1: Introduction

1.1: What is a computer?

Summary: A computer is fundamentally any machine that can perform a small number of very simple operations on numbers. It turns out that a machine that can perform these operations extremely quickly is all it takes to enable the many complex uses of computers that we experience every day.

Definitions: computer

The future is now. You can speak and text with friends on the other side of the world, fly on an airplane to visit them, play games that are so realistic you can pass off screenshots as photographs, and browse the internet with a device that fits in your pocket. What is it precisely that cell phones, the internet, game consoles, and air-traffic control systems all have in common? Certainly they are computer-based systems, but the specific thing that they are all fundamentally doing, as we will come to understand, is surprisingly banal:

A **computer** is simply any system that is capable of four things:

- **Arithmetic:** Basic arithmetic and logical operations on numbers
- **Storage:** Storing the results of these operations
- **Branching:** Choosing which operations to perform next, based on the result of another operation
- **Input/Output (often written "I/O"):** Receiving signals from external sources (input) and sending signals to the outside world (output)

All computers, from the microchip in a thermostat to smartphones, laptops, game-consoles, and the ones that power Google's and Facebook's servers, are at heart nothing more than machines that can do these four things really, really quickly.

We shall spend the rest of this book discussing two questions:

- How, if we have a 'computer' (in our apparently limited sense), we can use it to actually construct some of the complicated systems we associate with computers? We will focus specifically on the examples of search engines, game consoles, and smartphones.
- Once we are convinced that this notion of computer is actually powerful enough to do everything we would expect, how do we actually construct a physical machine that can perform these four kinds of operations?

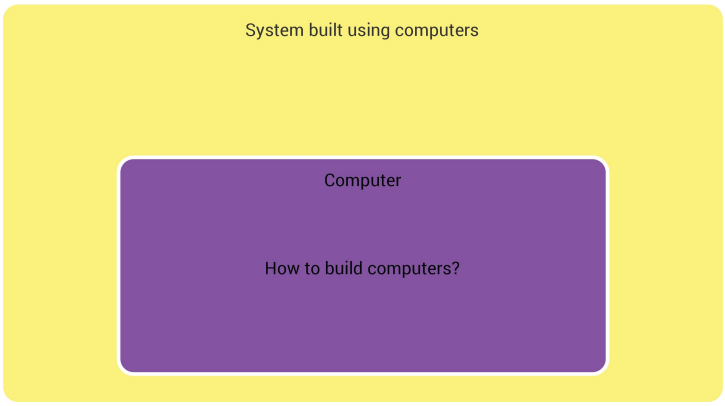

Figure 1.1.1: The two sides of the story of this text.

1.2: Computers through history--Types of computers and what they can do

Summary: A computer can either be wired to perform a fixed specific sequence of arithmetic and storage operations, or it can be wired to accept lists of such operations that it will then perform. When we think of computers in this book, we will be speaking of the latter type. Further, if the operations that the computer can perform are general enough that they can be used to run any algorithm, we call that computer 'Turing-complete'.

Definitions: fixed program, stored program, program, Turing completeness

One of the world's first computers was designed to do one thing only: solve systems of linear equations. This was the Atanasoff-Berry computer (ABC). Brainchild of John Vincent Atanasoff and Clifford Berry around 1937, the ABC would take in equations encoded onto punched paper cards, perform the equation-solving process, store the results of its computations in an electronic memory,

and then output the final result to an external display.

Such were the times. If we had a computation we needed done repeatedly and frequently (be it solving a set of linear equations or computing the correct aiming point for an anti-aircraft gun) we would of course first have to figure out a precise sequence of steps that would solve the problem. Then, because we do not want to execute these steps by hand over and over again, we would design a machine that would take a specific instance of the problem as input (say, an actual system of linear equations), mechanically perform the steps we prescribed for solving it, and then output the result for us to use.

For instance, suppose we need to compute a lot of square roots. Before we can make a machine to do this, we first need a procedure to approximate the square root of any given input by hand. For example, consider the following procedure: In it, we'll have two numbers called *s* and *x*. *x* will be the number whose square root we want to find, and s will be a value that we repeatedly update throughout the procedure until it becomes close enough to the desired output--the square root of x. The steps will be as follows:

1. Start by changing s to be s = x/2.
2. Take whatever s is currently and add the number x/s to it, making that sum the new value of s.
3. Divide s by 2, making the result the new value of s.
4. If s*s is not yet as close as we want it to be to x, return to step 2 and continue from there.

