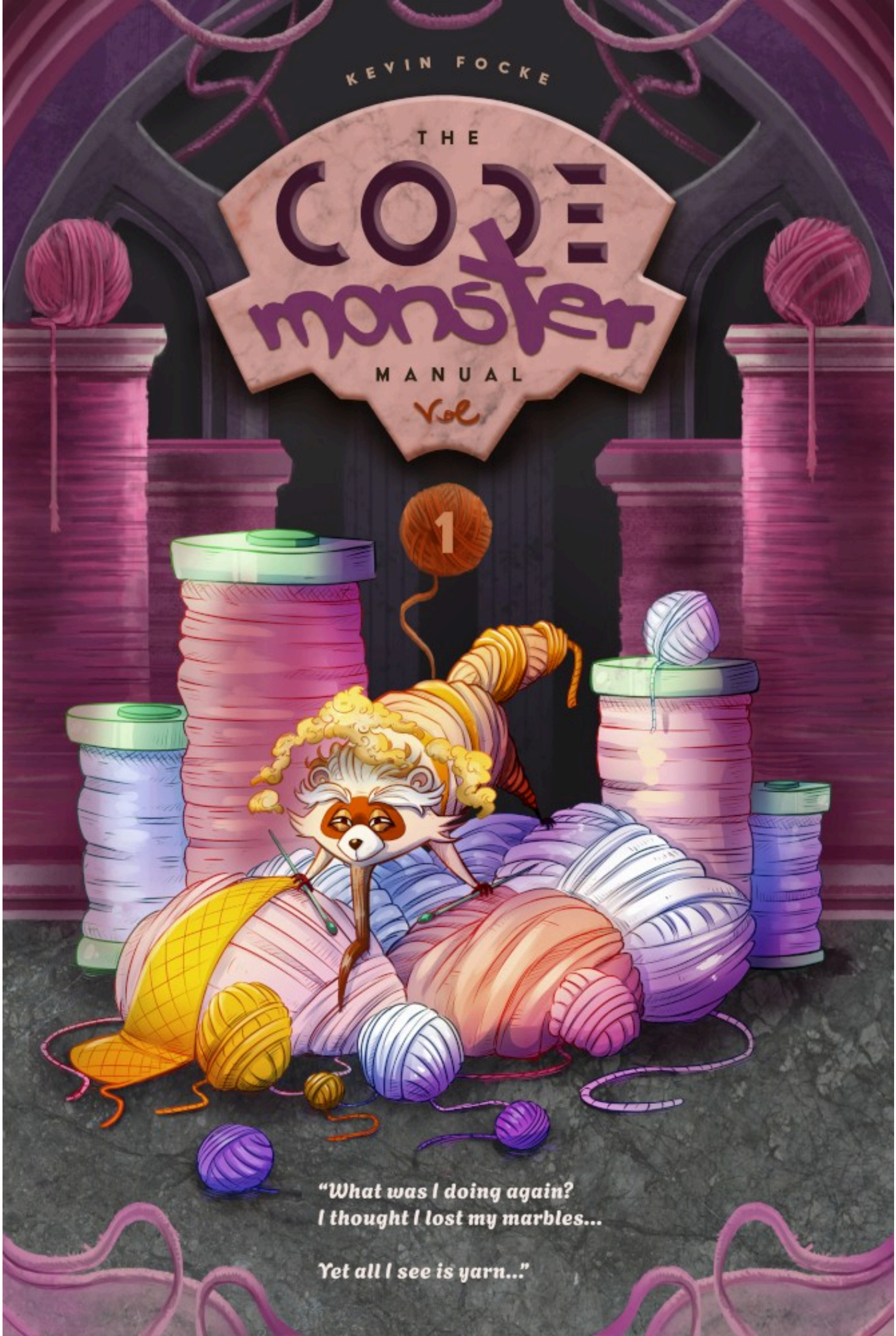

KEVIN FOCKE
THE
CORE
monster
MANUAL
Vol
1
"What was I doing again?
I thought I lost my marbles...
Yet all I see is yarn..."

THE CODE MONSTER MANUAL VOLUME 1 (Sample)

Processors And Tangled Threads

What every programmer should know about processors.

Part Of The Code Monster Manual Trilogy:
- Vol 0: Escape From Number Hell
- Vol 1: Processor And Tangled Threads
- Vol 2: Networks In The Cloud

TABLE OF CONTENTS

CHAPTER 0: Spacechip Transistor

INTRODUCTION

“There are only two hard things in Computer Science: Numbers, processors, off-by-one-errors, data that surprisingly changed since last time you checked, and discovering that you underestimated how many complexities there are in a problem that initially seemed so simple.” - Phil Karlton (heavily abridged)

The processor is the most important component of your computer. After all, it is the component that *computes*—it calculates.

The goal of the book: Make computer processors more approachable & intuitive. When processor security exploits like Spectre happen, you’ll have enough knowledge to understand a brief, intermediate-level explanation.

I hope you feel encouraged by someone you’ve never met. The book series is my earnest attempt to help solve the problems and frustrations I faced, with a bit of fun & charm sprinkled throughout.

This is the second volume in *The Code Monster Manual* series. What did we learn previously?

The main takeaway of Vol. 0 was that computers are versatile because discrete numbers can represent different things. Ambiguity is avoided by using types, even in languages that are not explicitly object-oriented. As a programmer, you cannot always choose to define a custom type, but your programming language certainly has its own types called `primitives`.

