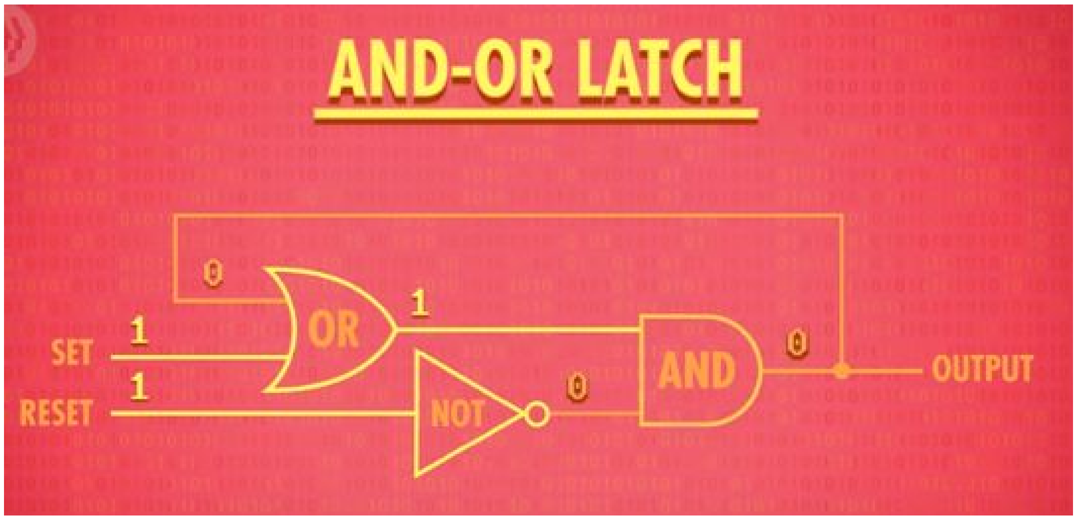
So last chapter, using just logic gates, we built a simple ALU, which performs arithmetic and logic operations, hence the 'A' and the 'L'. But of course there's not much point in calculating a result only to throw it away – it would be useful to store that value somehow, and maybe even run several operations in a row. That's where computer memory comes in! If you've ever been in the middle of a long RPG campaign on your console, or slogging through a difficult level Minesweeper on your desktop, and your dog came by, tripped and pulled the power cord out of the wall, you know the agony of losing all your progress. Condolences. But the reason for your loss is that your console, your laptop and your computers make use of Random Access Memory, or RAM, which stores things like game state – as long as the power stays on. Another type of memory, called persistent memory, can survive without power, and it's used for different things; we'll look at that type of memory in a later chapter. In this chapter, we're going to start small – literally by building a circuit that can store one single bit of information. After that, we'll scale up, and build our very own memory module, and we'll combine it with our ALU next time, when we finally build our very own CPU! All of the logic circuits we've discussed so far go in one direction – always flowing forward like our 8-bit ripple adder from last chapter. But we can also create circuits that loop back themselves. Let's try taking an ordinary OR gate, and feed the output back into one of its input and see what happens.



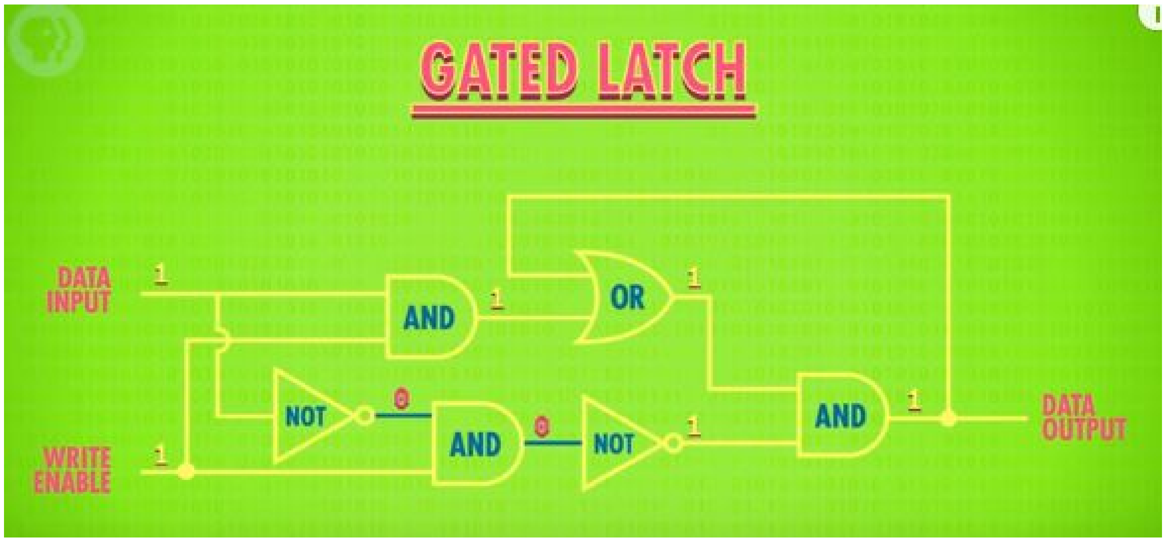
First, let's set both inputs to 0. So 0 OR 0 is 0, and so this circuit always outputs 0. If we were to flip input A to 1. 1 OR is 1, so now the output of the OR gate is 1. A fraction of a second later, that loops back around into input B, so the OR gate sees that both of its inputs are now 1. 1 OR 1 is still 1, so there is no change in output. If we flip input A back to 0, the OR gate still outputs 1. So now we've got a circuit that records "1" for us. Except, we've got a teensy tiny problem – this change is permanent! No matter how hard we try, there's no way to get this circuit to flip back from a 1 to a 0. Now let's look at this same circuit, but with an AND gate instead.



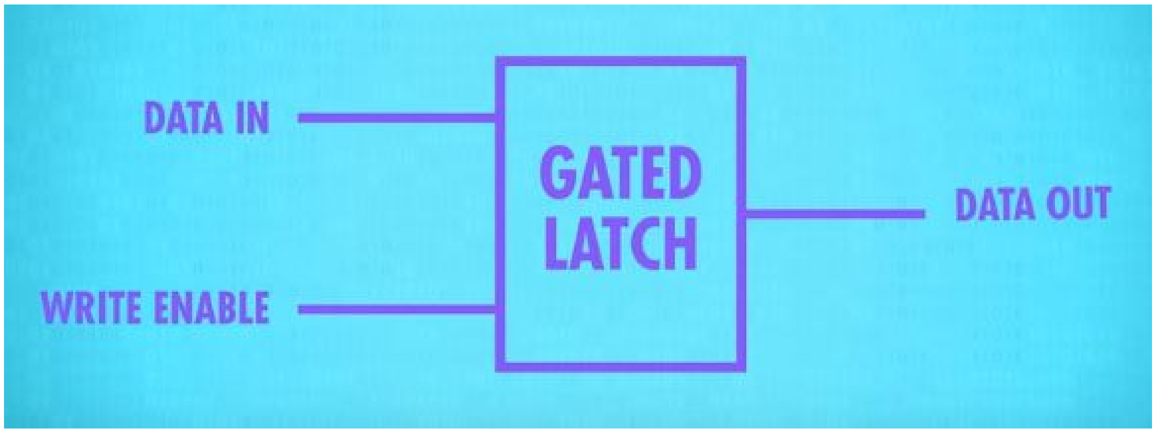
We'll start inputs A and B both at 1. 1 AND 1 outputs 1 forever. But, if we then flip input A to 0, because it's an AND gate, the output will go to 0. So this circuit records a 0, the opposite of our other circuit. Like before, no matter what input we apply to input A afterwards, the circuit will always output 0. Now we've got circuits that can record both 0s and 1s. The key to making this a useful piece of memory is to combine our two circuits into what is called the AND-OR Latch.



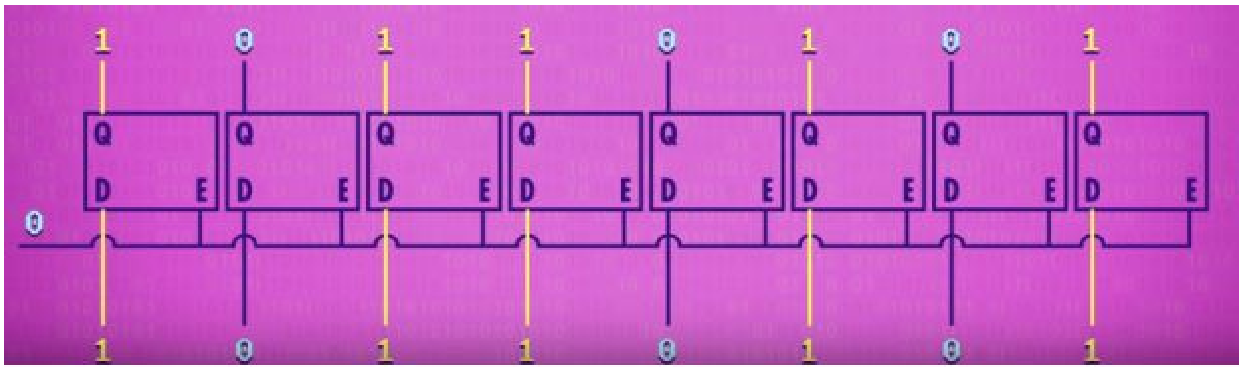
It has two inputs, a "set" input, which sets the output to a 1, and a "reset" input, which resets the output to a 0. If set and reset are both 0, the circuit just outputs whatever was last put in it. In other words, it remembers a single bit of information! Memory! This is called a “latch” because it “latches onto” a particular value and stays that way. The action of putting data into memory is called writing, whereas getting the data out is called reading. Ok, so we’ve got a way to store a single bit of information! Great! Unfortunately, having two different wires for input – set and reset – is a bit confusing. To make this a little easier to use, we really want a single wire to input data, that we can set to either 0 or 1 to store the value. Additionally, we are going to need a wire that enables the memory to be either available for writing or “locked” down – which is called the write enable line. By adding a few extra logic gates, we can build this circuit, which is called a Gated Latch since the “gate” can be opened or closed.



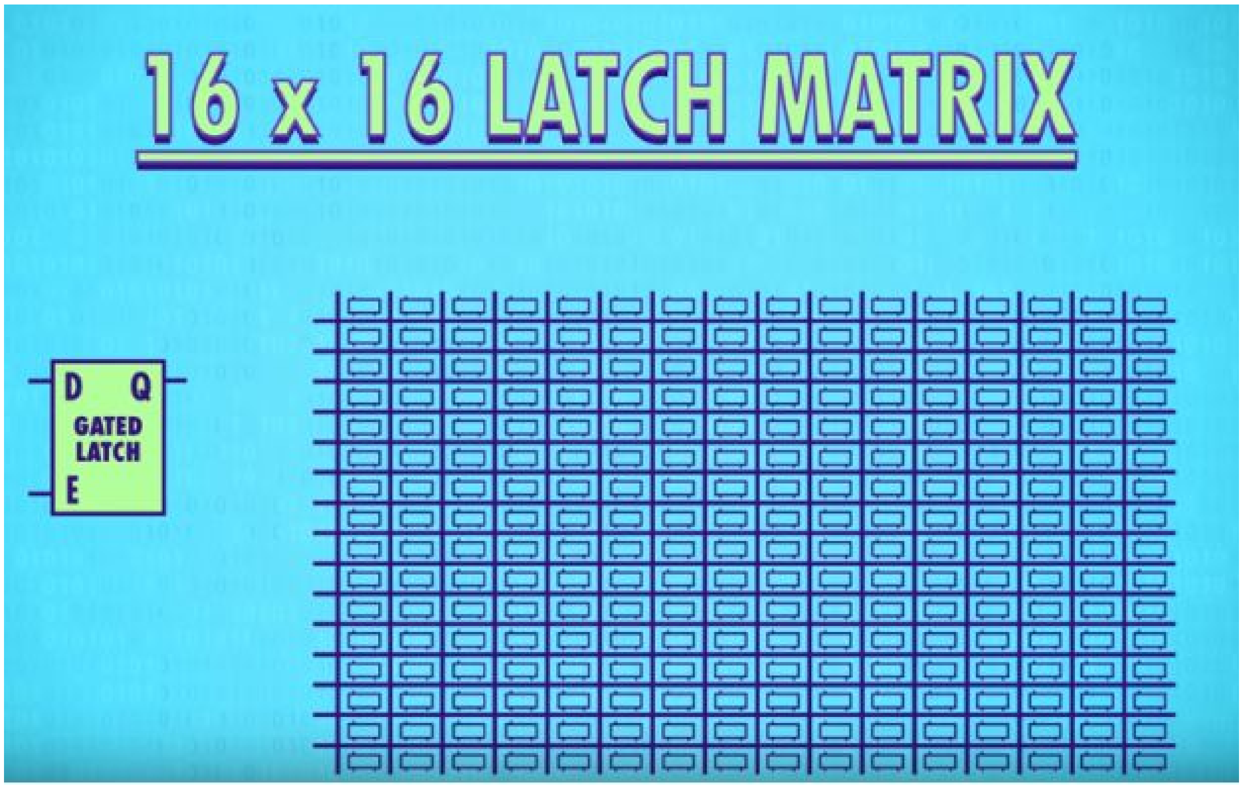
Now this circuit is starting to get a little complicated. We don’t want to have to deal with all the individual logic gates… so as before, we’re going to bump up a level of abstraction, and put our whole Gated Latch circuit in a box – a box that stores one bit. Let’s test our new component!