This is an abstract recipe, written for human understanding with no particular machine in mind. Such a recipe will be called an "algorithm". (It may not actually be obvious that this algorithm actually computes the square root of x. In fact it relies on a technique from calculus called 'Newton's method of approximation', but it is not necessary for the moment to understand why it works. If you have a calculator, try using this method to compute the square root of 3 and compare what you get after 4 repetitions with the actual answer of approximately 1.73205.)

Now that we have contrived an algorithm that turns the problem "Compute some square roots" into a sequence of simple arithmetic operations, we want an actual machine to compute some actual square roots so that we don't have to run this procedure by hand. To that end, we can design some separate bits of hardware for storing, multiplying, adding, dividing, and comparing numbers, and hook them up as in the figure below:

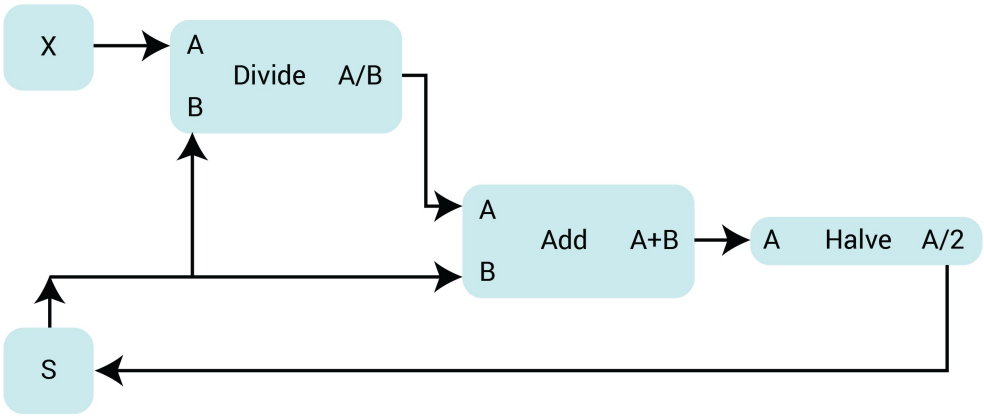

Figure 1.2.1: The square-root-inator: A physical device for computing square roots.

To use this machine, we would store our desired input into x, and store half of that--x/2--into s. Then a single run of the machine will run steps 2 and 3 of the algorithm. When it is finished, we can look at the new value of s, and if s^2 is close enough to x, we can take that s to be the square root. Otherwise, we can simply run the machine again to get an even better value of s, and repeat until we are satisfied.

(As a note, we could make an even more complicated "machine" out of components like these that would perform for us step 4 rather than doing it manually. The reader is invited to think about this at leisure.)

This square root machine, ABC, and other machines like them, are computers, in that they do computations. However, they are single-purpose, or **fixed-program**, devices. That is, they are designed around and can execute only a single algorithm, whose implementation is hard-wired into the device.

The next major step, historically, was a device called the ENIAC. ENIAC, like the ABC and square-root-inator, had the basic hardware for arithmetic operations and storage. But unlike the above, where the sequence in which these arithmetic and storage operations were performed was hard-wired, the owner of the ENIAC was able to tell it what sequence of such operations to perform on its input. Such a machine is called a **stored-program** device, because the user would store the sequence of operations he wanted performed on the input. Any sequence of operations that is stored inside such a device is called a **program**. So when someone with an ENIAC has a problem and has designed an algorithm to solve it, the next step is to craft a program that implements that algorithm. So the ENIAC owner could store the sequence of operations--the program--implementing the above square root algorithm, and suddenly his ENIAC behaves like a square-root-inator. Until he decides to rewrite the program to implement equation solving, and now his ENIAC is an ABC clone.

Beyond just being able to run programs that implemented more than one algorithm, the ENIAC was in a sense profoundly general-purpose: it was in fact the case that absolutely any algorithm could be implemented as an ENIAC program written by a sufficiently clever user. This property of a machine being able to run any algorithm on a single machine is referred to as that machine being **Turing complete**.

> **Aside:** (It bears mention that when we say "algorithm", we refer to procedures that involve manipulations of numbers. The "algorithm" that says simply "shoot down all enemy aircraft" does not specify a precise sequence of calculations, but only an end-goal. It is also clearly outside the scope of a computer that doesn't also come with a gun!)

