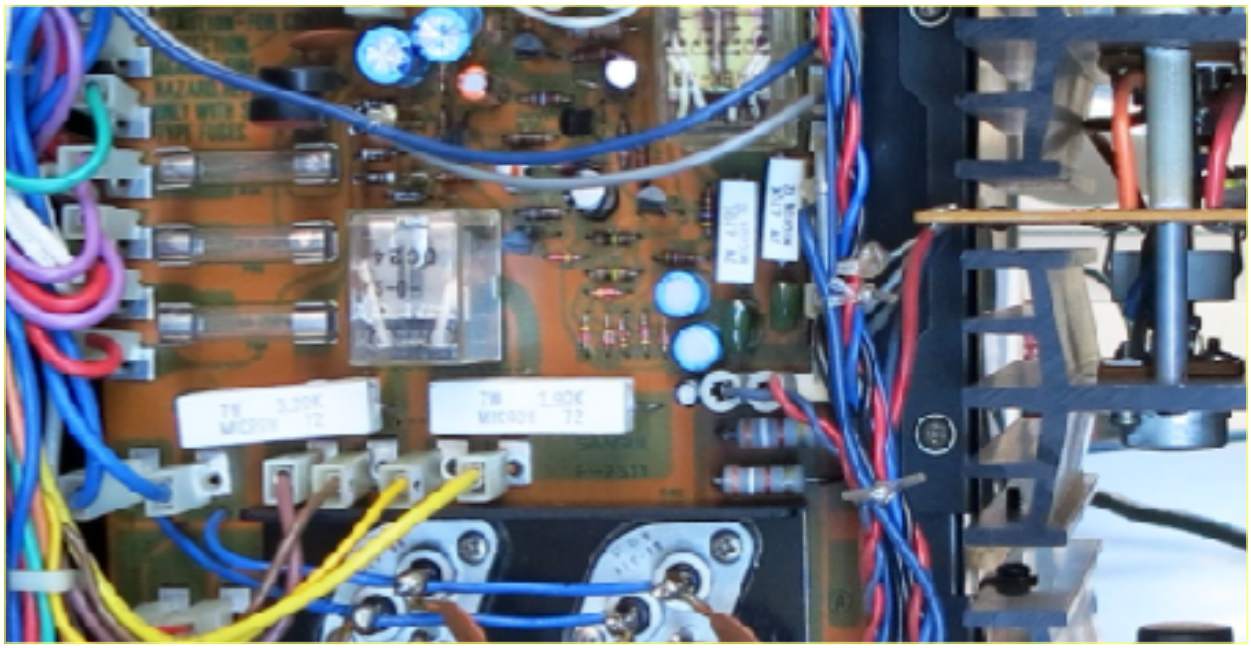
Advanced Learning

Tech Talks

Most people are told "Computers are a box of switches" or "What's inside are just ones and zeros." This is deceptively over-simplified. Yes, the math approach is binary not tens-based. The math comes out the same math concepts but it takes a lot more digits to represent the same numbers. That's okay because computers are very fast. The gain is the binary system only has two states thus the ones and zero explanation. This matches up with the box of switches that also only have two states. Two states are easily handled by an electronic transistor switches that can be made very small. The gap in the explanation is that most of these switches are used to maintain control of logic circuits, switch in and out whole bus systems, and performs math algorithms by controlling the state of the whole machine during each step. This binary math/control is ubiquitous because it simplifies the electronic circuits not because it is magic.



Macro to micro took a couple of decades but the concept didn't change

So to understand computing hardware we need to focus on building blocks not conspiracy theories. Logic gates, address busses, data bus access, and all the rest are controlled by the state of the whole machine. Each step from pulling an instruction from memory, parsing it into opcodes, address schemes, pulling data from memory, acting on that data, storing the results, and moving on to the next instruction is done by setting control logic values to be either on or off in a register that can maintain the state of the machine during each step. Many steps later, something gets done. It's the many little pieces acting in turn during each instruction cycle that gets results. It's like a clock whose ticks are controlled by an escapement for each second (It takes a tick and a tock to get a second of time). Those seconds are cogged out into minutes and hours by gear reduction all synchronized literally within the machine. In a computer, adding two numbers requires a lot of little steps before the result ends up back in memory.

This paper looks at building blocks that are part of every computer and gadgets that perform various operations that could be controlled by a microcontroller or perform stand-alone functionality. The value in being aware of these circuits means you can just build parts of a computer, but you can also design unique functionality by reusing components from a computer. This is what makes hobby projects so much fun.