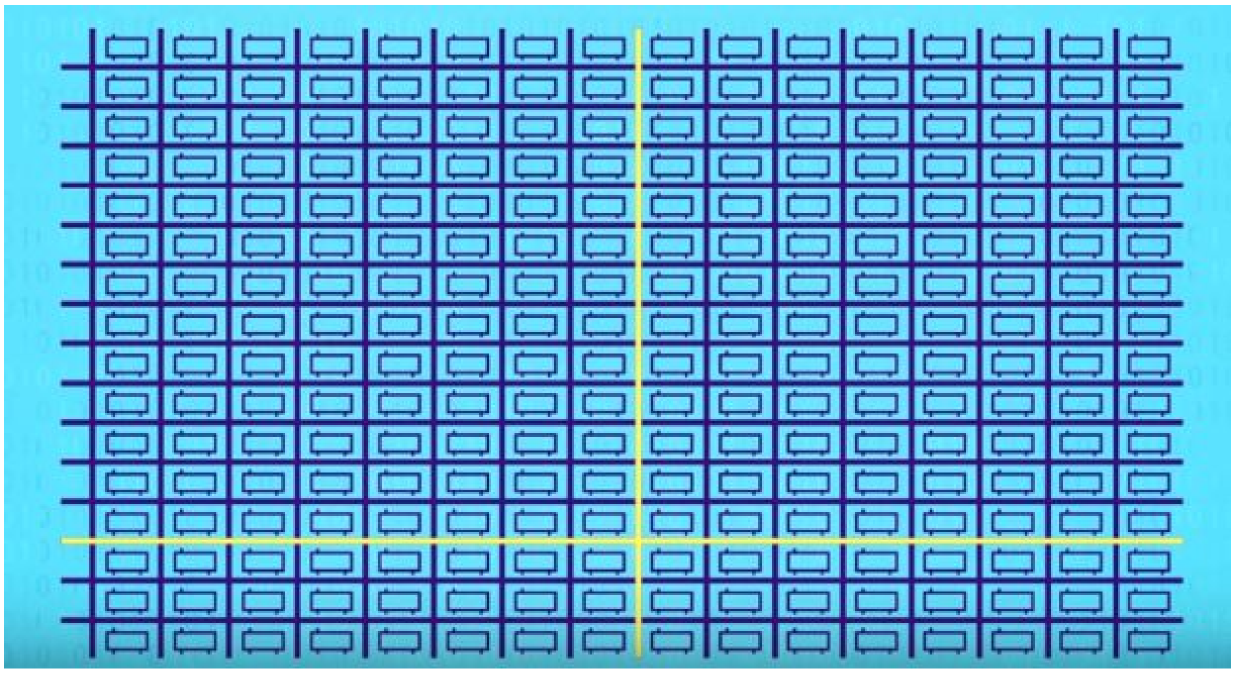
Let’s start everything at 0. If we toggle the Data wire from 0 to 1 or 1 to 0, nothing happens – the output stays at 0. That’s because the write enable wire is off, which prevents any change to the memory. So we need to “open” the “gate” by turning the write enable wire to 1. Now we can put a 1 on the data line to save the value 1 to our latch. Notice how the output is now 1. Success! We can turn off the enable line and the output stays as 1. Once again, we can toggle the value on the data line all we want, but the output will stay the same. The value is saved in memory. Now let’s turn the enable line on again use our data line to set the latch to 0. Done. Enable line off, and the output is 0. And it works! Now, of course, computer memory that only stores one bit of information isn’t very useful – definitely not enough to run Frogger. Or anything, really. But we’re not limited to using only one latch. If we put 8 latches side-by-side, we can store 8 bits of information like an 8-bit number. A group of latches operating like this is called a register, which holds a single number, and the number of bits in a register is called its width. Early computers had 8-bit registers, then 16, 32, and today, many computers have registers that are 64-bit wide. To write to our register, we first have to enable all the latches. We can do this with a single wire that connects to all of their enable inputs, which we set to 1. We then send our data in using the 8 data wires, and then set enable back to 0, and the 8-bit value is now saved in memory.



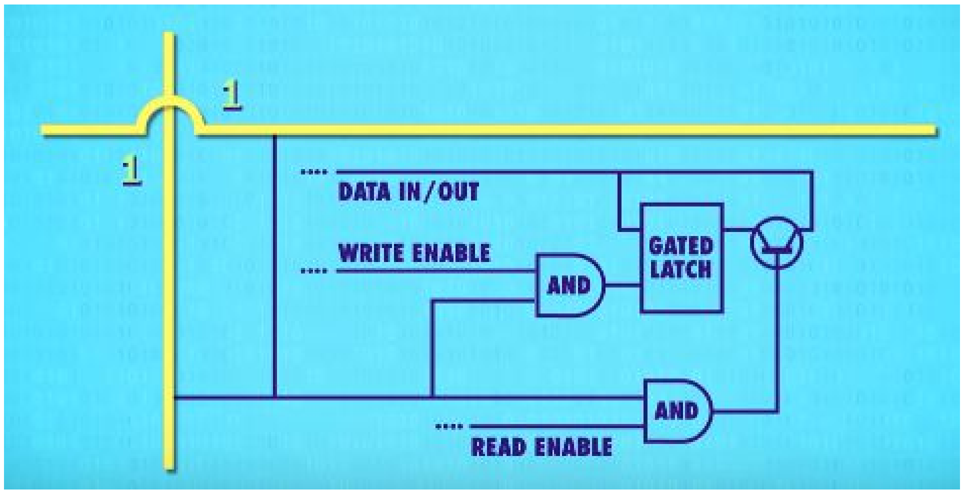
Putting latches side-by-side works ok for a small number of bits. A 64-bit register would need 64 wires running to the data pins, and 64 wires running to the outputs. Luckily we only need 1 wire to enable all the latches, but that's still 129 wires. For 256 bits, we end up with 513 wires! The solution is a matrix!



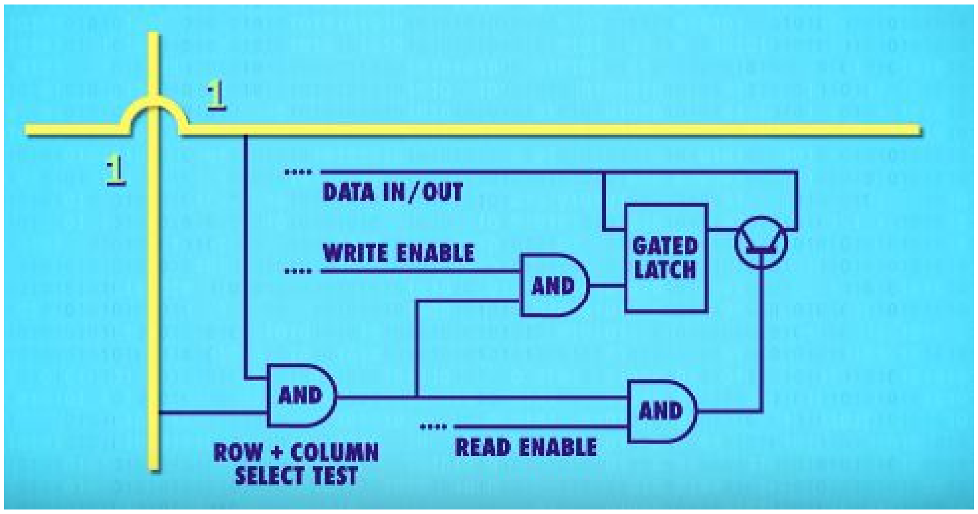
In this matrix, we don't arrange our latches in a row, we put them in a grid. For 256 bits, we need a 16 by 16 grid of latches with 16 rows and columns of wires. To active any one latch, we must turn on the corresponding row AND column wire.



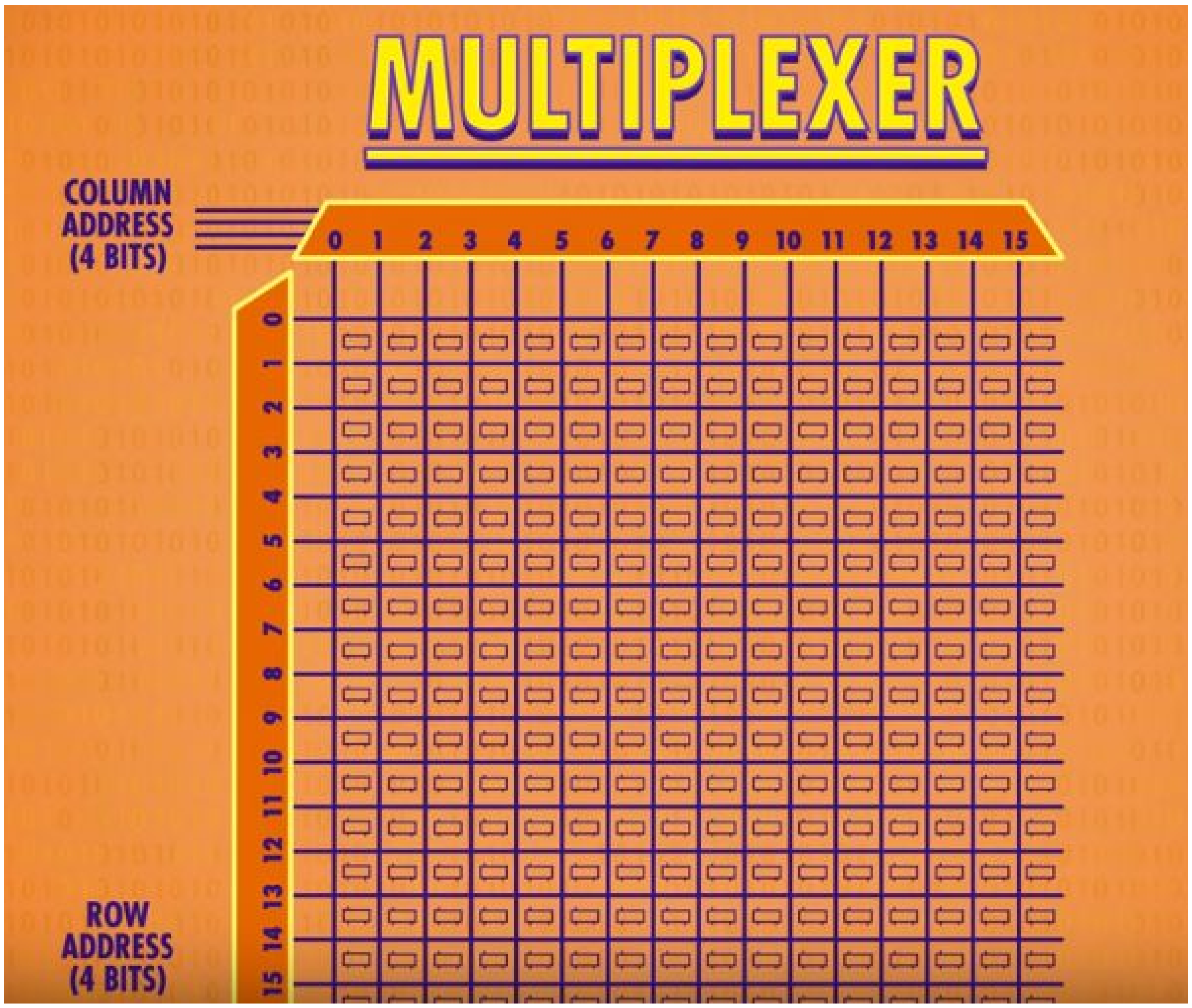
Let’s zoom is and see how this works.



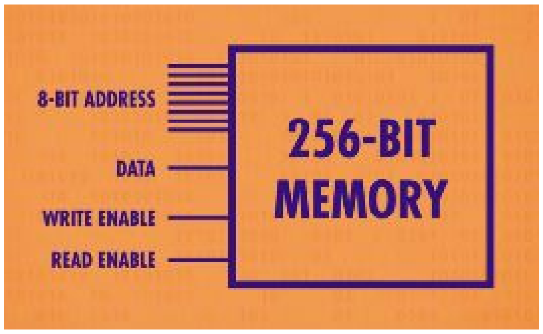
We only want the latch at the intersection of the two active wires to be enables, but all of the other latches should stay disabled. For this, we can use our trusty AND gate!



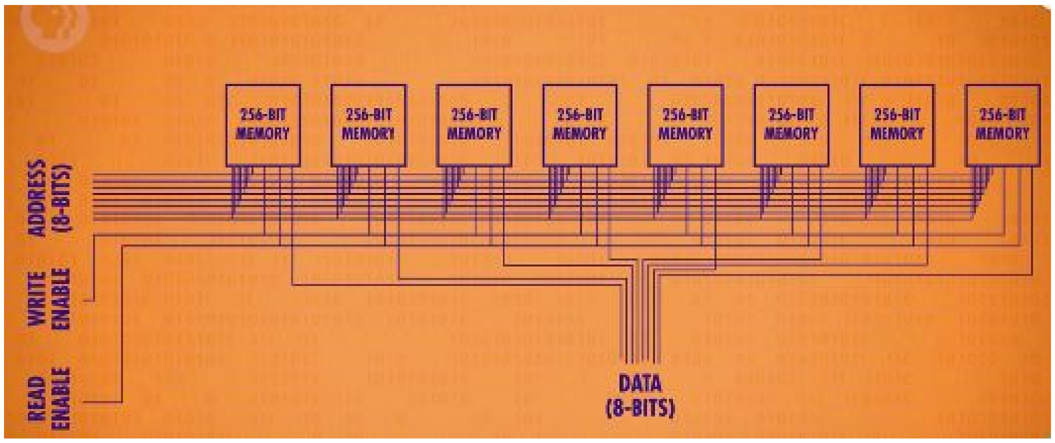
The AND gate will output a 1 only if the row and the column wires are both 1. So we can use this signal to uniquely select a single latch. This row/column setup connects all our latches with a single, shared, write enable wire. In order for a latch to become write enabled, the row wire, the column wire, and the write enable wire must all be 1. That should only ever be true for one single latch at any given time. This means we can use a single, shared wire for data. Because only one latch will ever save the data – the rest of the latches will simply ignore values on the data wire because they are not write enabled. We can use the same trick with a read enable wire to read the data later, to get the data out of one specific latch. This means in total, for 256 bits of memory, we only need 35 wires – 1data wire, 1 write enable wire, 1 read enable wire, and 16 rows and columns for the selection. That's significant wire saving! But we need a way to uniquely specify each intersection. We can think of this like a city, where you might want to meet someone at 12th avenue and 8th street – that's an address that defines intersection. The latch we just saved our one bit into has an address of row 12 and column 8. Since there is a maximum of 16 rows, we store the row address in a 4-bit number. 12 is 1100 in binary. We can do the same for the column address: 8 is 1000 in binary. So the address for the particular latch we just used can be written as 11001000. To convert from an address into something that selects the right row or column, we need a special component called a multiplexer – which is the computer component with a pretty cool name at least compared to the ALU. Multiplexers come in all different sizes, but because we have 16 rows, we need a 1 to 16 multiplexer.