So we may rephrase our two basic questions simply as: "If we are given a programmable computer, how do we program it?" and "How do we make a programmable computer?" More precisely:

- Given a programmable computer capable of arithmetic and storage operations, how can we program it to do the things we want?
- How can we design a physical machine that can receive a program--i.e. a sequence of arithmetic, storage, branching, and I/O operations--and execute it?

1.3: The big idea -- Abstraction

Summary: To allow us to think about complicated systems without drowning in the complexity, we use a technique called abstraction. To wit: We divide the system into separate subsystems, and we understand the system as a whole by thinking only about how these subsystems interact with each other (i.e., in terms of their "interfaces"). Then, when the need arises, we can isolate a subsystem and consider how it makes its particular interactions happen separately--that is, we consider that subsystem's "implementation".

The key to answering both of our central questions lies in the concept of **abstraction**, also known as **layering** or **black-boxing**. The basic idea is the following: When presented with a problem (e.g., "Create a 3D video game") we do not begin by thinking about every electron that will ever move in one of our system's wires. In fact, we do not think about anything so low-level as inidividual wires at all, at least at first.

A neater approach is to divide the solution into separate functional units, all performing distinct, specified tasks. We first try to understand how to combine these units to solve the problem, pretending all the while that the units simply work as we intend without caring about how they do it--they are "black boxes" into which we need not peek.

When we have a solution built out of black boxes, then all that will remain is to actually make the black boxes. So we then focus on each black box in turn and treat it as a sub-problem. Some black boxes might themselves be complicated enough that we have to solve them using simpler black boxes that will then constitute sub-sub-problems. Eventually, however, all our problems should reduce to sufficiently small sub-problems that we can handle them without punting any further.

We shall proceed with a detailed example in the next section.

1.3.1: Abstraction -- A first example

Summary: We provide an example of a waffle bakery as a complicated system. At a top-level view, the bakery is just a black-box that accepts customer orders and responds with waffles. Peering into this black-box, we see it is comprised of smaller subsystems: A website, a kitchen, and a delivery team.

Definitions: interface, implementation, abstraction

Consider a hypothetical waffle-delivery service--WaffleCo. From the customer's perspective, this consists of a web site he goes to where he types in his phone number, address, credit card info, and desired sort of waffle. Within 30 minutes, he will receive a text message alerting him that the waffle of his dreams is on his doorstep:

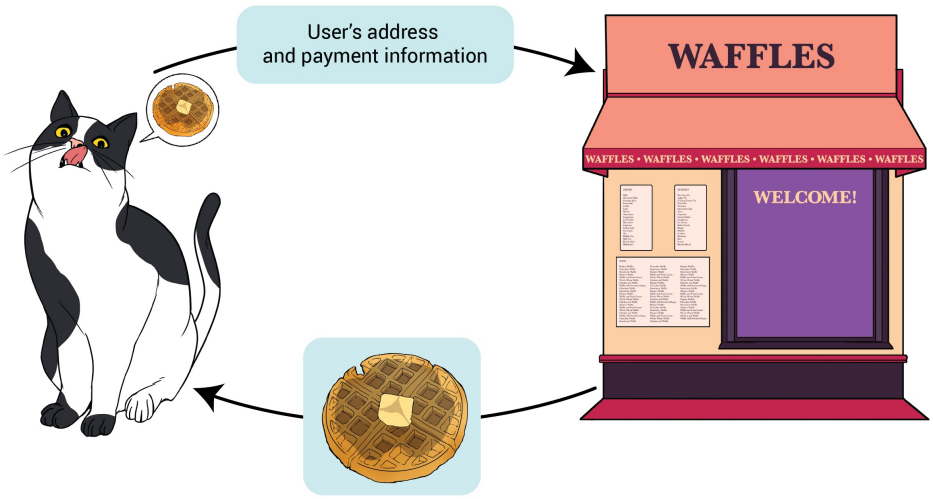

Figure 1.3.1.1: Level-1 view of WaffleCo--the system as a whole

In this picture, the system as a whole is a black box to the user--he needs not understand how the waffle comes to be, let alone how it comes to his location but only how to cause this to happen. The above specification: 'User gives us an address, card number, and waffle selection', is the system's interface. An **interface**, in general, is a description of what inputs to provide to a system, and what outputs will be expected as a result. The actual mechanism that in turn causes the user's inputs (waffle selection, etc.) to be turned into corresponding outputs (delivered waffle) is called the system's **implementation**. So in short, the user cares only about the interface, and nothing about the implementation. This description of the system using only its interface and ignoring its implementation is what is called an **abstraction** of the system--it isn't a complete description of everything that happens in the waffle-making process, but it is a complete description of everything the user needs to know about it.