In the Beginning

Within the guts of all computing devices that harbor a central processing unit, memory system, an I/O labyrinth of interconnectivity, are numerous very simple constructs. Just saying this is much more complicated than the real silicon circuits themselves. What I'm talking about are logic gates that combine together to form multiplexors, encoders, priority encoders, decoders, adders, routing gateways, arithmetic calculations among the many specific purposed circuits. And I do mean specific. Computers are state machines. That means, each and every functional state has to be repeatable and distinct. It can be combinational logic or it can be timed or synchronous logic, but even timing is under distinct control to know the state of each operation. The fundamental chunks of a processor are input, output, memory, data path, and control all orchestrated with logic gates. At the intersection of the real-world and the digital world there are, of course, special electronic circuit designs that bridge the gap. Building a memory system, whether a single register or a bank of interlinked memory bits is a specialized circuit intersection with our basic logic gates. Even these circuits are based on simple circuit elements. Once you go beyond the processor boundary past the I/O terminals, all bets are off. Electronic sensors and external controls are complex. Energy types like controlling RF and data delivery systems are

extremely complex and the software to keep up with them equally complex but how a processor is implemented in circuitry is not. Let's look at the building blocks.

Logic Gates

Let me first say, there is an associated YouTube video that goes into detail as the companion to this paper called "Core Building Blocks." The basis of controlling digital functionality is accomplished by combining various "logic gates" to perform a function. This Combinational Logic, as it is known, consists of AND, OR, XOR, and inverters. Their names describe their function and can be shown using a "truth table." AND-gates require all the input signals to be On or present for the output to go High or On or from the engineer's point of view be at a design level voltage to be used by the next logic gate. The fact there are several terms for the same result creates confusion but remember, in the end, it's a voltage (5v for TTL) or the absence of that voltage that activates the logical outcome. The OR-gate is a nexus point that allows any High signal to produce an output High. OR-gates are the connecting tissue within a logical system. XOR-gate only allows an output when only one of the inputs at a time is present, in other words, one input is exclusive to all others. I have assumed up until now we are comparing one signal to one other but the functionality is same with multiple inputs to get one output thus a gating (filtering) effect. At first, you might wonder where's the logic. Think of the AND-gate this way, "If this is true and that is true, then it follows that the outcome is true." If you're Greek, this all makes perfect sense. Same with OR and exclusive OR. The real question is how can these simple ideas amount to anything useful? Let's start with an additional circuit type called the D-flip-flop. This small circuit has two states. The circuit has complementary outputs meaning it has another one that tracks the input value and one that is always the opposite state. We place a High or Low at the D-input pin (data), and when a clock signal is applied at another input, the output tracks the D-input and that is called the Q-output. Note the Q-bar-output will be the opposite state and they are available for more processing at the same time. The second output is there as a convenience if an inverted value is used later in the circuit. The main point of the circuit is whatever value is clocked-in is stored in the circuit until another value is clocked-in at some later time. Now we have a way to store state in general for a set of data inputs.

So with gates to accomplish some logical decision making and D-flip-flops storage the result, we have another useful element called a register. The number of these flip-flops needed for the register is matched to the number of bits we want to move around as data or addresses. The memory part of the Central

Processing Unit (CPU) has a generalized set of these registers that are essential for CPU operations. Registers can be used as buffers or places to hold data temporarily or collect data until needed. Registers can be used to set up a group of control signals where each of the Q outputs of the register goes to other combinational logic for further define operations or just report current status of the processor. But notice, we have introduced a clock into the processor configuration. By controlling the clocking, we can control the state of how whole machine operation unfolds.

Design Factors

Besides logic gates and registers, we have mentioned memory and clocking. When designing a CPU processing system, we have to settle on a word size in bits to match address location size of a memory space and the size of data we can store. Part of the Operating System (OS) needs to know how big various data types are like integers, floating point numbers, strings and the like to determine how many words are needed to fit any particular data type into memory and a scheme to keep track of the address space in general, especially when what's stored takes up more than one word location. Memory units are normally set up by 8-bits groups called bytes. If a design calls for 32 bit word size that would imply there are four bytes stored at any one given address. This chunk size of a byte was chosen as the base because each American Standard Code for Information (ASCII) defined letter, number, or punctuation can be held in one byte. ASCII is a table defined by 4-bits for a row and 4-bits for a column where each letter, number, or punctuation that we are all aware of is at an intersection. So a byte of data can get you a letter you want.

Also, be aware that we have three items to keep track of to do anything useful with the memory - the address, the data, and the data type. The address bus can be any size depending on the addressing scheme to encompass the total volume of the memory space. That could be TeraBytes. The data size and its bus match the word size to efficiently stuff data into the memory at word boundaries. Gathering all these design specifications together and turning them into clumps of combinational logic and specialize circuit bits is clearly going to become unwieldily. And at the core, this is mainly a very organized bunch of switches stitched together into a set of states that do something useful like add, subtract, multiply, and divide not to mention branching, comparing, and reporting status and it is all done with a mass of little circuit components. The clocking is the way we get from one state to another no matter how small the steps are or how many steps it takes to accomplish any one given task. It's clear that the integrated

circuit is the natural path to get all this functionality into a small manageable space.