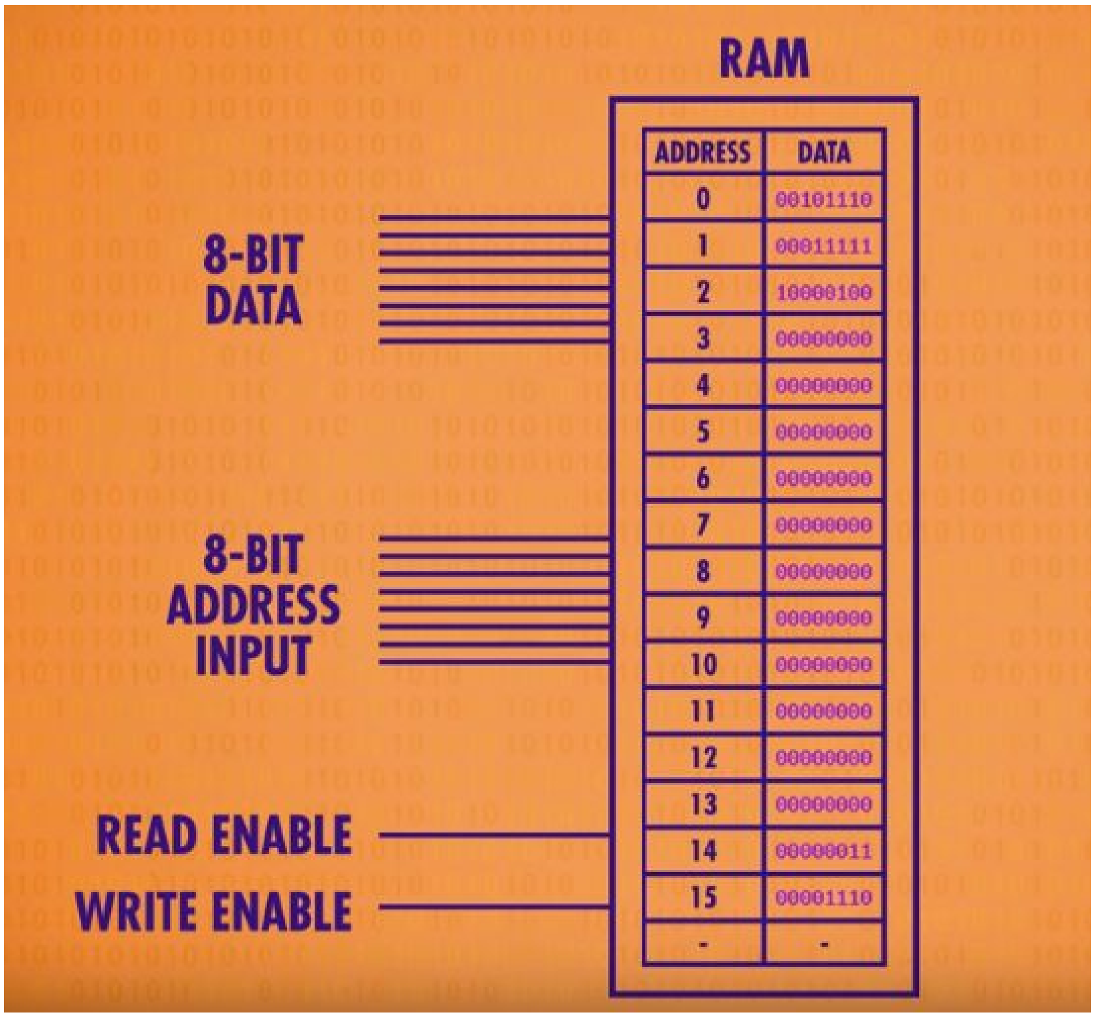
It works like this. You feed it a 4-bit number, and it connects the input line to a corresponding output line. So if we pass in 0000, it will select the very first column for us. If we pass in 0001, the next column is selected, and so on. We need one multiplexer to handle our rows and another multiplexer to handle the columns. Ok, it's starting to get complicated again, so let's make our 256-bit memory its own component. Once again a new level of abstraction! It takes an 8-bit address for input – the 4-bits for the column and 4 for the row. We also need write and read enable wires. And finally we need just one data wire, which can be used to read or write data.



Unfortunately, even 256-bits of memory is not enough to run much of anything, so we need to scale up even more! We're going to put them in a row. Just like with the registers. We'll make a row of 8 of them, so we can store an 8-bit number – also known as a byte.

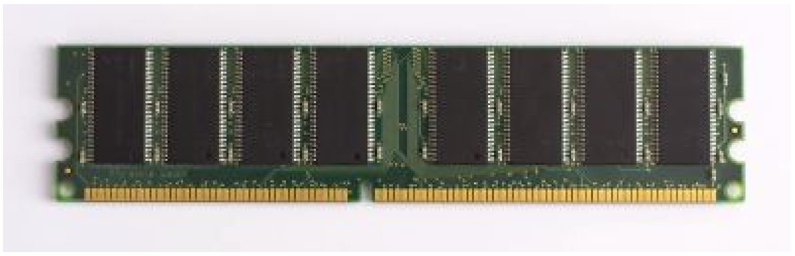


To do this, we feed the exact same address into all 8 of our 256-bit memory components at the same time, and each one saves one bit of the number. That means the component we just made can store 256-bytes at 256 different addresses. Again, to keep things simple, we want to leave behind this inner complexity. Instead of thinking of this as a series of individual memory modules and circuits, we'll think of it as a uniform bank of addressable memory. We have 256 addresses, and at each address, we can read or write an 8-bit value.



We are going to use this memory component next chapter when we build our CPU. The way that modern computers scale to megabytes and gigabytes of memory is by doing the same thing we've been doing here – keep packing up little bundles of memory into larger, and larger, and larger arrangements. As the number of memory locations grow, our addresses have to grow as well. 8 bits hold enough numbers to provide addresses for 256 bytes of our memory, but that's all. To address a gigabyte – or a billion bytes of memory – we need 32-bit addresses. An important property of this memory is that we can access any memory location, at any time, in a random order. For this reason, it's called Random-Access Memory or RAM. When you hear people talking about how much RAM a computer has – that's computer's memory. RAM is like a human's short term or working memory, where you keep track of things going on right now – like whether or not you had lunch or paid your phone bill.

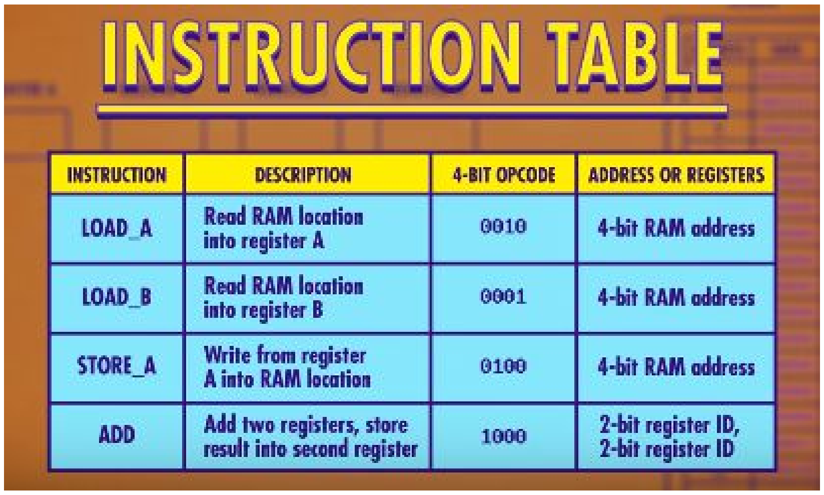
Here's an actual stick of RAM – with 8 memory modules soldered onto the board.



If we carefully opened up one of these modules and zoomed in, the first you would see are 32 squares of memory. Zoom into one of those squares, and we can see each one is comprised of 4 smaller blocks. If we zoom in again, we get down to the matrix of individual bits. This is a matrix of 128 by 64 bits. That's 8192 bits in total. Each of our 32 squares had 4 matrices, so that's 32 thousand, 7 hundred and 68 bits. And there are 32 squares in total. So all in all, that's roughly 1 million bits of memory in each chip. Our RAM stick has 8 of these chips, so in total, this RAM can store 8 million bits, otherwise known as 1 megabyte. That's not a lot of memory these days – this is a RAM module from the 1980's. Today you can buy RAM that has a gigabyte or more of memory – that's billions of bytes of memory. So, in this chapter, we built a piece of SRAM – Static Random-Access Memory – which uses latches. There are other types of RAM, such as DRAM, Flash memory, and NVRAM. These are very similar in function to SRAM, but use different circuits to store the individual bits – for example, using different logic gates, capacitors, charge traps, or memristors. But fundamentally, all of these technologies stores bits of information in massively nested matrices of memory cells. Like many things in computing, the fundamental operation is relatively simple… it's the layers and layers of abstraction that's mind blowing – like a Russian doll that keeps getting smaller and smaller and smaller.

7.The Central Processing Unit (CPU)

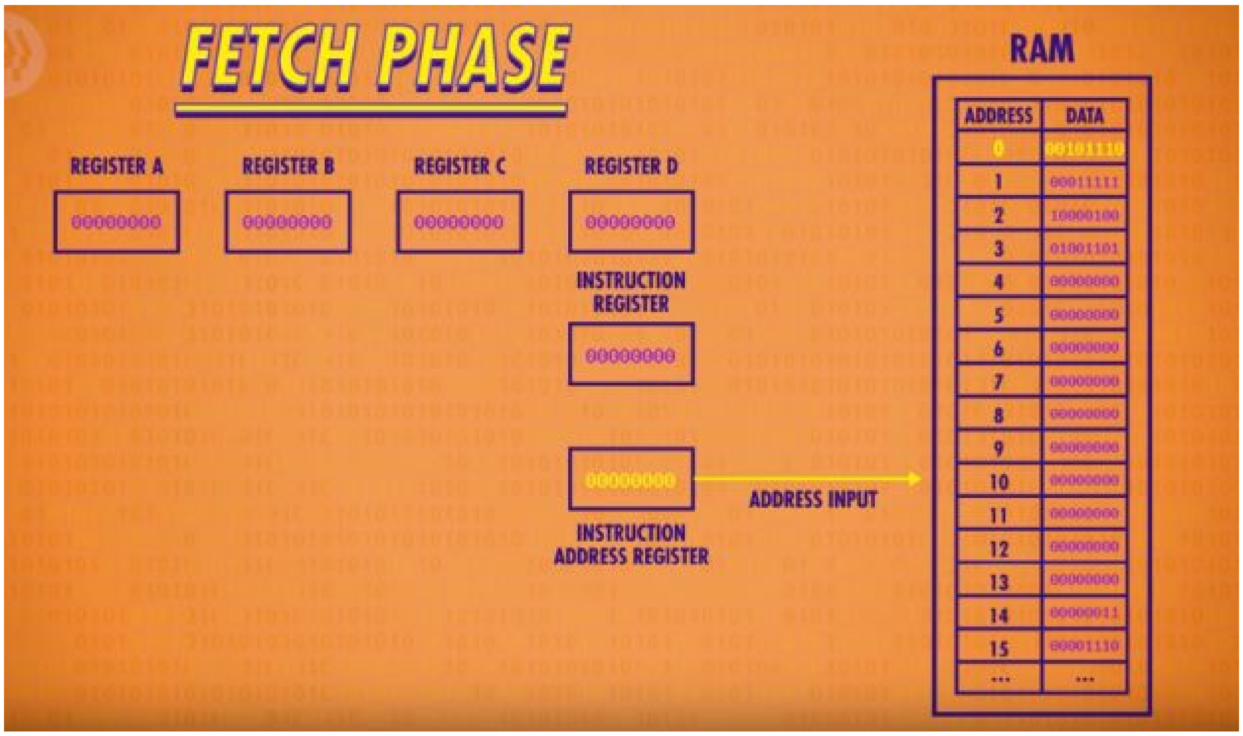
In this chapter we're going to look at processors. Just a warning though – this is probably the most complicated chapter in this book. So once you get this, you're golden. We've already made Arithmetic and Logic Unit, which takes in binary numbers and performs calculations, and we've made two types of computer memory: Registers – small, linear chunks of memory, useful for storing a single value – and then we scaled up, and made some RAM, a larger bank of memory that can store a lot of numbers located at different addresses. Now it's time to put it all together and build ourselves the heart of any computer, but without any of the emotional baggage that comes with human hearts. For computers, this is the Central Processing Unit, most commonly called the CPU. A CPU's job is to execute programs. Programs, like Microsoft Office, Safari, or your beloved copy of Half Life: 2, are made up of a series of individual operations, called instructions, because they "instruct" the computer what to do. If these are mathematical instructions, like add or subtract, the CPU will configure its ALU to do the mathematical operation. Or it might be a memory instruction, in which case the CPU will talk with memory to read and write values. There are a lot of parts in a CPU, so we're going to lay it out piece by piece, building up as we go. We'll focus on functional blocks, rather than showing every single wire. When we do connect two components with a line, this is an abstraction for all of the necessary wires. This high level view is called the microarchitecture. Ok, first, we're going to need some memory. Let's drop in the RAM module we created in the last chapter. To keep things simple, we'll assume it only has 16 memory locations, each containing 8 bits. Let's also give our processor four, 8-bit memory registers, labeled A, B, C and D which will be used to temporarily store and manipulate values. We already know that data can be stored in memory as binary values and programs can be stored in memory too.



INSTRUCTION TABLE

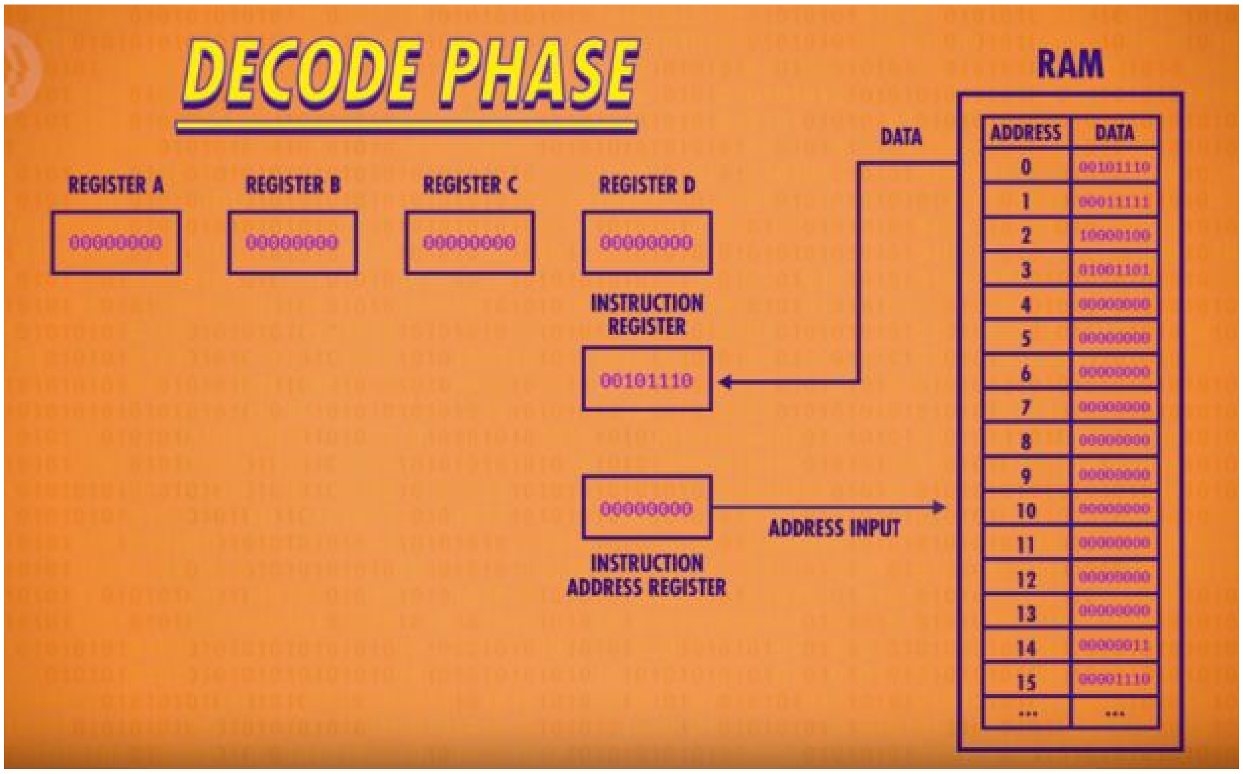
INSTRUCTION	DESCRIPTION	4-BIT OPCODE	ADDRESS OR REGISTERS
LOAD_A	Read RAM location into register A	0010	4-bit RAM address
LOAD_B	Read RAM location into register B	0001	4-bit RAM address
STORE_A	Write from register A into RAM location	0100	4-bit RAM address
ADD	Add two registers, store result into second register	1000	2-bit register ID, 2-bit register ID

We can assign an ID to each instruction supported by our CPU. In our hypothetical example, we use the first four bits to store the “operation code”, or opcode for short. The final four bits specify where the data for that operation should come from – this could be registers or an address in memory. We also need two more registers to complete our CPU. First, we need a register to keep a register to keep track of where we are in a program. For this, we use an instruction address register, which as the name suggests, stores the memory address of the current instruction. And then we need the other register to store the current instruction, which we’ll call the instruction register. When we first boot up our computer, all of our registers start at 0. As an example, we’ve initialized our RAM with a simple computer program that we’ll to though in this chapter. The first phase of a CPU’s operation is called the fetch phase. This is where we retrieve our first instruction. First, we wire our Instruction Address Register to our RAM module. The register’s value is 0, so the RAM returns whatever value is stored in address 0. In this case, 00101110.