For example, an unsigned integer is a common primitive. Some types are made of other types; a `String` contains one or more `Char` units.

And then… There's the mythical magical function—every programmer's favourite abstraction! Functions are optimized for programmer ease-of-understanding, not processor ease-of-running.

A `compiler` translates functions to actually-runnable instructions. Even if you're not hyper-focused on processor performance, it's useful (and in my opinion fascinating) to learn more about how the computer actually works.

Programming languages are fundamentally a tooling product abstracting away the complexity of processors. The general goal of programming is to create something that is simpler to use than the underlying implementation. This works great *most of the time*, but no abstraction is

perfect *every time*.

When the computer acts weird… You're meeting the monsters again. Processors can execute functions in surprising ways that impact performance, compatibility, and security.

I want to linger on a couple points here because it took me a while to fully grasp: A program is a bunch of functions. When a compiler translates functions to a specific processor architecture, the compiler creates a runnable approximation of the functions. Hopefully, nothing gets lost in translation.

For example, your data structure optimizations might not work as expected on different processors—especially if those optimizations rely on unique & quirky processor features. The program might not even run at all on a different processor! Video game consoles are notorious for this; old games are often not compatible with newer hardware. That's kinda odd, isn't? The newer consoles are supposed to be *more* capable!

Bit Of Trickery

When you squeeze more juice out of particular hardware, it tends to make the program less compatible with other systems. (It also tends to increase the maintenance burden.) But, sometimes, that's just what you have to do… The gamers demand more frames per second!

The Exactor strikes again! Different processor architectures don't run the *exact* same program, even if they originate from the *exact* same code. Still, it's a modern miracle how well compilers work. Compilers are so good that, if you encounter a problem in your program, it's probably a human error rather than a compiler error.

Whenever you explain interrelated concepts, you have to choose a simplified starting point. In the previous volume, we focused on numbers and `handwaved away` the complexity of processors. In other words, we chose discrete software numbers as our `abstraction boundary`.

There are always monsters lurking beneath the black curtains of the boundary…

CHAPTER OVERVIEW

Similar to a physical building, the strength of the foundation decides what goes on top. In other words, the capabilities of your processor and the abstraction layers built on top of them, decide the programs you can build.

In this volume, we'll talk about topics where programming & processors clash. We can't fully explore *all* the nuances of processors—that would be too much work. Instead, I want to focus on the most bang-for-your-buck topics.

Bright Idea

If you ever feel overwhelmed, try defining your non-goals. For example, I don't want to talk about electrical circuits nor show you 598 diagrams of logic gates. Would that be useful for processor designers? Sure. Would it help you become a better programmer? Probably not.

Chapter overview:

In Chapter 1, we will learn how to interact with your processor using instructions.

Chapter 2 is focused on functions. What are useful mental models and concepts to know? How are functions converted into machine code?

With a compiler! We'll learn about why compilers exist and the approaches to compilation in Chapter 3.

Then, we'll dive deeper into the principles of computation. I'm quite proud of Chapter 4. It's like a whirlwind tour explaining how we turn the messy ambiguity of the world into the binary predictability of a computer.

In Chapter 5, we will simplify processors using the `thread` abstraction, learn why a `context switch` is expensive, and discuss how to reduce waiting time.

Finally, in Chapter 6, we will optimize processors using speculative execution, learn about pipeline hazards, and briefly explain the Spectre security vulnerability.

CHAPTER 1: Gluey The Glue-Gate

Spirit of the Monster: Processor instructions allow you to control the hardware from the software. In other words, they are the glue-gate between the hardware and software realm. No one is more excited about that fact than Gluey The Glue-Gate.

“I aM Me,
and I iS yoU.
I am computer!
Liminal!

What is my name, says you?
I am Gluey The Glue-Gate?
I say **Woo!**”

KINDS OF PROCESSORS

There are two kinds of processors: General-purpose processors & specialized processors.

General-purpose processors (CPUs) can do pretty much every task, but are very slow at *some* tasks.

Specialized processors (GPUs & ASICs) **cannot do every task**, but are extremely fast at the *specific* tasks they're optimized for.

Overview of abbreviations:

CPU = Computer Processing Unit.

GPU = Graphics Processing Unit.

ASIC = Application-Specific Integrated Circuit.

For the rest of the chapter, we will mainly focus on the CPU. Though notably, many of the same principles apply to more specialized processors.

Let's improve your intuition about why specialized chips perform so much better than general-purpose chips. It involves minimal math.

Imagine we want to calculate this scenario:

I have 3 packs of water. In each pack there are 6 bottles. How many bottles do I have?

```
3 * 6 = ? bottles
```

Okay, we can calculate this result with a general-purpose CPU and a specialized CPU optimized for making this *exact* calculation. How would that look?

Each processor can do *1* operation at a time. We'll ignore LOAD and STORE instructions.

The general-purpose CPU only has an operation to ADD two numbers. It can still calculate the result by converting the multiplication into a sum.

```
3 * 6 = 3 + 3 + 3 + 3 + 3 + 3 = ?
```

Instructions of the general-purpose CPU:

```
- ADD 3  +  3
- ADD 6  +  3    # 6 was the result from the previous calculation
- ADD 9  +  3    # 9 was the result from the previous calculation
- ADD 12 +  3   # 12 was the result from the previous calculation
- ADD 15 +  3   # 15 was the result from the previous calculation
```

It takes 5 instruction to calculate the result. How does that look on our specialized CPU?