These were a simple ideas at this top-level view, but let us now see how the concepts of interfaces, implementations, and abstraction let us understand WaffleCo more deeply, but still in a neatly organized way, by peering into the black box one layer deeper:

For starters, there is the website that allows the user to order waffles. The website doesn't itself make the waffles--all it does is provide the mechanism for receiving the user's input. The website then has to forward this request on to another piece of the system that can actually prepare waffles. Say it does this by sending an email to the bakery specifying the chosen waffle and target address.

But then, of course, preparing the waffles isn't enough either--they have to be delivered. So the bakery will send the waffle with the address to the delivery team, who will convey the waffle to the provided address.

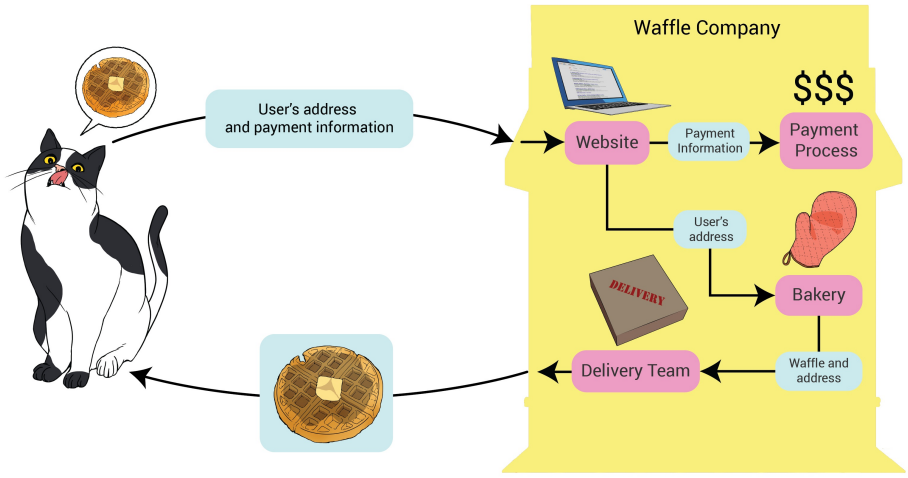

Figure 1.3.1.2: Level-2 view of WaffleCo

Each of these three subsystems is a black box to the others--the website doesn't care how the bakery prepares the waffles. It only needs to know what to include in the email to the bakery, namely the type of waffle to prepare, and the address to send the waffle to. That is, it needs only to know the bakery's interface.

The bakery, in turn, doesn't need to know whether the waffles get delivered by an army of 12-year-olds on bicycles or by GPS-equipped quadcopter drones--that is a matter of the delivery system's implementation--it just needs to know the delivery system's interface, namely, that it needs to box up the waffle in the special company-provided insulated packaging with the destination address printed on top of it.

Thus we have the level-2 view of the system: Earlier, we described the whole system just in terms of its interface--i.e. we described only an abstraction of the system. Now, we've described additionally the implementation details of the system--how it actually processes the requests and makes the waffles, but we did so in terms of smaller abstractions.

This organization has many benefits. To name a couple:

- The various subsystems can be improved or modified independently: Once the company upgrades from 12-year-olds on bicycles to the drones, the bakery won't have to change any procedures if the delivery interface remains the same.
- Specialists can do what they are good at: It will be harder to find someone who can code websites and also organize a bakery to craft excellent waffles than to find two different people for the two separate jobs. If the bakery and website internals are completely separated apart from their use of the agreed-upon interface of emailing the orders, then the website programmer need not think about the bakery's organization to do his job well. He can trust that whatever he does, as long as it sends the user's orders to the bakery in the agreed-upon manner, will work.

As we study the complicated real-world systems built using computers, we will perform a similar layer-by-layer analysis, first describing all the abstractions and then later peering into their implementations to find further abstractions until we get down to the level of something so simple it can be realized as a physical device--the transistor. At that point, since everything else we describe will be built on that one idea, everything else will become as real as that device. And all the innovation brought to bear on improving the implementation of the one fundamental idea will ripple out, allowing smaller and faster computers and computer-based systems.