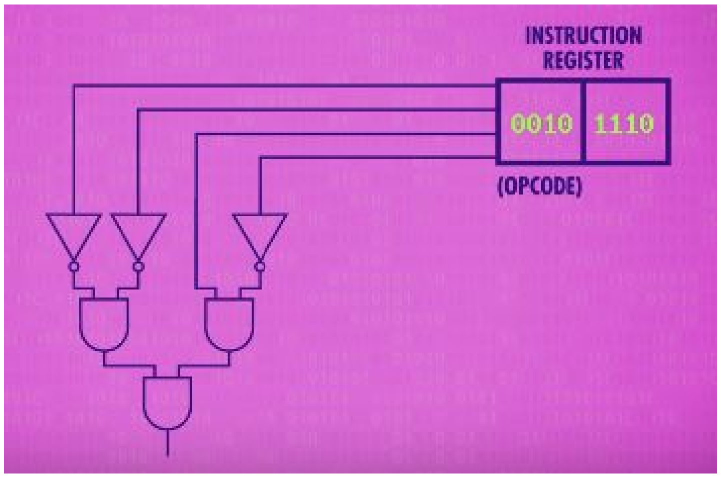
ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

Then this value is copied into our instruction register. Now that we've fetched an instruction from memory, we need to figure out what that instruction is so we can execute it. That is run it. Not kill it. This is called decode phase.

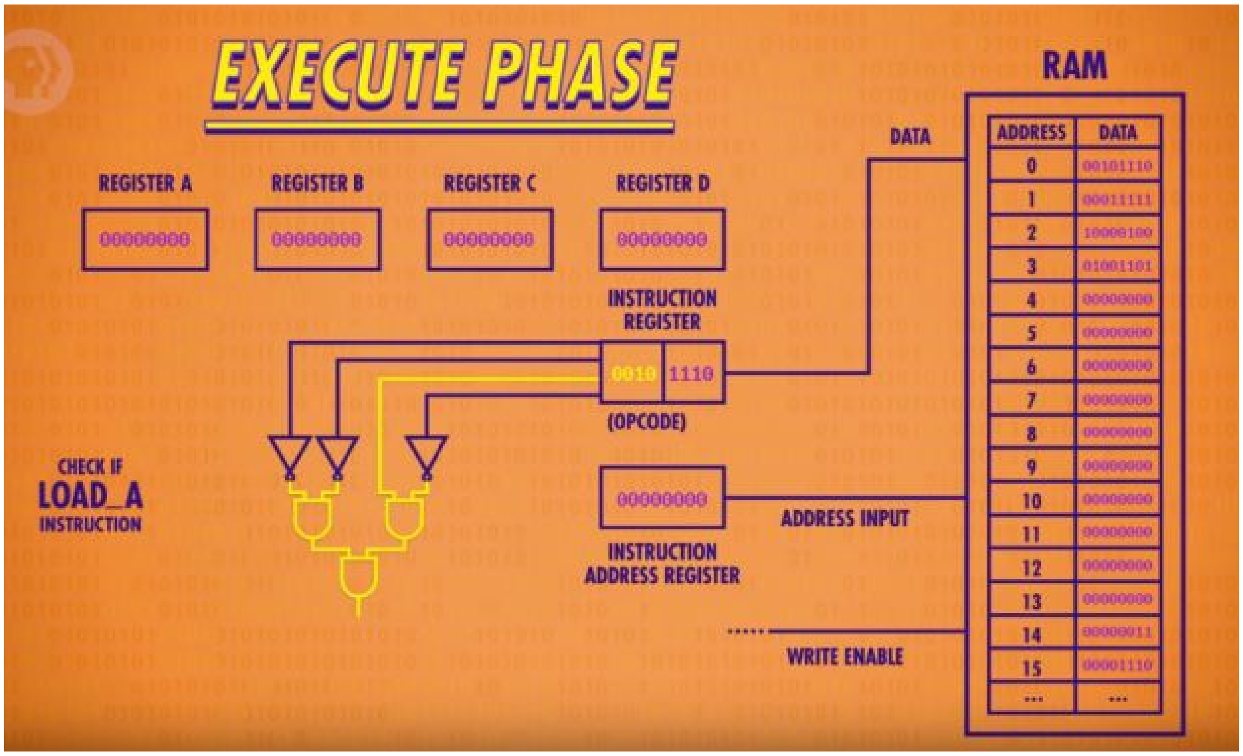


ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

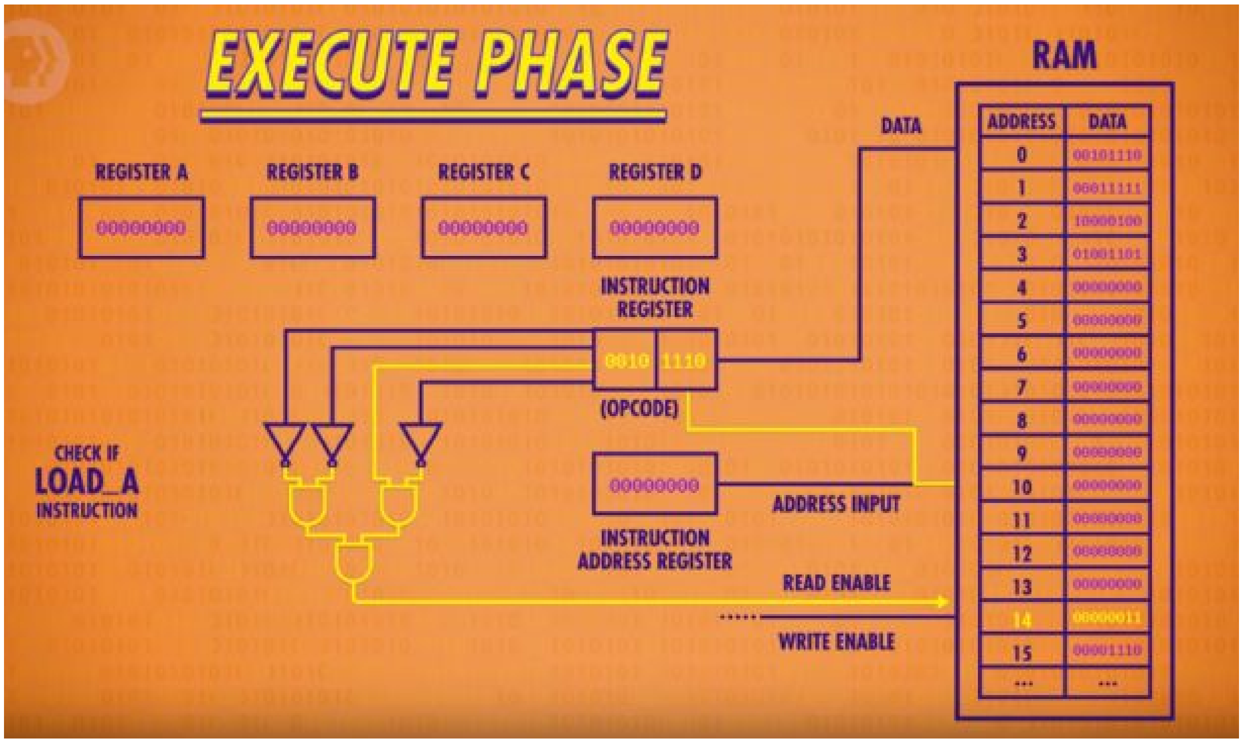
In this case the opcode, which is the first four bits, is: 0010. This opcode corresponds to the “LOAD A” instruction, which loads a value from RAM into register A (you can see it on the instruction table). The RAM address is the last four bits of our instruction which are 1110, or 14 in decimal. Next, instructions are decoded and interpreted by a Control Unit. Like everything else we’ve built, it too is made out of logic gates.



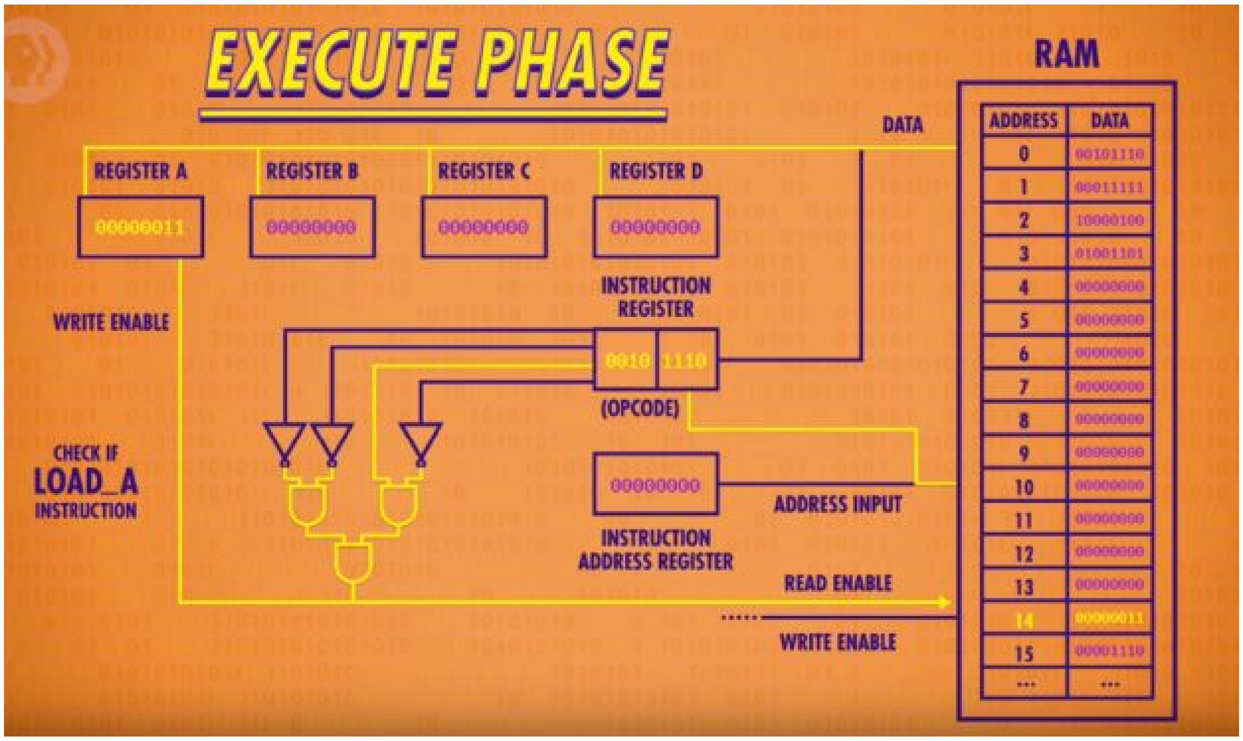
For example, to recognize a LOAD A instruction, we need a circuit that checks if the opcode matches 0010 which we can do with a handful of logic gates. Now that we know what instruction we’re dealing with, we can go ahead and perform that instruction which is the beginning of the execute phase!



Using the output of our LOAD_A checking circuit, we can turn on the RAM's read enable line and send in address 14.

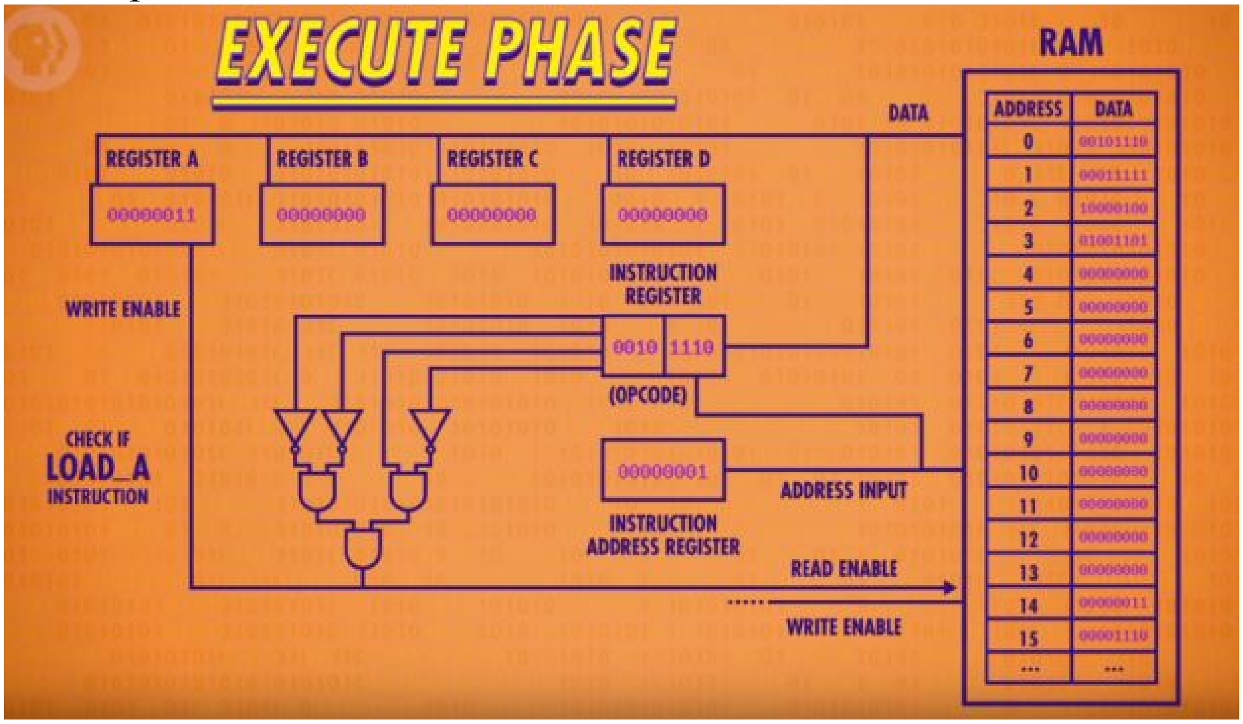


The RAM retrieves the value at that address. Which is 00000011, or 3 in decimal. Now, because this is a LOAD_A instruction, we want that value to only be saved into Register A and not any of the other registers. So if we connect the RAM’s data wires to our four data registers, we can use our LOAD_A check circuit to enable the write enable only for Register A.