Instructions of the specialized CPU:

```
MULTIPLY 3 * 6
```

In our fictional example, the specialized CPU can calculate the result of 3 * ANY NUMBER, extremely fast. It takes only 1 instruction on the specialized CPU.

The big caveat: You must be able to calculate the result by multiplying any number with 3. Let’s name our MULTIPLY instruction accordingly:

```
MULTIPLY3WITH 6
```

New scenario:

I have *4* packs of water. In each pack there are *5* bottles. How many bottles do I have?

```
4 * 5 = ?
```

The general-purpose CPU can calculate the result in this way:

```
4 * 5 = 4 + 4 + 4 + 4 + 4 = ?
```

Instructions of the general-purpose CPU:

```
- ADD 4  +  4
- ADD 8  +  4    # 8 was the result from the previous calculation
- ADD 12 +  4    # 12 was the result from the previous calculation
- ADD 16 +  4    # 16 was the result from the previous calculation
```

It takes 4 instructions to calculate the new scenario with the general-purpose CPU.

By contrast, the specialized CPU **cannot** calculate the result. We can't multiply any integer number with 3 to get the same result.

That's why computers used by regular people always have a general-purpose CPU. You'd rather calculate a result *slowly*, than have no result

at all.

The example is somewhat contrived, but the goal is to build your intuition. For a more accurate explanation, I could say that GPUs are usually optimized for matrix multiplication to represent vector calculations. Alright, fine, it would have made me sound smart, but would it have improved your understanding more? Probably not. The point of communication is to be understood, not to stroke your own ego (unless you're an academic looking for tenure). I love the beauty of words, but do my best to explain complex problems in simple-yet-nuanced ways. Say "nay" to grandiloquence!

Bright Idea

If you don't need a complex mental model, use a simpler model. For example, in chemistry, the simplified atomic model is *good enough* for the layman application of weight measurement; Bohr's model representing atoms as a nucleus with electrons spinning around it can adequately explain the emission spectrums of Hydrogen, but it struggles to explain the energy levels of multi-electron atoms. A non-pertinent consideration when measuring mass-based properties emerging from the underlying chemical configuration. If you require macro measurement with low precision to create a delicious fruit salad, you can just measure the weight of that banana with a kitchen scale! Keep it simple, life is already complicated enough!

In fact, **most users do not want to know how the internals of their products work, they just want that the product works**.

Notably, there is a special kind of processor where the *operations* can be changed. This is called a `field-programmable gate array` (FPGA).

It’s a marvelous piece of technology if you need that flexibility. Speed-wise it’s slower at specialized tasks than ASICs, but comparable in speed (or faster) at some tasks than CPUs.

Adam Savage and Jamie Hyneman did one my favourite demonstrations for how a GPU works.

A CPU is like firing one paintball per second.

A GPU is like firing a cannon of 50 paintballs after 20 seconds.

It really begs the question: Would you rather have 20 paintballs with a gradual feeling of scenic progression or an explosion of 50 paintballs after 20 seconds?

It’s possible to combine multiple kinds of processors on the same chip.

For example, two general-purpose CPUs can be

placed on the same chipboard, with one CPU having a bigger L3 cache than the other.

Counter-intuitively, if the workload can fully run on a single CPU with more L3 cache, it might be better to temporarily disable the other CPU (called `parking the CPU`) rather than sharing the workload across 2 CPUs. Why?

When you use multiple CPUs, you have to coordinate between them. Communication overhead is a well-known computing problem for both machines *and humans*. In the classic book *The Mythical Man-Month*, Fred Brooks argues that adding more programmers to a project can make that project finish *later*. Similarly, adding more CPUs can also make your program *slower*. Counter-intuitive, yet true in practice.

Bright Idea

Compare the performance of your applications using real-world benchmarks. The results are often surprising. Shakespeare did not say: ‘To park the CPU or not to park the CPU, that *is* the question.’ But maybe he should have.

INSTRUCTIONS, OPERATORS & OPERANDS


MELENCOLIA
16 3 2 13
5 10 11 8
9 6 7 12
4 15 14 1

When you play with a deck of cards, your main concern is *not* the thickness of the paper (unless you are a psychopath.) Rather, you care more about how to use the cards.

Just provide the instructions and the processor will figure out how to run it. Programmers (mostly) don't need to worry about task scheduling & caching…

Processor instructions are composed of an `operator` and one or more `operands`.

An operator decides an action. The operands are the things you apply the actions on.

For example, we want to format an instruction that will calculate two numbers:

1 + 3 = ?

The operator is `ADD`, the operands are `1` and `3`. How can we apply the addition operation?

Well, we need to describe every step *exactly*. As a prerequisite, we need a place to store the numbers in the CPU. The storage of the CPU is

called a `register`. It’s a tiny, extremely-high-performance storage location.

In our simplified example, we assume that all actions are permitted on the processor (that would be a security nightmare on a real computer).

Our simple processor has 4 registers:

- Register 0; 6 bits used for storing the currently processing instruction.

- Register 1; 4 bits used for storing data.

- Register 2; 4 bits used for storing data.

- Register 3; 4 bits used for storing the output of the ADD instruction.

We also have a hard disk. In practice, the hard disk is relatively slow with a huge memory capacity. However, to keep our explanation simple, we will pretend the hard disk has a small capacity.