Warning: All Abstractions Leak

We have just described abstraction and how it allows us to think about complicated systems without getting overwhelmed by complexity. A pithy rephrasing of this idea comes from Gregor Kiczales [1]:

'*[Abstraction] is a primary concept in all engineering disciplines and is, in fact, a basic property of how people approach the world. We simply can't cope with the full complexity of what goes on around us, so we have to find models or approximations that capture the salient features we need to address at a given time, and gloss over issues not of immediate concern.*'

He was describing how we benefit from abstraction, but was also highlighting (as was in fact the main point of his article) the fact that thinking of a subsystem only in terms of its interface is fundamentally a simplification of the world. In particular, it can and often does fail to capture all the features of the subsystem that we, even though working at a different level or on a separate subsystem, may need to consider.

For example, from the abstracted view of WaffleCo, the website designer may think that all he needs to do is get the customer's address and pass it on to the bakery to pass on to the delivery subsystem. But if the delivery subsystem is implemented using 12-year-olds on bicycles, he may wish to code in a way of warning any customers whose address is in a remote suburb that their delivery may take several hours, whereas if the delivery system is quadcopter drones, he may wish to code in a way of warning customers of delays if there is an ongoing thunderstorm. In other words, in practice it is possible the website programmer may have to care a little about the implementation of the delivery subsystem rather than just its interface.

In such situations, we say the abstraction has 'leaked'. Indeed, insofar as every abstraction is letting you hide some of the truth about what happens 'behind the curtain', every abstraction can leak. This doesn't contradict the fact that abstraction is an extremely powerful organizational tool. However, it qualifies the idea we hinted at that, say, if you are only working on the bakery, you don't have to think about the delivery subsystem. Or, more to the point, if you only want to write computer games, you don't have to know how a computer fundamentally works.

This is part of the benefit of what we will be doing in the remainder of this book, namely peeling back the layers of abstraction surrounding the computer and the computer-based systems around us. Even if you only ever want to write software, the abstraction that programming languages provides you is actually a very leaky one. And in the day when one of the abstractions leaks in a critical way, understanding the implementation behind the interface is key. [2]

As we discuss the various layers of abstractions, in this text, we will not tend to dwell excessively on every little way in which they can leak. However, in cases where we do wish to point out a potential leak, that discussion will live in a warning box like this one to separate it from the main body of the text.

1.3.2: Abstraction applied to our two questions

Summary: The layers of abstraction involved in programming computers are the low-level instructions that the computer understands (its instruction-set architecture, or ISA), and the human-friendly high-level programming language that gets translated into ISA instructions by a compiler.

The first layer involved with building a computer is where the computer is broken down into basic functional units--called the computer's microarchitecture. Then these units are expressed in terms of basic components called logic gates. Finally, logic gates are made up of transistors, which are actual physical devices one can build.

Definitions: ISA (basic), programming language (basic), compiler (basic), microarchitecture (basic)

We now turn back to our original questions: How to program a computer, and how to build a computer.

To address the first question, we were supposing the computer already exists and not worrying about how it works internally; in other words, treating it as a black box! What we shall do is build several layers of black-boxes on top of it, making it progressively easier until we get to the system at large that we are trying to build (be it a search engine or whatever). We'll start at the computer and work our way up. As we said, the computer is to us a black box, and so, just like the black boxes of the waffle bakery, the first thing we want to do is to specify its interface.

We've already hinted at the interface by saying a computer was a device that can perform certain arithmetic and logical operations and that can store and retrieve the results of these. The actual interface will consist of a set of operations that the computer is allowed to perform. These operations will vary from computer to computer--in some, 'multiply two numbers' might be a valid operation, and in other more basic computers, only 'add two numbers' is available. All computers come with a precise specification of what operations they understand, called its **Instruction Set Architecture** or **ISA**. This is the computer's interface.