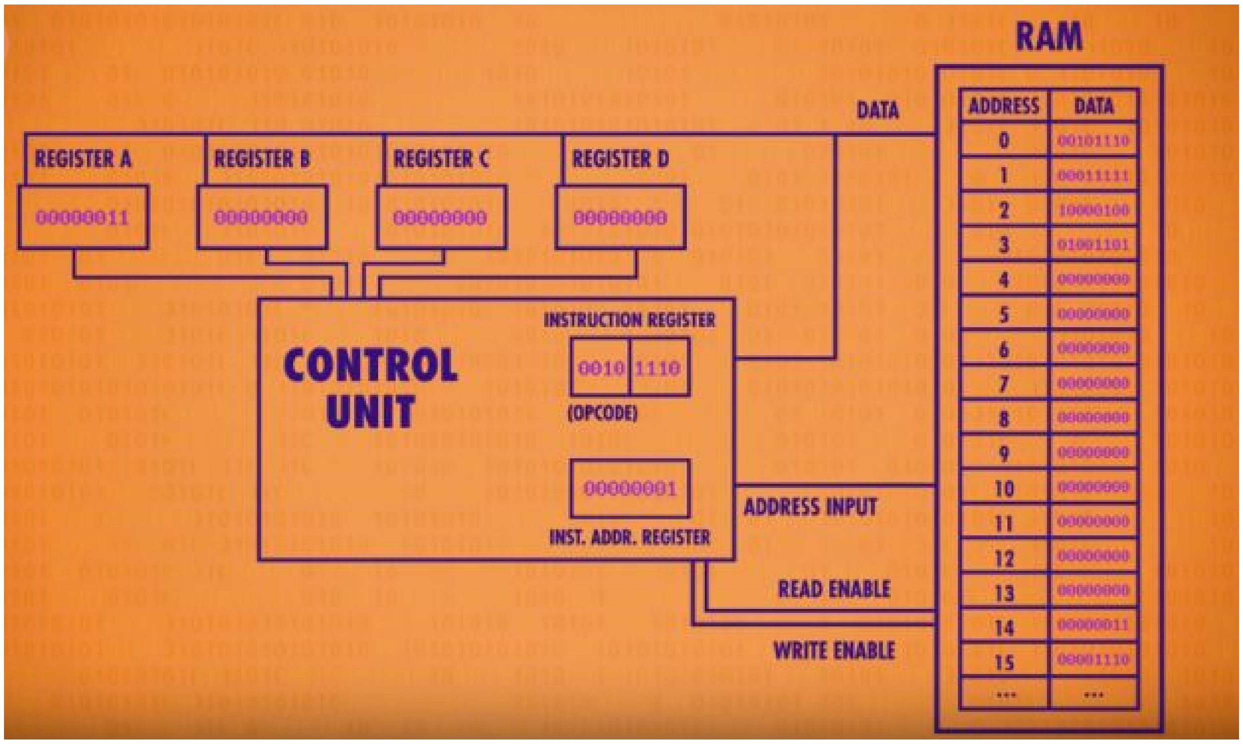
ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

And there you have it – we’ve successfully loaded the value at RAM address 14 into Register A. We’ve completed the instruction, so we can turn all of our wires off, and we’re ready to fetch the next instruction in memory. To do this, we increment the Instruction Address Register by 1 which completes the execute phase.



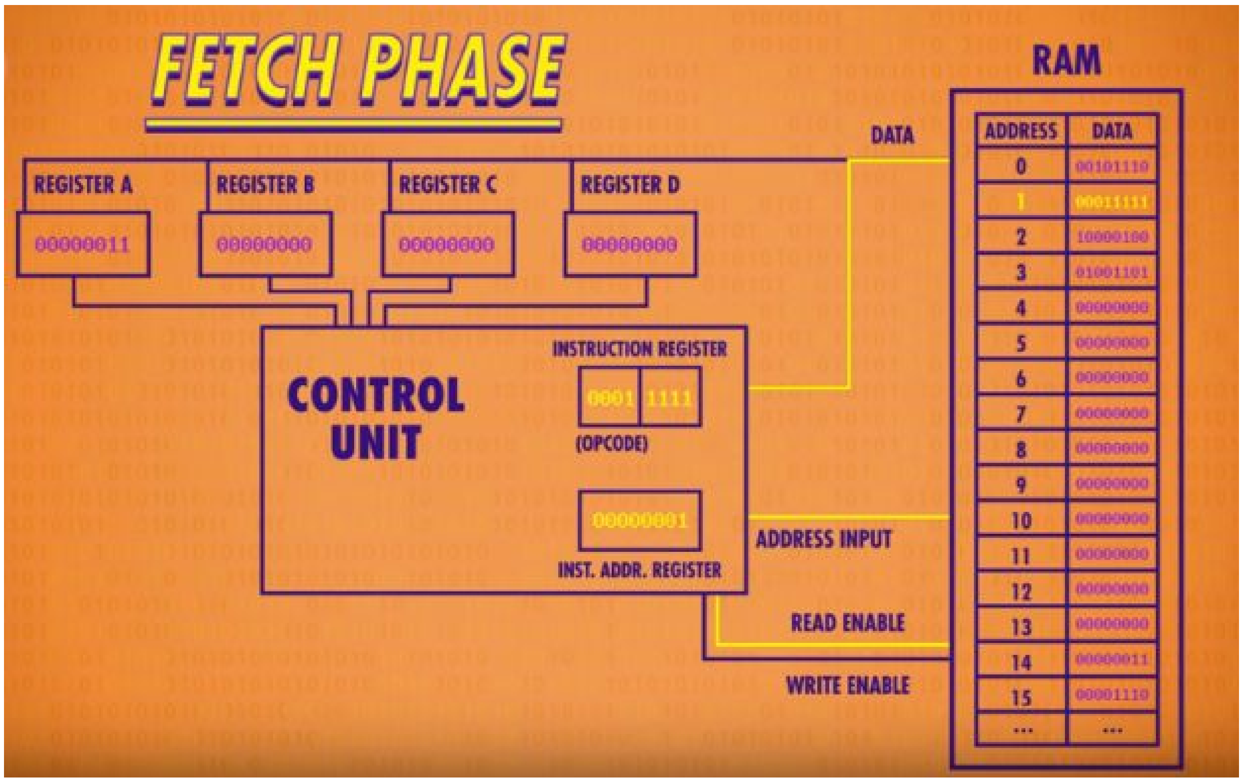
ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

LOAD_A is just one of several possible instructions that our CPU can execute. Different instructions are decoded by different logic circuits, which configure the CPU's components to perform that action. Looking at all those individual decode circuits is too much detail, so since we looked at one example, we're going to go head and package them all up as a Single Control Unit to keep things simple.

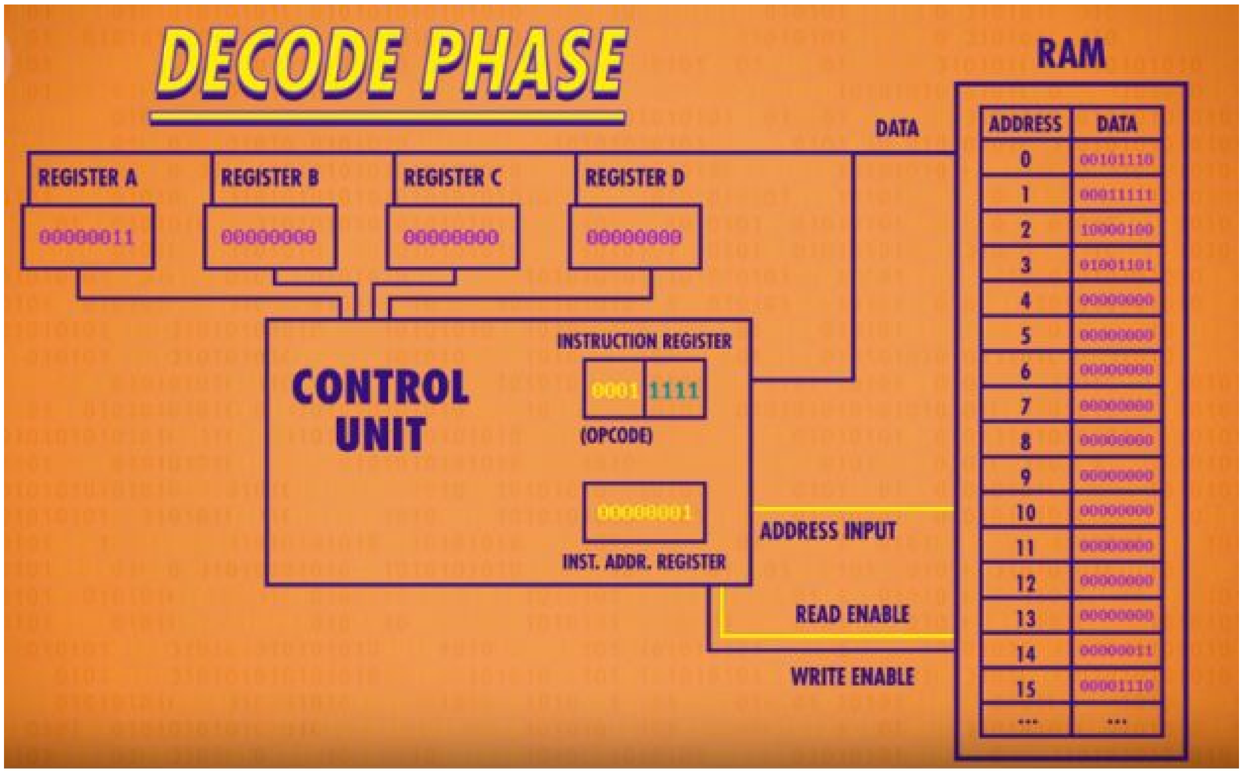


ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

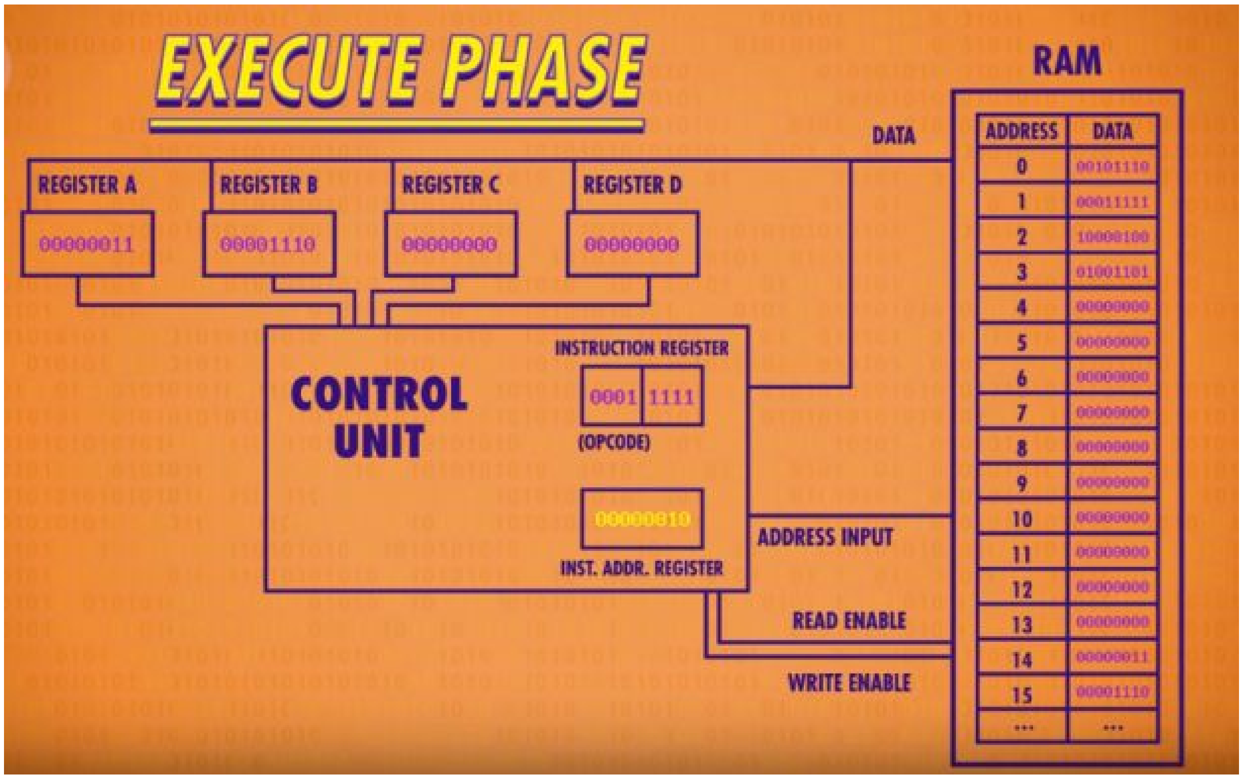
That's right a new level of abstraction. The Control Unit is comparable to the conductor of an orchestra, directing all of the different parts of the CPU. Having complated one full fetch/decode/execute cycle, we're ready to start all over again, beginning with the fetch phase. The Instruction Address Register now has the value 1 in it, so the RAM gives us the value stored at address 1, which is 0001 1111.