The hard disk has 4 addressable memory locations containing 4-bit numbers:

Hard drive state:

Memory address	Contains Binary Number	Number As Decimal
00	0001	1
01	0011	3
10	0000	0
11	0000	0

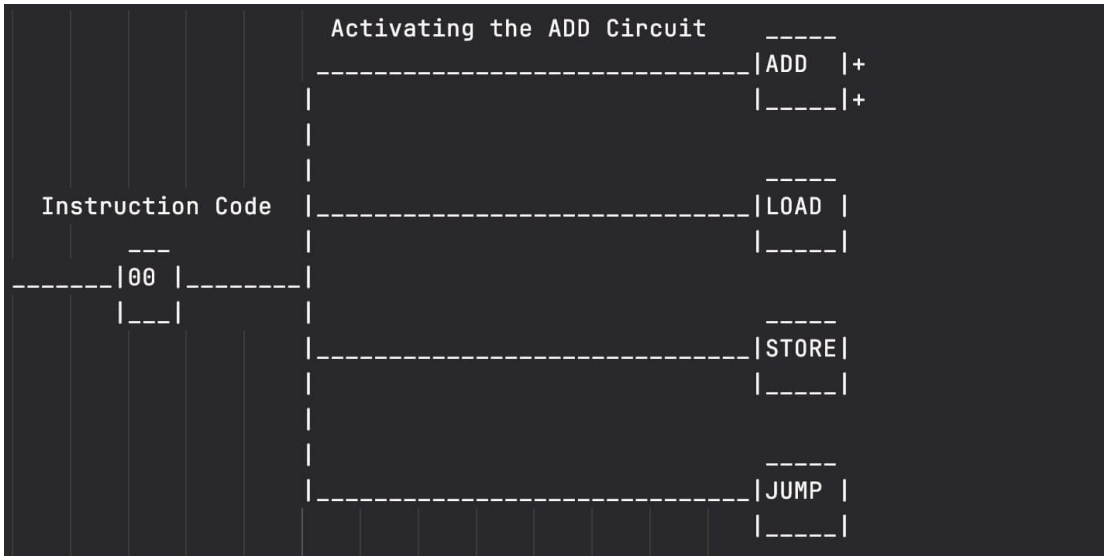
Notably, the memory addresses of the hard drive already contain data.

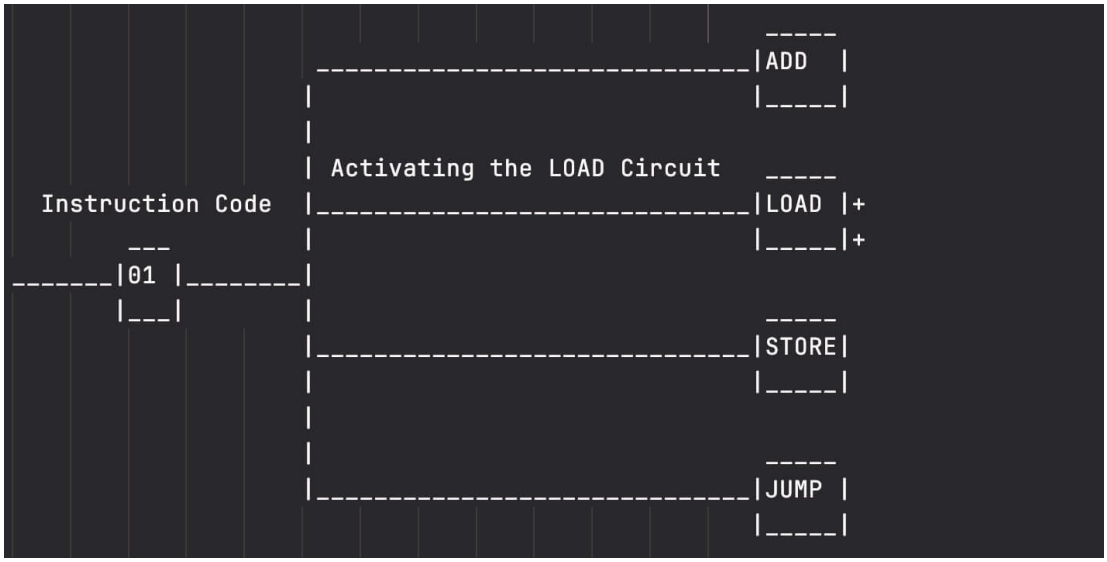
As we saw in the previous volume, EVERYTHING in a computer is a discrete binary number interpreted in different ways, including the operations. Our simple processor has 4 possible operations. Each operation is thus represented by a 2-bit operation code (OpCode):

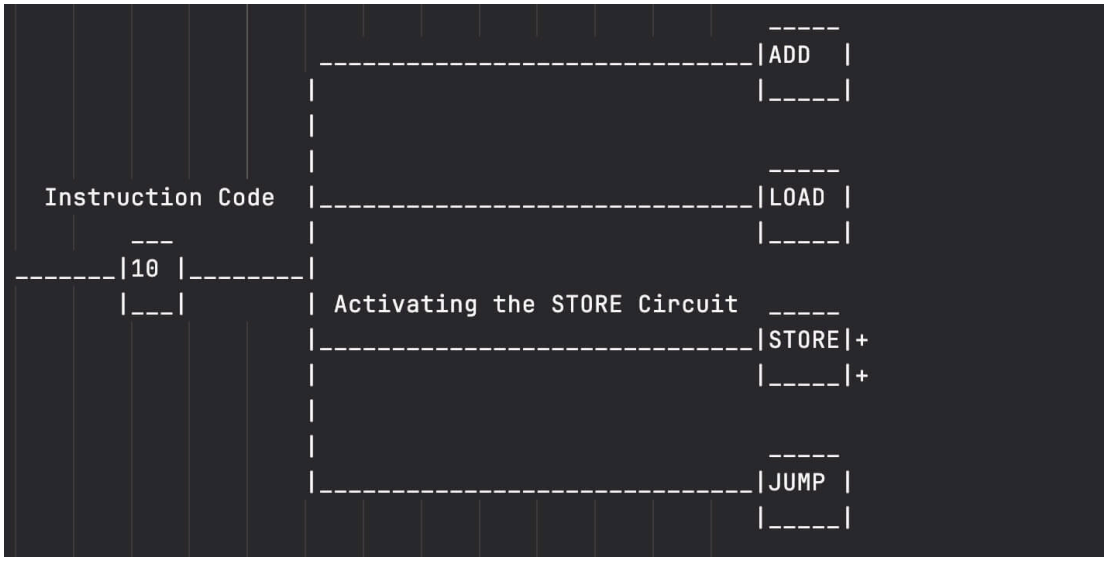
Processor operations:

