What I know about...

How a CPU Works

Charles Clayton

Last updated: December 23, 2016

I. Introduction

Computers aren't really that smart, but they're way better than people at certain things like mathematical computations. The problem is that they need to be told very clearly and painstakingly what to do and how to do it. Programmers tell them how to do these things by writing code.

But that doesn't explain how a computer, which is a bunch of pieces of metal and silicon and doesn't have any moving parts, can take those words and make them into reality.

For instance, to create this animation, I wrote the following lines of code in the Python language:

```
circle = topics.geometry.Circle()
               .highlight(VS_COLORS.turquoise)

square = topics.characters.TextMobject("This animation")
               .highlight(VS_COLORS.violet)
               .shift(10*LEFT)
self.play(*(
     animations.GrowFromCenter(circle),
     transforms.ApplyMethod(square.shift, 10*RIGHT)
), run_time=3)
```

But how did feeding that code into a computer turn into the video you're watching?

II. Instruction Set Architecture

Well, just as I could describe an animation, there are lots of different types and levels of coding languages that can describe all sorts of behaviour. Some of this code is easy for people to write and understand.

```
x = factorial(10)
```

But in order for a computer to execute it, it needs to be compiled into something called assembly language.

```
     LOAD  R1, #10
     LOAD  R2, #1
L1:  MUL.D R2, R2, R1
     SUB.D R1, R1, #1
     BNEZ  R1, L1
```

Assembly is harder to understand, but looks simple because it is made up of very basic instructions that only do one thing, like subtract two values, multiply two values, or load a single piece of information from memory.

There are only 60 or so of these instructions, and these make up something called the instruction set architecture (ISA). These instructions are the only things that a computer can actually do. Everything else, from playing movies to (???), is executed by being simplified into a series of these simple instructions.

III. Machine Code

Okay, so we haven't actually gotten very far to answering our question yet. People wrote programs called compilers, to convert code that's easy to write into code that's harder to writer. We're sill left with the question of how the program executes the code.

The first problem is that computers can't read this code. Pieces of metal and silicon don't know how to read words or even what words are. However, they do know what voltage is. So instead of using Latin characters which have 26 characters and Arabic numerals which have ten characters, we use binary which only has two characters, 0 and 1, and we tell these to the hardware by representing the zeros as some voltage and ones as no voltage.

The nice thing about assembly code is that it can be directly translated into something called machine code, which is just ones and zeros.

As an aside, you've probably heard people talking about "ones and zeros" before, but what does that mean? Humans use a base-10 counting system, because we decided can distinguish between 10 different levels. After we get up to 10 levels, we just decide to roll it over and start counting again.

Computers can only distinguish between two levels: either there's no voltage on a wire, which we call 0, or there's some voltage, which we call 1. This is a base-2 system, binary.

When we translate our assembly into machine code, we identify our 60 instructions by numbering them instruction `0`, `1`, `2`, `3`, `4`, etc. So in binary, these are represented as `000000`, `000001`, `000010`, `000011`, `000100`, etc. We do the same with our operands, the chunks of data we're doing operations on. So each line of assembly code is represented in machine code as 32 ones and zeros.

```
00000101101000000101101O
01011010100101011101111
00000100000010010101000
11110101011101101000111
10100010110111011000000
```

IV. Program Counter

This sequence of ones and zeros representing instructions is then stored in the computer's memory. The computer knows what instruction to execute using a register called the program counter, which stores the memory address of the instruction being executed. After the instruction is complete, the program counter is incremented to point at the next memory location.

```
00000101101000000101101O
```

```
01011010100101011011111
00000100000001001010100   <-  PC
11110101011011010001111
10100010110111010000000
```

In the case of loops or conditional instructions, the PC can be modified to point back and redo a previous instruction or to hop over instructions that don't need to be executed.

V. Stages

There are five stages to completing each instruction: Instruction Fetch, Instruction Decode, Execute, Memory Access, Write-Back.

Each stage is completed entirely in hardware.

A. Fetch

I'll talk in more detail about how memory works and how data is fetched from it another time, but at this point suffice it to say that the computer grabs the instruction code from the memory location pointed at by the PC and stores it in a register called the Instruction Register, IR.

B. Decode

The decode hardware here will use the ones and zeros in the instruction register to forward the operands to the corresponding functional unit. Functional units are pieces of hardware that execute a task like add, multiply, or compare.

C. Execute

The execute takes place in the functional unit. These are built using some clever design with logic gates, which are blocks of hardware that will give a high or low voltage based on inputs of high or low voltages.

D. Memory Access

If the instruction is to load a value from memory or to store a value into memory, that will happen here.

E. Write-back

After the instruction is complete, the value output from the functional unit will be written to the destination register identified by the instruction.