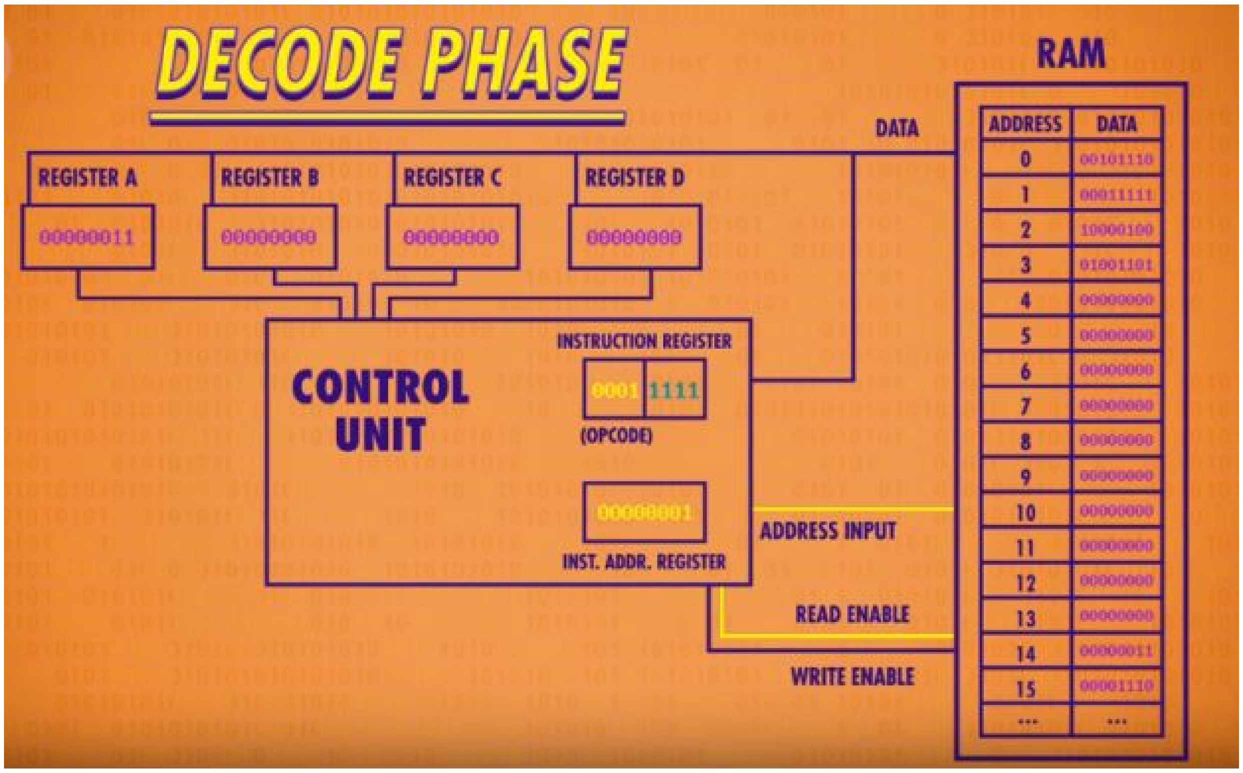
On to the decode phase! 0001 is the "LOAD B" instruction, which moves a value from RAM into Register B (from Instruction Table). The momory location this time is 1111, which is 15 in decimal.



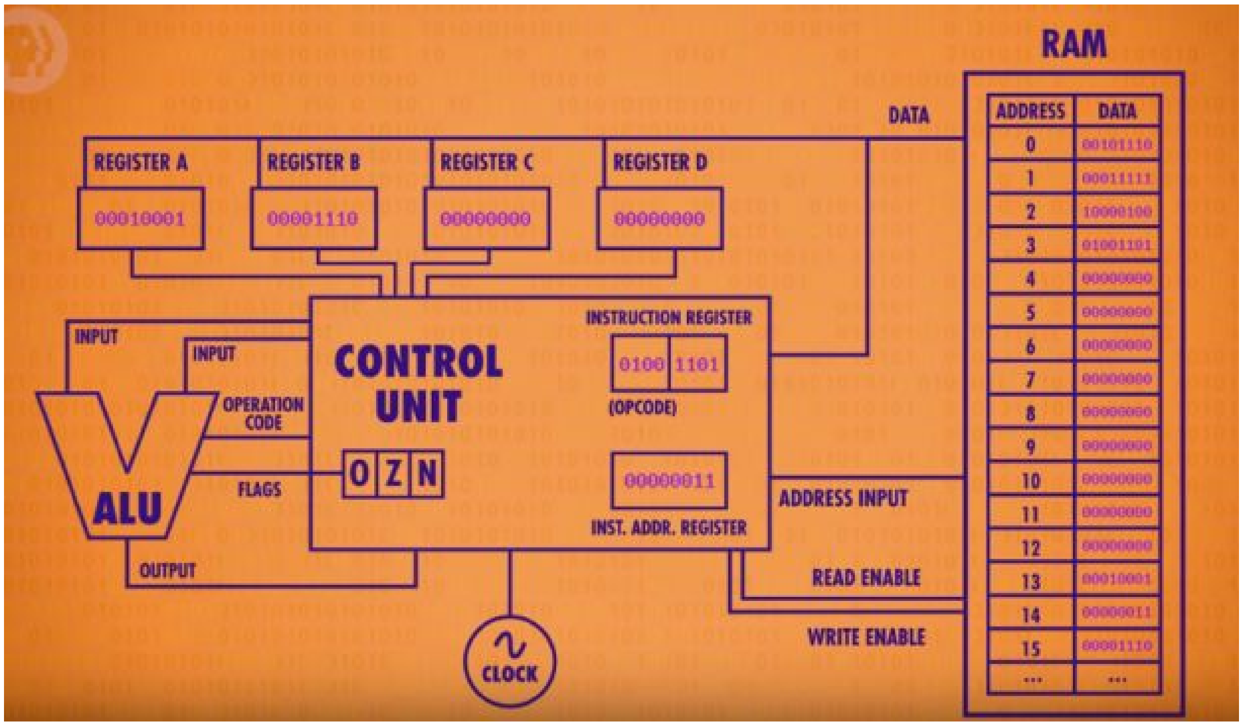
Now to the execute phase! The Control Unit configures the RAM to read address 15 and configures Register B to recive the data. Bingo, we just saved the value 00001110, or the number 14 in decimal, into Register B. Last thing to do is increment our instruction address register by 1, and we're done with another cycle.



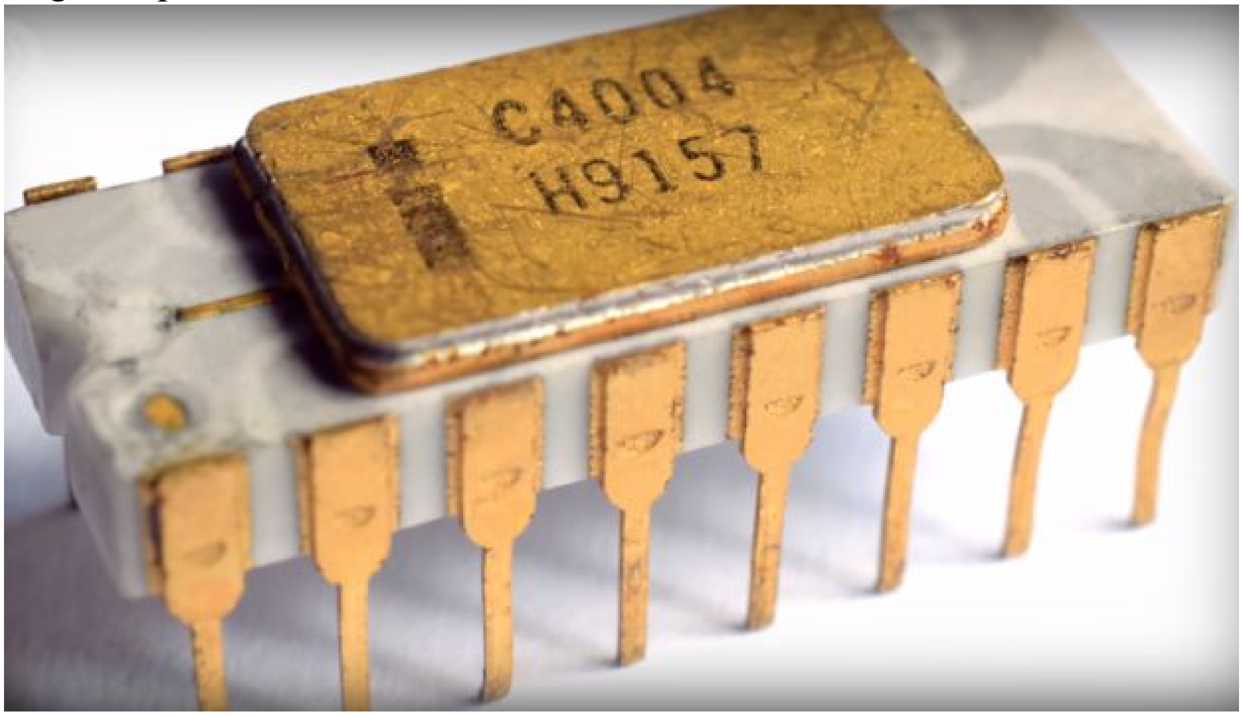
Our next instruction is a bit different. Let's fetch it. 1000 01 00. That opcode 1000 is an ADD instruction (from Instruction Table). Instead of an 4-bit RAM address, this instruction uses two sets of 2 bits. Remember that 2 bits can encode 4 values, so 2 bits is enough to select any one of our 4 registers. The first set of 2 bits is 01, which in this case corresponds to Register B, and 00, which is Register A. So "1000 01 00" is the instruction for adding the value in Register B into the value in register A. so to execute this instruction, we need to integrate the ALU we made in chapter 5 into our CPU.



The Control Unit is responsible for selecting the right registers to pass in as inputs, and configuring the ALU to perform the right operation. For this ADD instruction, the Control Unit enables Register B and feeds its value into the first input of the ALU. It also enables Register A and feeds it into the second ALU input. As we already discussed, the ALU itself can perform several different operations, so the Control Unit must configure it to perform an ADD operation by passing in the ADD opcode. Finally, the output should be saved into Register A. But is can't be written directly because the new value would ripple back into the ALU and then keep adding to itself. So the Conterol Unit uses an internal register to temporarily save the output, turn off the ALU, and then write the value into the proper destination register. In this case, our inputs were 3 and 14, and so the sum is 17, or 00010001 in binary, which is now sitting in Register A. As before, the last thing to do is increment our instruction address by 1, and another cycle is complete. Okay, so let's fetch one last instruction: 01001101. When we decode it we see that 0100 is a STORE_A instruction, with a RAM address of 13. As usual, we pass the address to the RAM module, but instead of read-enabling the memory, we write-enable it. At the same time, we read-enable Register A. This allows us to use the data line to pass in the value stored in register A. Congrats, we just ran our first computer program! It loaded two values from memory, add them together, and then saved that sum back into memory. Of course, I was manually transitioning the CPU through its fetch, decode and execute phases. But there isn't a mini James Wells inside of every computer. So the responsibility of keeping the CPU ticking along falls to a component called the clock.



As its name suggests, the clock triggers an electrical signal at a precise and regular interval. Its signal is used by the Control Unit to advance the internal operation of the CPU, keeping everything in lock-step – like the dude on Roman galley drumming rhythmically at the front, keeping all the rowers synchronized… or a metronome. Of course you can't go too fast, because even electricity takes some time to travel down wires and for the signal to settle. The speed at which a CPU can carry our each step of the fetch-decode-execute cycle is called its Clock Speed. This speed is measured in Hertz – a unit of frequency. One hertz means one cycle per second. The very first, single-chip CPU was the Intel 4004, a 4-bit CPU released in 1971.



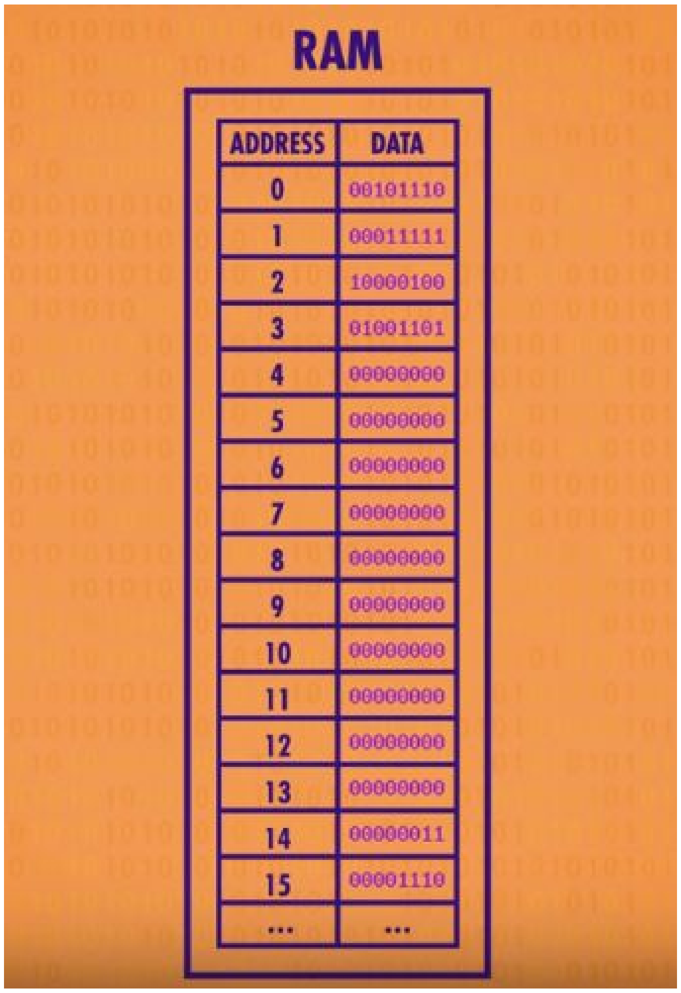
It’s microarchitecture is actually pretty similar to our example CPU. Despite being the first processor of its kind, it had a mind-blowing clock speed of 740 Kilohertz – that’s 740 thousand cycles per second. You might think that’s fast, but it’s nothing compared to the processors that we use today. One megahertz is one million clock cycles per second, and the computer you have is no debt a few gigahertz – that’s BILLIONs of CPU cycles every… single… second. Also, you may have heard of people overclocking their computers. This is when you modify the clock to speed up the tempo of the CPU – like when the drummer speeds up when the Roman Galley needs to ram another ship. Chip makers often design CPUs with enough tolerance to handle a little bit of overclocking, but too much can either overheat the CPU, or the produce gobbledygook as the signals fall behind the clock. And although you don’t hear very much about underclocking, it’s actually super useful. Sometimes it’s not necessary to run the processor at full speed… maybe the user has stopped away, or just not running a particular demanding program. By slowing the CPU down, you can save lot of power, which is important computers that run on batteries, like laptops and smartphones. To meet these needs, many modern processors can increase or decrease their speed based on demand, which is called dynamic frequency scaling. So, with the addition of clock, our CPU is complete. We can now put a box around it, and make it its own component. A new level of abstraction!

RAM, lies outside the CPU as its own component, and they communicate with each other using address, data and enable wires. Although the CPU we designed today is a simplified example, many of the basic mechanics we discussed are still found in modern processors. In the next chapter, we’re going to beef up our CPU, extending it with more instructions as we take our first baby steps into software.

8.Instructions & Programs

In the last chapter, we combined an ALU, control unit, some memory, and a clock together to make a basic, but functional Central Processing Unit – or CPU – the beating, ticking heart of a computer. We've done all the hard work of building many of these components from the electronic circuits up, and now it's time to give our CPU some actual instructions to process! The thing that makes a CPU powerful is the fact that it is programmable – if you write a different sequence of instructions, then the CPU will perform a different task. So the CPU is a piece of hardware which is controlled by easy-to-modify software!

Let's quickly revisit the simple program that we stepped though in the last chapter. The computer memory looked like this.