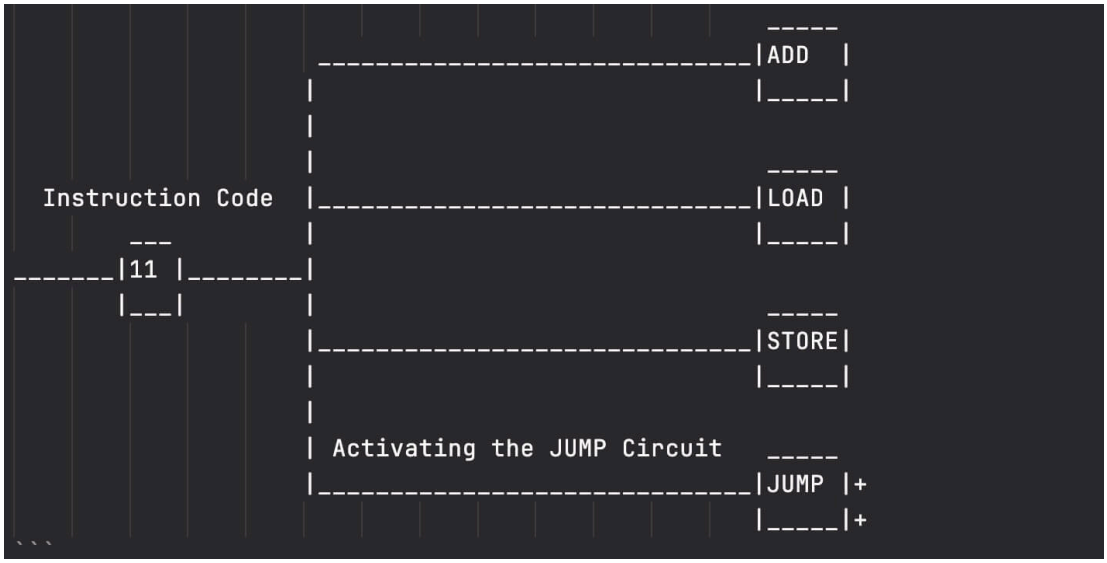
OpCode	Operation	What It Does
00	ADD	Add two numbers and store the result in register 3
01	LOAD	Load a number from a disk into a data register
10	STORE	Store a number from a data register to a disk
11	JUMP	Change the instruction counter

Whenever the processor encounters an OpCode, it will activate a different electrical circuit.


Activating the ADD Circuit
ADD
LOAD
STORE
JUMP
Instruction Code
00


ADD
Activating the LOAD Circuit
LOAD
STORE
JUMP
Instruction Code
01





Now, let’s create our addition, step by step. First, we need to load two numbers into their

separate registers:

- LOAD The decimal number `1` from the hard drive binary memory address `00` INTO Register 1

- LOAD The decimal number `3` from the hard drive binary memory address `01` INTO Register 2

Then, we can calculate the result:

- ADD the numbers in Register 1 and Register 2. Temporarily store the result in Register 3.

Finally, we want to write the result back to the hard drive:

- STORE the number from Register 3 to hard drive binary memory address `10`

How can we translate these steps to 6-bit instructions?

The operation code (OpCode) will be represented by the first two bits. The remaining bits represent *something else* (`Operand` aka `DataCode`). The semantic meaning of the Operand/DataCode depends on the OpCode.

Bit Of Trickery

Strictly speaking, OpCodes and DataCodes are all just numbers. Yet, by giving the segments of the instruction different names, we're able to distinguish the parts that make sense in a context. The context is provided by the `instruction set`. The instruction set is a specification describing how the instructions should be formatted and used (similar to a character set, as discussed in the previous volume).

For explanation purposes, I will separate each meaningful bundle of bits in the instruction with spaces.

Here are the previous steps translated to instructions:

Instruction	Opcode	Operation	DataCode
01 0 00 0	01	LOAD	0 00 0
01 1 01 0	01	LOAD	1 01 0
00 0000	00	ADD	0000
10 10 00	10	STORE	10 00

Here's the starting state of the registers and the hard drive:

Register state:

Register	Binary Number	Last Changed By	What does it store?
0	000000	Initial state	Currently running instruction
1	0000	Initial state	A loaded number
2	0000	Initial state	A loaded number
3	0000	Initial state	The result of an ADD instruction

Hard drive state:

Memory address	Contains Binary Number	Number As Decimal	Last Changed By
00	0001	1	Set by user
01	0011	3	Set by user
10	0000	0	Initial state
11	0000	0	Initial state

We'll apply this instruction:

Instruction	Opcode	Operation	DataCode
01 0 00 0	01	LOAD	0 00 0

The first two bits `01` indicate a LOAD operation. The third bit `0` indicates the destination register. There are two possible registers we can load to: Register 1 and Register 2. Register 1 is represented by the third bit `0`, Register 2 is represented by the third bit `1`.