Instruction Set Architecture(ISA)

Where all of this circuit design that gives us a functional processor, memory, and the software to organize it, is called the ISA. We have to store our instructions in the memory so we need to consider word size. A Reduced Instruction Set Computer (RISC) matches each instruction length to the word size of memory. At this point, we are talking about a Central Processing Unit (CPU) and leave behind the simple logic designs. So let's say we have 32-bit words. A typical design would have six of the most-significant-bits (MSB) devoted to instruction codes or opcodes. This is enough bits to cover all the types of instructions the CPU will need. The next three sets of five bits give us a register number to store two operands and a place to store the results. That could be up to 32 registers since 5 bits max count is 32. We have 10 bits left to subdivide functions or typically take advantage of bit-shifting to double or divide by a power of two or bitwise manipulation. Other instructions will enable branching to other memory locations to facilitate decision making as in an "if statement". All of this starts to beg a way to write code that turn bits (machine code) that can be understood by humans to load into memory as instructions to get the ball rolling. Each one of these bits are used to control the state of the machine. The data is stored in another area of memory to be acting upon. Instructions are pulled from another place in memory and interpreted from the opcodes. This could be instructions from the operating system or the loaded app or program.

Each instruction bit pattern sets up a bunch of logic gates to pull data, place it into registers, move data operands into a circuit called the Arithmetic Logic Unit (ALU) that is yet more logic gates that can add, subtract, compare as AND or OR logic, increment or decrement to adjust memory locations, the main one being moving to the next instruction in memory after the previous instruction completes. Be advised that each instruction takes several micro-steps to complete so the clocking controls are paramount. The point is, that up to now we have been slamming together little electronic circuits that flip on and off and not one mention of binary counting or math operations. The math part is really straight forward because binary representation of numbers is the same as the decimal system. It just takes a lot more digits to represent the same value. The gain in using binary numbers is they can be represented with only two states. As long as you have a one to count, you can always start from there and add more counts. The beauty of binary is we can place numerical data in memory as a series of “1s” & “0s" also. It looks just like logic gate manipulation information, which

emphasizes the need to clearly separate control signals from numbers. Everything else is a matter of defining what a series of bits mean. For example the bits 01001000 or one byte, if known by programmers as string data it will be interpreted as a capital H. Here again, complexity can be represented in various ways. The 01001000 bits can be split in the middle where 0100 = 4 and 1000 = 8 so 48 in hexadecimal or just read as the whole binary value as 72 in decimal. The point being that any set of bytes can be pulled out in various representations but in the end a H is printed to a screen thus bits to intelligent information has taken place - yeah! Since all these clusters of “1s” & “0s” only make sense if we as humans decide on a convention to establish what they mean. The standard for lettering was first implemented with the American Standard Code for Information(ASCII) which gives us one byte per symbol. How to get that representation onto a screen requires a bunch more digital logic gates!

Software Abstraction

I have to assume you now have a glimpse of how logic gates end up with “1s” & “0s” in memory of a CPU design. The gorilla in the room is going from machine code to something a Web Dev programmer writes that uses this machine to deliver a useful outcome. It starts with the processor manufacturers deciding on an ISA and then writing out how to go from what the logic does to code a human can read to represent it. Once that bridge is crossed, we have abstracted hardware into software. The manufacturer tells the world in a product spec what the "assembly code" is to reach the instruction set operations. Then software developers pull together a "higher order language" that is easy to write and read that relates back to the assembly codes via a compiler or a software interpreter that converts your source code into assembly code, which is linked down to machine code. There is now a direct relationship between this higher order language, whatever it might be, down to the machine code that makes the computer sing. The first code to write is the Operating System(OS) to encompass file organization, peripheral functionality(keyboard, mouse, screen, etc.), and use the ISA and other key elements like the BIOS, which is a small memory area that tells the OS what functional features are currently available in that particular box to control. This provides flexibility for the computer as a product so changes can be made without having the redesign the whole machine. We'll discuss the software aspect next, but at this point realize we have OS code based on a spec definition of an ISA and assembly language to make it easier to construct software and the user's program and data as a minimum to make the whole thing work. The hardware is clocked so definite steps can track the state of each process and know when it's complete. Also, the clock provides a benchmark to determine how fast the machine can accomplish tasks.

Summary

Of course, this is a 70,000 foot look at what has become the most complex thing mankind has ever put together, especially when you bring in the Internet and all the data centers in the world, but realize this - it's the meager little logic gates that are the bricks in the wall. It's software that is the architect that makes it all useful. It's the beauty of humans that have the ability to abstract ideas into constructs that eventually get down to activating electronic circuits which is the thing we call a computer. Check out the associated video (Core Building Blocks) for more details.