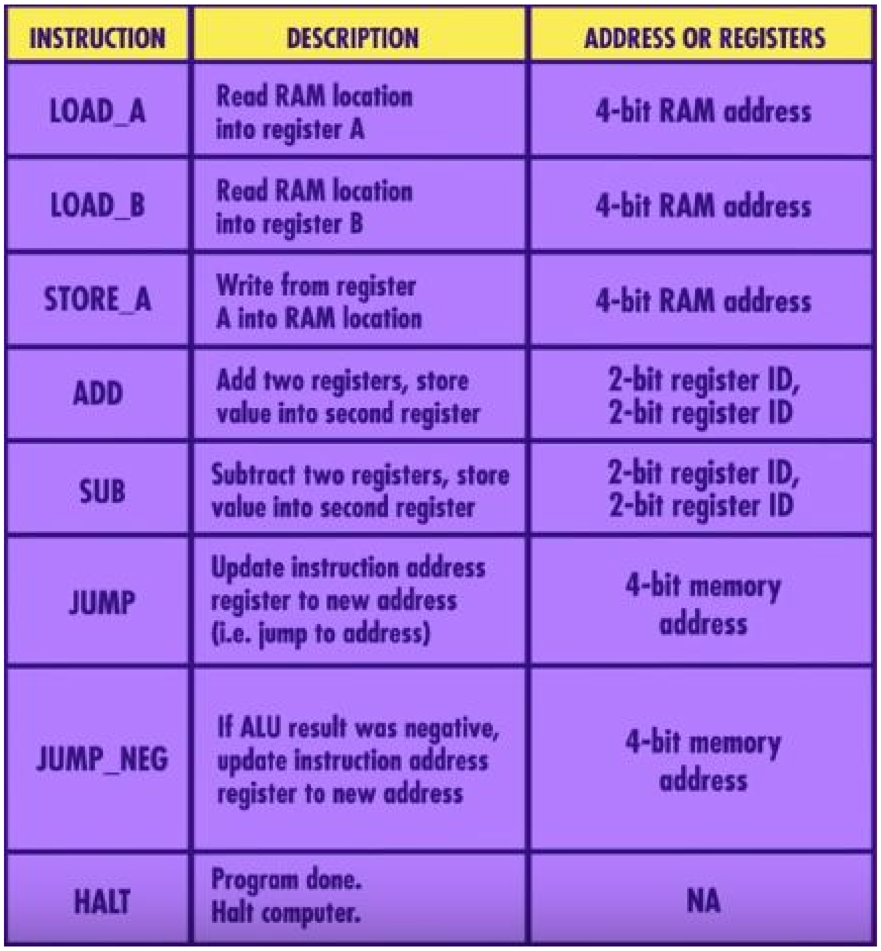


RAM

ADDRESS	DATA
0	00101110
1	00011111
2	10000100
3	01001101
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110
...	...

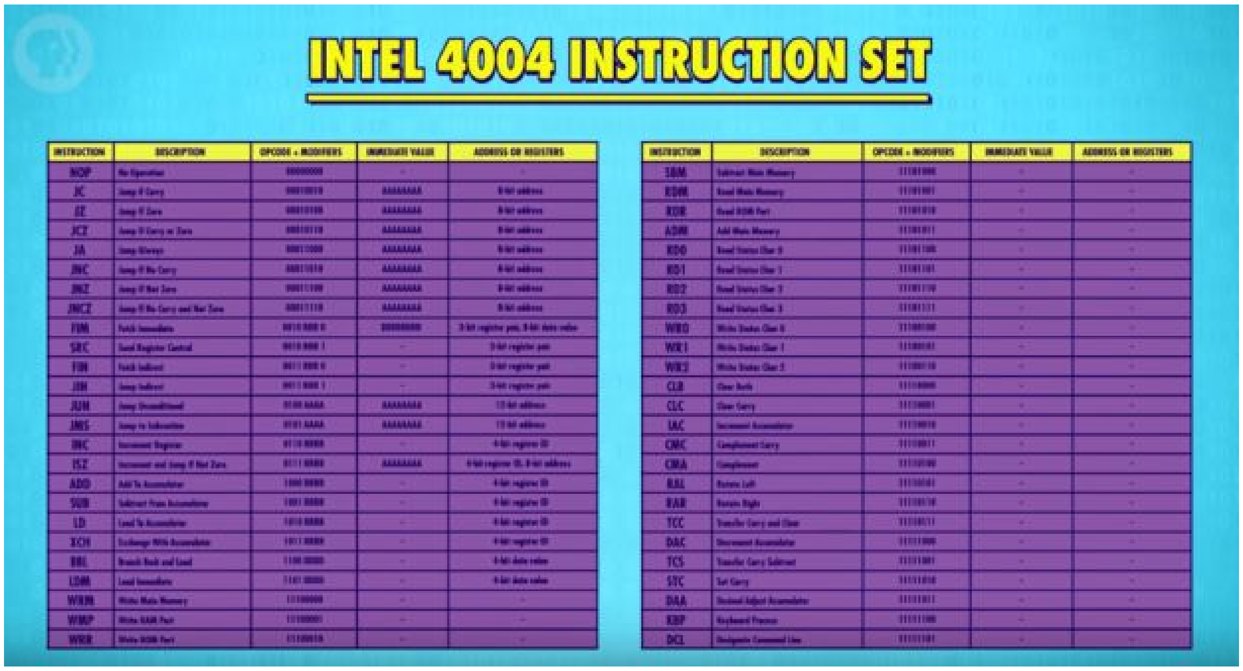
Each address contained 8 bits of data. For our hypothetical CPU, the first four bits specified the operation code, or opcode, and the second set of four bits specified an address or registers. In memory address zero we have 00101110. Again, those first four bits are our opcode which corresponds to a "LOAD_A" instruction. This instruction reads data from a location of memory specified in those last four bits of the instruction and saves it into Register A. In this case, 1110, or 14 in decimal. So let's not think of this memory address 0 as "00101110", but rather as the instruction "LOAD_A 14". That's much easier to read and understand! And we can do the same thing for the rest ot the data in memory. In this case, our program is just four instructions long (from last chapter Instruction Table), and we've put some numbers into memory too, 3 and 14. So now let's step through this program: First is LOAD_A 14, which takes the value in address 14, which is the number 3, and stores it into Register A. Then we have a "LOAD_B 15" instruction, which takes the value in memory location 15, which is the number 14, and saves it into Register B. Okay. Easy enough. But now we have an "ADD" instruction. This tells the processor to use the ALU to add two registers together, in this case, B and A are specified. The ordering is important, because the resulting sum is saved into the second register that's specified. So in this case, the resulting sum is saved into Register A. And finally, our last instruction is "STORE_A 13", which instructs the CPU to write whatever value is in Register A into memory location 13. Yesss! Our program adds two numbers together. That's about as exciting as it gets when we only have four instructions to play with. So let's add some more! Now we've got a subtract function, which like ADD, specifies two registers to operate on. We've also got a fancy new instruction called JUMP. As the name implies, this causes the program to "jump" to new location. This is useful if we want to change the order of instructions, or choose to skip some instructions. For example, a JUMP 0, would cause the program to go back to the beginning. At a low level, this is done by writing the value specified in the last four bits into the instruction address register, overwriting the current value. We've also added a special version of JUMP called JUMP_NEGATIVE. This only jumps the program if the ALU's negative flag is set to true.The negative flag is only set when the result of an arithmetic operation is negative (from chapter 5). If the result of the arithmetic was zero or the positive, the negative flag would not be set. So the JUMP NEGATIVE won't jump anywhere, and the CPU will just continue to the next instruction.

And finally, computers need to be told when to stop processing, so we need a HALT instruction. Our previous program really should have looked like this to be correct, otherwise the CPU would have just continued on after the STORE instruction, processing all those 0's. But there is no instruction with an opcode of 0, and so the computer would have crashed! Here's the Table.



INSTRUCTION	DESCRIPTION	ADDRESS OR REGISTERS
LOAD_A	Read RAM location into register A	4-bit RAM address
LOAD_B	Read RAM location into register B	4-bit RAM address
STORE_A	Write from register A into RAM location	4-bit RAM address
ADD	Add two registers, store value into second register	2-bit register ID, 2-bit register ID
SUB	Subtract two registers, store value into second register	2-bit register ID, 2-bit register ID
JUMP	Update instruction address register to new address (i.e. jump to address)	4-bit memory address
JUMP_NEG	If ALU result was negative, update instruction address register to new address	4-bit memory address
HALT	Program done. Halt computer.	NA

It's important to point out here that we're storing both instructions and data in the same memory. There is no difference fundamentally – it's all just binary numbers. So the HALT instruction is really important because it allows us to separate the two. Okay, so let's make our program a bit more interesting, by adding a JUMP. We'll also modify our two starting values in memory to1 and 1. Lets step though this program just as our CPU would. First, LOAD_A 14 loads the value 1 into Register A. Next, LOAD_B 15 loads the value1 into Register B. As before, we ADD registers B and A together, with the sum going into Register A. 1+1 = 2, so now Register A has the value 2 in it (stored in binary of caurse). Then the STORE instruction saves that into memory location 13. Now we hit a “JUMP 2” instruction. This causes the processor to overwrite the value in the instruction address register, which is currently 4, with the new value, 2. Now, on the processor’s next fetch cycle, we don’t fetch HALT, instead we fetch the instruction at memory location 2, which is ADD B A. We’ve jumped! Register A contains the value 2, and register B contains the value 1. So 1+2= 3, so now Register A has the value 3. We store that into memory. And we’ve hit the JUMP again, back to ADD B A. 1+3 = 4. So now register A has the value 4. See what’s happening here? Every loop, we’re adding one. Its computing up! Cooool! But notice there’s no way to ever escape. We’re noever… ever… going to get to that halt instruction, because we’re always going to hit that JUMP. This is called an infinite loop – a program that runs forever… ever… ever… ever… To break the loop, we need a conditional jump. A jump that only happens if a certain condition is met. Our JUMP_NEGATIVE is one example of a conditional jump, but computers have other types too – like JUMP IF EQUAL and JUMP IF GREATER. So let’s make our code a little fancier and step through it. Just like before, the program starts by loading values from memory into registers A and B. in this example, the number 11 gets loaded into Register A, and 5 gets loaded into Register B. Now we subtract register B from register A. That’s 11 minus 5, which is 6, and so 6 gets saved into Register A. Now we hit our JUMP NEGATIVE. The last ALU result was 6. That’s a positive number, so the negative flag is false. That means the processor does not jump. So we continue on to the next instruction… which is JUMP 2. No conditional on this one, so we jump to instruction 2 no matter what. Ok, so we’re back at our SUBTRACT Register B from Register A. 6 minus 5 equals 1. So 1 gets saved into register A. Next instruction. We’re back again at our JUMP NEGATIVE. 1 is also a positive number, so the CPU continues on to the JUMP 2, looping back around again to the SUBTRACT instruciton. This time is different though. 1 minus 5 is negative 4. And so the ALU sets its negative flag to true for the first time. Now, when we advance to the next instruction, JUMP_NEGATIVE 5, the CPU executes the jump to memory location 5. We’re out of the infinite loop! Now we have a ADD B to A. Negative 4 plus 5, is positive 1, and we save that into Register A. Next we have a STORE instruction that saves Register A into memory address 13. Lastly, we hit our HALT instruction and the computer rests. So even though this program is only 7 instructions long, the CPU ended up executing 13 instructions, and that’s because it looped twice internally. This code calculated the reminder if we divide 5 into 11, which is one. With a few extra lines of code, we could also keep track of how many loops we did, the count of which would be how many times 5 went into 11… we did two loops, so that means 5 goes into 11 two times… with a reminder of 1. And of course this code could work for any two numbers, which we can just change in memory to whatever we want: 7 and 81, 18 and 54, it doesn't matter – that's the power of software! Software also allowed us to do something our hardware could not. Remember, our ALU didn't have the functionality to divide two numbers, instead it's the program we made that gave us that functionality. And then other programs can use our divide program to do even fancier things. And you know what that means. New level of abstraction! So, our hypothetical CPU is very basic – all of its instructions are 8 bits long, with the opcode occupying only the first four bits. So even if we used every combination of 4 bits, our CPU would only be able to support a maximum of 16 different instructions. On top of that, several of our instructions used the last 4 bits to specify a memory location. But again, 4 bits can only encode 16 different values, meaning we can address a maximum of 16 memory locations – that's not a lot to work with. For example, we couldn't even JUMP to location 17, because we literally can't fit the number 17 into 4 bits. For this reason, real, modern CPUs use two strategies. The most straightforward approach is just to have bigger instructions, with more bits, like 32 or 64 bits. This is called the instruction length. Unsurprisingly. The second approach is to use variable length instructions. For example, imagine a CPU that uses 8 bit opcodes. When the CPU seems an instruction that needs no extra values, like the HALT instruction, it can just execute it immediately. However, if it seems something like JUMP instruction, it knows it must also fetch the address to jump to, which is saved immediately behind the JUMP instruction in memory. This is called, logically enough, an Immediate Value. It such processor designs, instructions can be any number of bytes long, which makes the fetch cycle of the CPU a tad more complicated. Now, our example CPU and instruction set is hypothetical, designed to illustrate key working principles. So I will leave you will a real CPU example. In 1971, Intel released the 4004 processor. It was the first CPU put all into a single chip and paved the path to the Intel processors we know and love today. It supported 46 instructions, shown here.