Thus, because the third bit is `0` we are

loading some number to Register 1. Which number? The number stored at the hard drive memory address `00`. What is that binary number? It's `0001` which is the decimal number `1`. We have loaded our first number into Register 1! The last bit of the instruction does not get used by the LOAD operation. Here's the processor and hard drive state:

Register	Binary Number	Last Changed By	What does it store?	
0	01 0 00 0	Instruction Counter*	Currently running instruction	< Changed
1	0001	LOAD operation	A loaded number	< Changed
2	0000	Initial state	A loaded number	
3	0000	Initial state	The result of an ADD instruction	

Hard drive state:

Memory address	Contains Binary Number	Number As Decimal	Last Changed By
00	0001	1	Set by user
01	0011	3	Set by user
10	0000	0	Initial state
11	0000	0	Initial state

We will learn about the mysterious Instruction Counter soon. There were no changes to the hard drive state. Next instruction:

Instruction	Opcode	Operation	DataCode
01 1 01 0	01	LOAD	1 01 0

The first two bits `01` indicate a LOAD operation. The third bit `1` indicates we are loading some number to Register 2. Which number? The number stored at the hard drive memory address `01`. What is that number? It's `0011` which is the decimal number 3. We have loaded our second number into Register 2! The last bit of the instruction does not get used by the LOAD operation. The register state of the processor changed again.

Register state:

Register	Binary Number	Last Changed By	What does it store?	
0	01 1 01 0	Instruction Counter	Currently running instruction	< Changed
1	0001	LOAD operation	A loaded number	
2	0011	LOAD operation	A loaded number	< Changed
3	0000	Initial state	The result of an ADD instruction	

Hard drive state:

Memory address	Contains Binary Number	Number As Decimal	Last Changed By
00	0001	1	Set by user
01	0011	3	Set by user
10	0000	0	Initial state
11	0000	0	Initial state

Tea break. Alright. Next instruction:

Instruction	Opcode	Operation	DataCode
00 0000	00	ADD	0000

The first two bits `00` indicate an ADD operation. The ADD operation adds the numbers stored in Register 1 and Register 2. The result is stored in Register 3. The remaining bits of the instruction do not get used by the ADD operation.

Register state:

Register	Binary Number	Last Changed By	What does it store?	
0	00 0000	Instruction Counter	Currently running instruction	< Changed
1	0001	LOAD operation	A loaded number	
2	0011	LOAD operation	A loaded number	
3	0100	ADD operation	The result of an ADD instruction	< Changed

Hard drive state:

Memory address	Contains Binary Number	Number As Decimal	Last Changed By
00	0001	1	Set by user
01	0011	3	Set by user
10	0000	0	Initial state
11	0000	0	Initial state

Notably, all of the changes happened only in the processor register; the hard drive did NOT change. Finally, we persistently store the calculated result to the hard disk with the instruction:

Instruction	Opcode	Operation	DataCode
10 10 00	10	STORE	10 00

The first two bits `10` indicate a STORE operation. The STORE operation writes the number of Register 3 to a hard drive memory address. What is the number of Register 3? It was the result of the ADD instruction (4). What is the drive memory address? The address is `10`. The remaining bits of the instruction are not used by the STORE operation.

```
Register state:

| Register | Binary Number | Last Changed By     | What does it store?              |
| -------- | ------------- | ------------------- | -------------------------------- |
| 0        | 000000        | Instruction Counter | Currently running instruction    | < Changed
| 1        | 0001          | LOAD operation      | A loaded number                  |
| 2        | 0011          | LOAD operation      | A loaded number                  |
| 3        | 0100          | ADD operation       | The result of an ADD instruction |
```

```
Hard drive state:

| Memory address | Contains Binary Number | Number As Decimal | Last Changed By     |
| -------------- | ---------------------- | ----------------- | ------------------- |
| 00             | 0001                   | 1                 | Set by user         |
| 01             | 0011                   | 3                 | Set by user         |
| 10             | 0100                   | 4                 | CPU STORE operation | < Changed
| 11             | 0000                   | 0                 | Initial state       |
```

We have calculated `3+1` using our processor. Hurray! However, we still handwaved away some complexity; how do we decide *which* instruction currently runs on the processor?

INSTRUCTION COUNTER AND OPTIMIZATION

To enable programming, you must be able to store multiple instructions AND change the instruction you are currently processing.

Disclaimer: CPUs can have multiple cores. Each CPU core processes one instruction at a time. The duration of each instruction is controlled by the system clock. For now, we will mostly ignore the complexities of the system clock—timing is a complicated theme of the next volume. Also, for simplicity, we will pretend our CPU only has a single core. Yeah, it's… complicated. But we can learn it all, step by step!

The Instruction Counter (IC) decides which instruction is currently being processed. When the computer is powered on, the IC starts at

index `000`. After the current instruction is processed, the processor automatically increments the IC index by `1`, unless the current instruction is a JUMP operation. Instructions that impact the IC are called `control flow` instructions.

Let’s imagine our simple computer can store 8 instructions. The previous instructions for calculating (1+3) have been set. The rest of the instructions are initialized to all zeros.

The Instruction Counter (IC) starts at index `000`. The CPU reads the OpCode of the instruction stored at that index. The OpCode is NOT `11` (JUMP), thus the current instruction (LOAD) is processed and the IC increments by 1.

Let’s escape the number hell of instructions. For readability, I have translated the raw binary numbers of the instruction into verbose `assembly`.

Abbreviations:

- reg1 = Register 1 Of The CPU.
- drive00 = Hard Drive Memory Address 00

```
| IC (Binary) | IC (Decimal) | Instruction | Assembly                    |
| ----------- | ------------ | ----------- | --------------------------- |
| 000         | 0            | 01 0 00 0   | LOAD TO reg1 FROM drive00   | < Currently processing
| 001         | 1            | 01 1 01 0   | LOAD TO reg2 FROM drive01   |
| 010         | 2            | 00 0000     | ADD reg1 reg2 STORE TO reg3 |
| 011         | 3            | 10 10 10    | STORE FROM reg3 TO drive10  |
| 100         | 4            | 000000      | ADD reg1 reg2 STORE TO reg3 |
| 101         | 5            | 000000      | ADD reg1 reg2 STORE TO reg3 |
| 110         | 6            | 000000      | ADD reg1 reg2 STORE TO reg3 |
| 111         | 7            | 000000      | ADD reg1 reg2 STORE TO reg3 |
```

Did you notice? The assembly language is worded based on the ordering of the instructions.

As a side note, from a user experience perspective, it might be more convenient if the assembly was worded like this:

```
| IC (Binary) | IC (Decimal) | Instruction | Assembly                    |
| ----------- | ------------ | ----------- | --------------------------- |
| 000         | 0            | 01 0 00 0   | LOAD FROM drive00 TO reg1   |
| 001         | 1            | 01 1 01 0   | LOAD FROM drive01 TO reg2   |
| 010         | 2            | 00 0000     | ADD reg1 reg2               | < Result implicitly stored in register 3
| 011         | 3            | 10 10 10    | STORE FROM reg3 TO drive10  |
| 100         | 4            | 000000      | ADD reg1 reg2               |
| 101         | 5            | 000000      | ADD reg1 reg2               |
| 110         | 6            | 000000      | ADD reg1 reg2               |
| 111         | 7            | 000000      | ADD reg1 reg2               |
```

But then… The gap between the abstraction and implementation would widen. As a processor designer, you could also change which bits represent the OpCode and DataCode, but that would require a change to the instruction set and to the circuit design. Furthermore, old program instructions would no longer work on the new instruction set. Anyhow, these are all considerations that you don’t have to worry

about, but that have an impact on the programmer user experience.

Let’s look back at our code execution. The instruction counter automatically jumps by 1 (aka increments), to the next step. The CPU reads the OpCode of the instruction stored at that index. If the OpCode is NOT `11` (JUMP), the current instruction (LOAD) is processed and the IC increments by 1.

IC (Binary)	IC (Decimal)	Instruction	Assembly	
000	0	01 0 00 0	LOAD TO reg1 FROM drive00	
001	1	01 1 01 0	LOAD TO reg2 FROM drive01	< Currently processing
010	2	00 0000	ADD reg1 reg2 STORE TO reg3	
011	3	10 10 10	STORE FROM reg3 TO drive10	
100	4	000000	ADD reg1 reg2 STORE TO reg3	
101	5	000000	ADD reg1 reg2 STORE TO reg3	
110	6	000000	ADD reg1 reg2 STORE TO reg3	
111	7	000000	ADD reg1 reg2 STORE TO reg3	

The same process repeats. Again.

IC (Binary)	IC (Decimal)	Instruction	Assembly	
000	0	01 0 00 0	LOAD TO reg1 FROM drive00	
001	1	01 1 01 0	LOAD TO reg2 FROM drive01	
010	2	00 0000	ADD reg1 reg2 STORE TO reg3	< Currently processing
011	3	10 10 10	STORE FROM reg3 TO drive10	
100	4	000000	ADD reg1 reg2 STORE TO reg3	
101	5	000000	ADD reg1 reg2 STORE TO reg3	
110	6	000000	ADD reg1 reg2 STORE TO reg3	
111	7	000000	ADD reg1 reg2 STORE TO reg3	

And again.

IC (Binary)	IC (Decimal)	Instruction	Assembly
000	0	01 0 00 0	LOAD TO reg1 FROM drive00
001	1	01 1 01 0	LOAD TO reg2 FROM drive01
010	2	00 0000	ADD reg1 reg2 STORE TO reg3
011	3	10 10 10	STORE FROM reg3 TO drive10	< Currently processing
100	4	000000	ADD reg1 reg2 STORE TO reg3
101	5	000000	ADD reg1 reg2 STORE TO reg3
110	6	000000	ADD reg1 reg2 STORE TO reg3
111	7	000000	ADD reg1 reg2 STORE TO reg3

We’ve now calculated the result of (1+3) and stored it to the hard drive.