INTEL 4004 INSTRUCTION SET

INSTRUCTION	DESCRIPTION	OPCODE + MODIFIERS	IMMEDIATE VALUE	ADDRESS OR REGISTERS
NOP	No Operation	00000000	-	-
JC	Jump If Carry	00010010	AAAAAAAA	8-bit address
JZ	Jump If Zero	00010100	AAAAAAAA	8-bit address
JCZ	Jump If Carry or Zero	00010110	AAAAAAAA	8-bit address
JA	Jump Always	00011000	AAAAAAAA	8-bit address
JNC	Jump If No Carry	00011010	AAAAAAAA	8-bit address
JNZ	Jump If Not Zero	00011100	AAAAAAAA	8-bit address
JNCZ	Jump If No Carry and Not Zero	00011110	AAAAAAAA	8-bit address
FIM	Fetch Immediate	0010 RRR 0	DDDDDDDD	3-bit register pair, 8-bit data value
SRC	Send Register Control	0010 RRR 1	-	3-bit register pair
FIN	Fetch Indirect	0011 RRR 0	-	3-bit register pair
JIN	Jump Indirect	0011 RRR 1	-	3-bit register pair
JUN	Jump Unconditional	0100 AAAA	AAAAAAAA	12-bit address
JMS	Jump to Subroutine	0101 AAAA	AAAAAAAA	12-bit address
INC	Increment Register	0110 RRRR	-	4-bit register ID
ISZ	Increment and Jump If Not Zero	0111 RRRR	AAAAAAAA	4-bit register ID, 8-bit address
ADD	Add To Accumulator	1000 RRRR	-	4-bit register ID
SUB	Subtract From Accumulator	1001 RRRR	-	4-bit register ID
LD	Load To Accumulator	1010 RRRR	-	4-bit register ID
XCH	Exchange With Accumulator	1011 RRRR	-	4-bit register ID
BBL	Branch Back and Load	1100 DDDD	-	4-bit data value
LDM	Load Immediate	1101 DDDD	-	4-bit data value
WRM	Write Main Memory	11100000	-	-
WMP	Write RAM Port	11100001	-	-
WRR	Write ROM Port	11100010	-	-

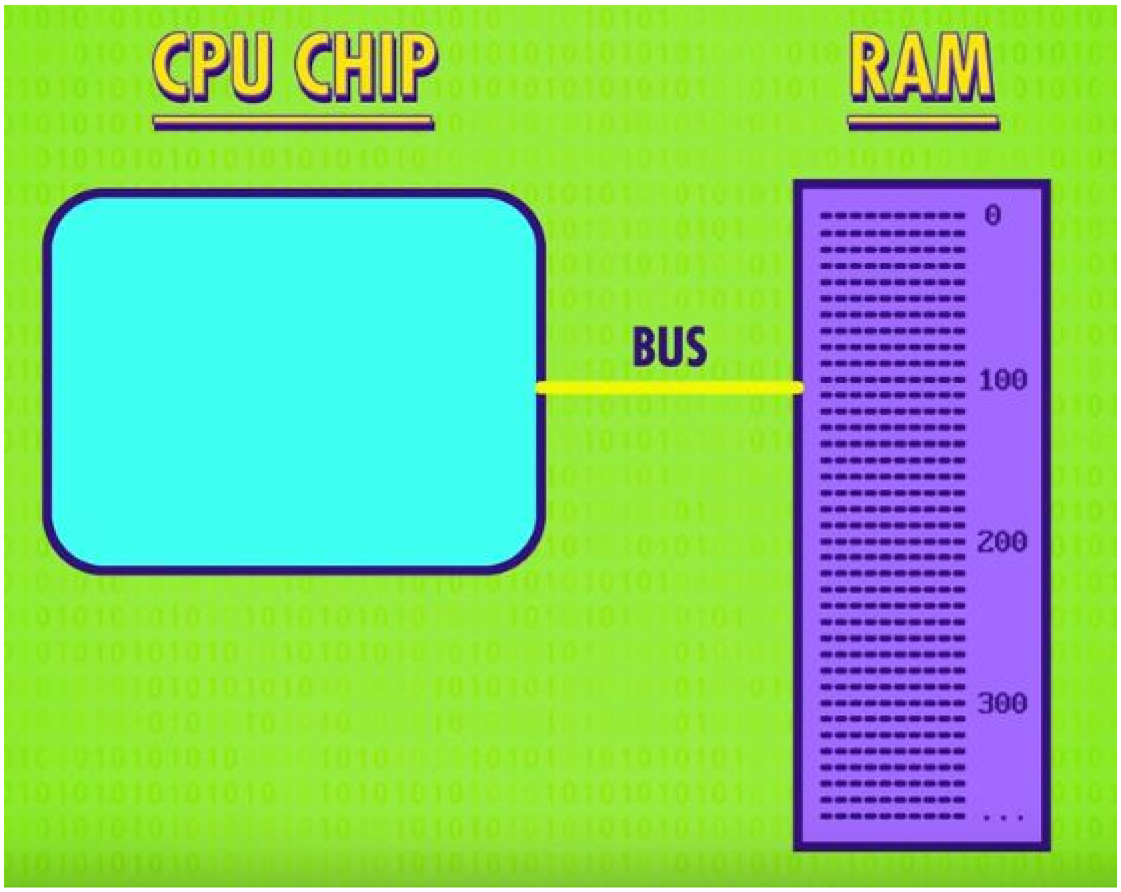
INSTRUCTION	DESCRIPTION	OPCODE + MODIFIERS	IMMEDIATE VALUE	ADDRESS OR REGISTERS
SBM	Subtract Main Memory	11101000	-	-
RDM	Read Main Memory	11101001	-	-
RDR	Read ROM Port	11101010	-	-
ADM	Add Main Memory	11101011	-	-
RD0	Read Status Char 0	11101100	-	-
RD1	Read Status Char 1	11101101	-	-
RD2	Read Status Char 2	11101110	-	-
RD3	Read Status Char 3	11101111	-	-
WR0	Write Status Char 0	11100100	-	-
WR1	Write Status Char 1	11100101	-	-
WR2	Write Status Char 2	11100110	-	-
CLB	Clear Both	11110000	-	-
CLC	Clear Carry	11110001	-	-
IAC	Increment Accumulator	11110010	-	-
CMC	Complement Carry	11110011	-	-
CMA	Complement	11110100	-	-
RAL	Rotate Left	11110101	-	-
RAR	Rotate Right	11110110	-	-
TCC	Transfer Carry and Clear	11110111	-	-
DAC	Decrement Accumulator	11111000	-	-
TCS	Transfer Carry Subtract	11111001	-	-
STC	Set Carry	11111010	-	-
DAA	Decimal Adjust Accumulator	11111011	-	-
KBP	Keyboard Process	11111100	-	-
DCL	Designate Command Line	11111101	-	-

Which was enough to build an entire working computer. And it used many of the instructions we've already know about JUMP ADD SUBTRACT and LOAD. It also uses 8-bit immediate values, like we looked at, for things like JUMPs, in order to address more memory. And processors have come a long way since 1971. A modern computer processor, like an Intel Core i7, has thousands of different instructions and instruction variants, ranging from one to fifteen bytes long. For example, there's over a dozens different opcodes just for variants of ADD! And this huge growth in instruction set size in due in large part to extra bells and whistles that have been added to processor designs overtime, which we'll look at in the next chapter.

9.Advanced CPU designs

As we looked at throughout the series, computers have come a long way from mechanical devices capable of maybe one calculation per second, to CPUs running at kilohertz and megahertz speeds. The computer you have is almost certainly running at Gigahertz speeds – that's billions of instructions executed every second. Which, trust me, is a lot of computation! In the early days of electronic computing, processors were typically made faster by improving the switching time of the transistor inside the chip – the ones that make up all the logic gates, ALUs and other stuff we've looked about over the past few chapters. But just making transistors faster and more efficient only went so far, so processor designers have developed various techniques to boost performance allowing not only simple instructions to run fast, but also performing much more sophisticated operations.

In the last chapter, we created a small program for our CPU that allowed us to divide two numbers. We did this by doing many subtractions in a row… so, for example, 16 diveded by 4 could be broken down into the smaller problem of 16 minus 6, minus 4, minus 4, minus 4. When we hit zero, or a negative number, we knew that we're done. But this approach gobbles up a lot of clock cycles, and isn't particularly efficient. So most computer processors today have divide as one of the instructions that the ALU can perform in hardware. Of course, this extra circuitry makes the ALU bigger and more complicated to design, but also more capable – a complexity-for-speed tradeoff that has been made many times in computing history. For instance, modern computer processors now have special circuits for things like graphics operations, decoding compressed video, and encrypting files – all of which are operations that would take many many many clock cycles to perform with standard operations. You may have even heard of processord with MMX, 3Dnow!, or SSE. This are processors with additional, fancy circuits that allow them to execute additional, fancy instructions – for things like gaming and encryption. These extensions to the instruction set have grown, and grown oevr time, and once people have written programs to take advantage of them, it's hard to remove them. So instruciton sets tend to keep getting larger and larger keeping all the old opcodes around for backwards compatibility. The Intel 4004, the first truly integrated CPU, had 46 instructions – which was enough to build a fully functional computer. But a modern computer processor had thousands of different instructions, which utilize all sorts of clever and complex internal circuitry. Now, high clock speeds and fancy instruction sets lead to another problem – getting data in and out of the CPU quickly enough. It's like having a powerful steam locomotive, but no way to shovel in coal fast enough. In this case, the bottleneck is RAM. RAM is typically a memory module that lies outside the CPU. This means that data has to be transmitted to and from RAM along sets of data wires, called a bus.



This bus might only be a few centimeters long, and remember those electrical signals are traveling the near the speed of light, but when you are operating at gigahertz speeds – that's billionths of a second – even this small delay starts to become problematic. It also takes time for RAM itself to lookup the address, retrieve the data, and configure itself for output. So a "load from RAM" instruction might take dozens of clock cycles to complete, and during this time the processor is just sitting there idly waiting for the data. One solution is to put a little piece of RAM right on the CPU – called a cache. There isn't a lot of space on a processor's chip, so most caches are just kilobytes or maybe megabytes in size, where RAM is usually gigabytes. Having a cache speeds things up in a clever way. When the CPU requests a memory location from RAM, the RAM can transmit not just one single value, but a whole block of data. This takes only a little bit more time then transmitting s single value, bit it allows this data block to be saved into the cache. This tends to be really useful because computer data is often arranged and processed sequentially. For example, let say the processor is totaling up daily sales for a restaurant. It starts by fetching the first transaction from RAM at memory location 100. The RAM, instead of sending back just that one value, sends a block of data, from memory location 100 though 200, which are them all copied into the cache. Now, when the processor requests the next transaction to add to its running total, the value at address 101, the cache will say "Oh, I've already got that value right here, so I can give it to you right away!" And there's no need to go all the way to RAM. Because the cache is so close to the processor, it can typically provide the data in a single clock cycle – no waiting required. This speeds things up tremendously over having to go back and forth to RAM every single time. When data requested in RAM is already stored in the cache like this it's called a cache hit, and if the data requested isn't in the cache, so you have to go to RAM, it's a called a cache miss. The cache can also be used like a scratch space, storing intermediate values when performing a longer, or more complicated calculation. Continuing our restaurant example, let's say the processor has finished totaling up all of the sales for the day, and wants to store the result in memory address 150. Like before, instead of going back all the way to RAM to save that value, it can be stored in cached copy, which is faster to save to, and also faster to access later if more calculations are needed. But this introduces an interesting problem – the cache's copy of the data is now different to the real version stored in RAM. This mismatch has to be recorded, so that at some point everything can get synced up. For this purpose, the cache has a special flag for each block of memory it stores, called the dirty bit – which might just be the best term computer scientists have ever invented. Most often this synchronization happens when the cache is full, but a new block of memory is being requested by the processor.

Before the cache erases the old block to free up space, it checks its dirty bit, and if it's dirty, the old block of data is written back to RAM before loading in the new block. Another trick to boost cpu performance is called instruction pipeling. Imagine you have to wash an entire hotel's worth of sheets, but you've only got one washing machine and one dryer. One option is to do it all sequentially: put a batch of sheets in the washer and wait 30 minutes for it to finish. Then take the wet sheets out and put them in the dryer and wait another 30 minutes for that to finish. This allows you to do one batch of sheets every hour. But, you can speed things up even more if you parallelieze your operation. As before, you start off putting one batch of sheets in the washer. You wait 30 minutes for it to finish. Then you take the wet sheets our and put them in the dryer. But this time, instead of just 30 minutes for the dryer to finish, you simultaneously start another load in the washing machine. Now you've got both machines going at once. Wait 30 minutes, and one batch is now done, one batch is half done, and another is ready to go in. This effectively doubles your throughput. Processor designs can apply the same idea. In chapter 7, our example processor performed the fetch-decode-execute cycle sequentially and in a continuous loop: Fetch-decode-execute, fetch-decode-execute, fetch-decode-execute, and so on. This meant our design required three clock cycles to execute one instruction. But each of these stages uses a different part of the CPU, meaning there is an opportunity to parallelize! While one instruction is getting executed, the next instruction could be getting decoded, and the instruction beyond that fetches from memory. All of these separate processes can overlap so that all parts of the CPU are active at any given time. In this pipelined design, an instruction is executed every single clock cycle which triples the throughput. But just like with caching this can lead to some tricky problems. A big hazard is a dependency in the instrucitons. For example, you might fetch something that the currently executing instruction is just about to modify, which means you'll end up with the old value in the pipeline. To compensate for this, pipelined processors have to look ahead for data dependencies, and if necessary, stall their pipelines to avoid problems. High and processors, like those found in laptops and smartphones, so one step further and can dynamically reorder instructions with dependencies in order to minimize stalls and keep the pipeline moving, which is called out-of-order execution. As you might imagine, the circuits that figure this all out are incredibly complicated.

Nonetheless, pipelining is tremendously effective and almost all processors implement it today. Another bit hazard are conditional jump instructions – we talked about one example, a JUMP NEGATIVE, in the last chapter. This instructions can change the execution flow of a program depending on a vlue. A simple pipelined processor will perform a long stall when it seems a jump instruction, waiting for the value to be finalized. Only once jump outcome is known, does the processor start refilling its pipeline. But, this can produce long delays, so high-end processors have some tricks to deal with this problem too. Imagine an upcoming jump instruction as a fork in a road – a branch. Advanced CPUs guess which way they are going to go, and start filling their pipeline with instructions based off that guess – a technique called speculative execution. When the jump instruction is finally resolved, if the CPU guessed currently, then the pipeline is already full of the correct instructions and it can motor along withour delay. However, if the CPU guessed wrong, it has to discard all its speculative results and perform a pipeline flush – sort of like when you miss a turn and have to do a u-turn to get back on route, and stop your GPS's insistent shuting. To minimize the effects of these flushes, CPU manufactures have developed sophisticated ways to guess which way branches will go, called branch prediction. Instead of being a 50/50 guess, today's processors can often guess with over 90% accuracy! In an ideal case, pipelining lets you complete one instruction every single clock cycle, but then superscalar processors came along which can execute more than one instruction per clock cycle. During the execute phase even in a pipelined design, whole areas of the processor might be totally idle. For example, while executing an instruction that fetches a value from memory, the ALU is just going to be sitting there, not doing a thing. So why not fetch-and-decode several instructions at once, and whenever possible, execute instructions that require different parts of the CPU all at the same time!? But we can take this one step further and add duplicate circuitry for popular instructions. For example, many processors will have four, eight or more identical ALUs, so they can execute many mathematical instructions in parallel! OK, the techniques we've looked at so far primarily optimize the execution throughput of a single stream of instructions, but another way to increase performance is to run several streams of instructions at once with multi-core processors. You might have heard of dual core or quad core processors. This means there are multiple independent processing units inside of a single CPU chip. In many ways, this is very much like having multiple separate CPUs, but because they're tightly integrated, they can share some resources, like cache, allowing the cores to work together on shared computations. But, when more cores just isn't enough, you can build computers with multiple independent CPUs! High end computers, often need the extra horsepower to keep it silky smooth for the hundreds of people watching videos simultaneously. Two- and four-processor configuration are the most common right now, but every now and again even that much processing power isn't enough. So we humans get extra ambitious and build ourselves a supercomputer! If you're looking to do some really monster calculations – like simulating the formation of the universe – you'll need some pretty serious compute power. A few extra processors in a desktop computer just isn't going to cut it. You're going to need a lot of processors. The world's fastest computer is located in The National Supercomputing Center in Wuxi, China. The Sunway TaihuLight contains a brain-melting 40,960 CPUs, each with 256 cores! That's over ten million cores in total… and each one of those cores runs at 1.45 gigahertz. In total, this machine can process 93 Quadrillion – that's 93 million-billions – floating point math operations per second, knows as FLOPS. And trust me, that's a lot of FLOPS!!! No word on whether it can run Crysis at max settings, but I suspect it might. So long story short, not only have computer processors gotten a lot faster over the years, but also a lot more sophisticated, employing all sorts of clever tricks to squeeze out more and more computation per clock cycle. Our job is to wield that incredible processing power to do cool and useful things. That's the essence of programming, which we'll start in the next chapter.