Unfortunately, the Instruction Counter does not know when the desired outcome is achieved. Thus, it will keep going…

IC (Binary)	IC (Decimal)	Instruction	Assembly
000	0	01 0 00 0	LOAD TO reg1 FROM drive00
001	1	01 1 01 0	LOAD TO reg2 FROM drive01
010	2	00 0000	ADD reg1 reg2 STORE TO reg3
011	3	10 10 10	STORE FROM reg3 TO drive10
100	4	000000	ADD reg1 reg2 STORE TO reg3	< Currently processing
101	5	000000	ADD reg1 reg2 STORE TO reg3
110	6	000000	ADD reg1 reg2 STORE TO reg3
111	7	000000	ADD reg1 reg2 STORE TO reg3

IC (Binary)	IC (Decimal)	Instruction	Assembly
000	0	01 0 00 0	LOAD TO reg1 FROM drive00
001	1	01 1 01 0	LOAD TO reg2 FROM drive01
010	2	00 0000	ADD reg1 reg2 STORE TO reg3
011	3	10 10 10	STORE FROM reg3 TO drive10
100	4	000000	ADD reg1 reg2 STORE TO reg3
101	5	000000	ADD reg1 reg2 STORE TO reg3	< Currently processing
110	6	000000	ADD reg1 reg2 STORE TO reg3
111	7	000000	ADD reg1 reg2 STORE TO reg3

IC (Binary)	IC (Decimal)	Instruction	Assembly	
000	0	01 0 00 0	LOAD TO reg1 FROM drive00	
001	1	01 1 01 0	LOAD TO reg2 FROM drive01	
010	2	00 0000	ADD reg1 reg2 STORE TO reg3	
011	3	10 10 10	STORE FROM reg3 TO drive10	
100	4	000000	ADD reg1 reg2 STORE TO reg3	
101	5	000000	ADD reg1 reg2 STORE TO reg3	
110	6	000000	ADD reg1 reg2 STORE TO reg3	< Currently processing
111	7	000000	ADD reg1 reg2 STORE TO reg3	

IC (Binary)	IC (Decimal)	Instruction	Assembly	
000	0	01 0 00 0	LOAD TO reg1 FROM drive00	
001	1	01 1 01 0	LOAD TO reg2 FROM drive01	
010	2	00 0000	ADD reg1 reg2 STORE TO reg3	
011	3	10 10 10	STORE FROM reg3 TO drive10	
100	4	000000	ADD reg1 reg2 STORE TO reg3	
101	5	000000	ADD reg1 reg2 STORE TO reg3	
110	6	000000	ADD reg1 reg2 STORE TO reg3	
111	7	000000	ADD reg1 reg2 STORE TO reg3	< Currently processing

At IC index `111`, the IC will overflow to `000`. That’s no problem in the current scenario.

—- By the way, have you noticed?

The processor keeps calculating the ADD instruction, even though we have finished calculating what we wanted! Instead, we are using the ADD instruction as a NOP—a No-Operation. It’s a funny operation. The intent of a NOP is to *literally calculate nothing*.

The bright news is, we could design the ADD instruction to be more flexible and less error-prone! Currently, only the first two bits of the ADD instruction are used. In our redesigned

instruction set, only a specific DataCode causes the instruction to be interpreted as an ADD instruction. This works like a `checksum`, a very common error-detection technique. A checksum is usually computed from other numbers, but we can also use it as a predefined password.

Imagine that OpCode 00 is only interpreted as an ADD instruction if it matches the DataCode `0101`.

Instruction	OpCode	Operation	DataCode	Checksum Matches?
00 0000	00	NOP	0000	No. It works like a NOP.
00 0001	00	NOP	0001	No. It works like a NOP.
00 0101	00	ADD	0101	Yes. Calculate!
00 0001	00	NOP	1001	No. It works like a NOP.

Sure, we need to rewire some circuits, but we still use the *exact* same amount of bits for the instructions—with more flexibility! Whenever there are unused bits, someone will find a use for it!

Same OpCode, different operation because of the DataCode. Alternatively, you can view an instruction starting with `00` as a 6-bit OpCode. This is not a theoretical exercise; `variable OpCode lengths` are used in practice to improve space efficiency.

The flexible interpretation of numbers is a difficult skill to learn, but packaging more

information in a smaller bundle is valuable, especially when you have to transfer data between computers. Less data to send means more money to spend on overpriced GPUs!

Rewriting the instructions:

IC (Binary)	IC (Decimal)	Instruction	Assembly
000	0	01 0 00 0	LOAD TO reg1 FROM drive00
001	1	01 1 01 0	LOAD TO reg2 FROM drive01
010	2	00 0101	ADD reg1 reg2 STORE TO reg3
011	3	10 10 10	STORE FROM reg3 TO drive10
100	4	00 0000	NOP
101	5	00 0000	NOP
110	6	00 0000	NOP
111	7	00 0000	NOP

This is complicated material, allow yourself some time for a little break. You deserve it.

Only the wicked do not rest! Us mere mortals, we *siesta*!

OVERCOMING HARDWARE LIMITATIONS

In the previous example, I intentionally created a processor where not every DataCode bit was used.

Bright Idea

Always consider the trade-off between efficiency and future flexibility; perhaps there are use cases for the unused bits you haven't thought of *yet*. Computers are complex monsters. Sooner or later, they will surprise you, **guaranteed**. Why back yourself into a future corner?

The world *as it should be* rarely matches with the world *as it is*, the gap between our opportunity for impact.

I started this series because I didn't find the right resources for studying programming. Hopefully, I improved that situation for the 9 of you who bought Vol. 0!

I like to imagine these people as The Benevolent Nine Riders! Thanks for your support! <3

Now, my Benevolent Nine Riders, are you ready to face your toughest challenge yet?

Enjoyed this sample?

Get the full book at

kevinfocke.com