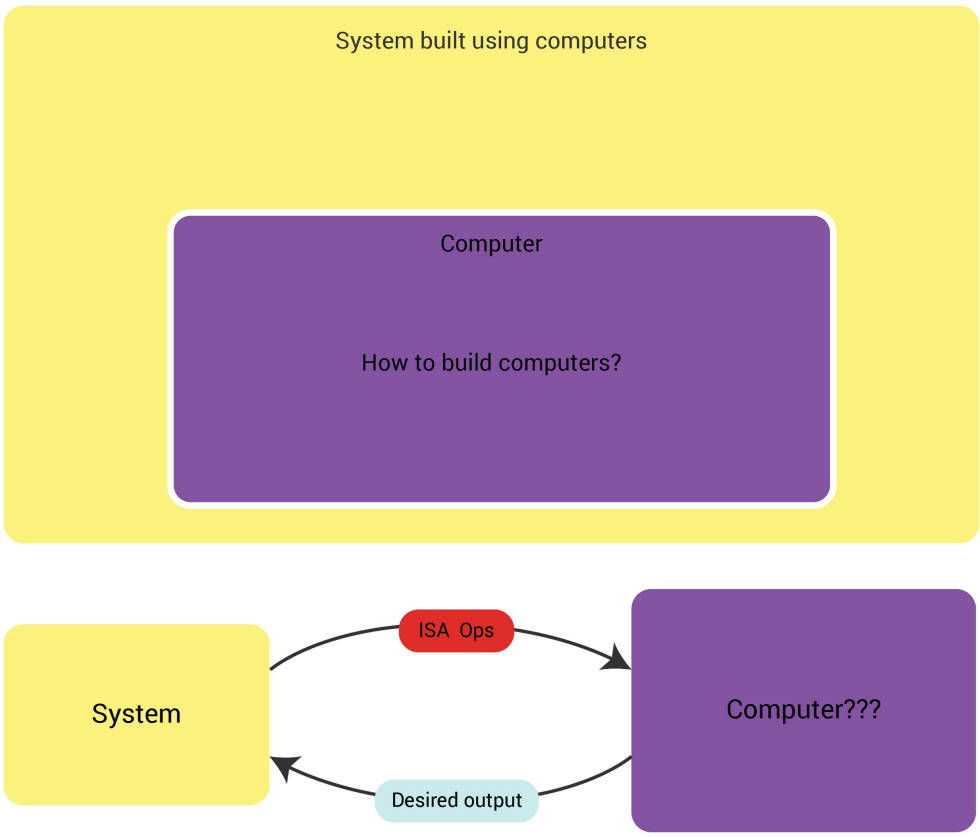

Figure 1.3.2.1: When the system requires a computer as a part of its implementation, the way it talks to the computer to get the desired output is through the ISA.

The ISA is a complete list of the basic operations a computer can perform, but does not represent how people typically program computers in practice. For example, to compute 2+7 using one kind of computer's ISA requires the programmer to write:

```
ldi r30,2
ldi r31,7
add r30,r31
```

Whatever this rather obscure incantation actually means, it certainly looks a lot more cumbersome than we would like. A sane programmer would prefer to be able to simply tell the computer '2+7' and have something automatically figure out the appropriate ISA operations that correspond to this computation (namely, the above). A **programming language** is a more human-friendly way of specifying to the computer what computations to perform. The trouble is that a computer only understands operations from the ISA, so we need an intermediary that can translate the programming language into ISA operations. This is called the **compiler**.

So the programmer, instead of telling the computer directly to execute the complicated ISA operations, writes simpler code in his programming language. The compiler turns his simple-looking program into the appropriate ISA operations, which can then be fed to the computer for execution.

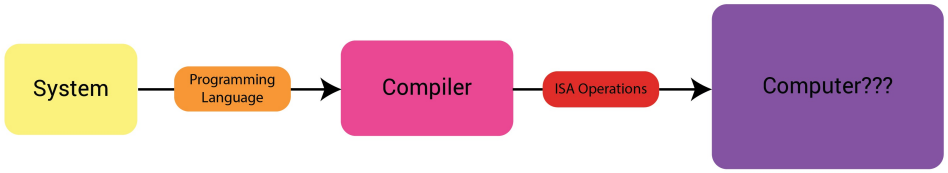

Figure 1.3.2.2: In practice, the system doesn't talk directly to the computer using the complicated ISA, but talks instead using a programming language to a compiler, which turns the programming language commands into ISA operations which it passes on to the actual computer.

Now that the programmer can use the programming language instead of the ISA directly to control the computer, this starts to make possible the complicated tasks needed to create systems like Google and Facebook, as we'll discuss in 1.7 and beyond.

So we have the layers that comprise programming a computer: The ISA layer, and then on top of that, the programming language layer, on top of which the actual system is built.

We now turn to the question of what layers are involved with implementing a computer. This will proceed similarly--we are asking about the implementation of a computer. We shall describe this in terms of large-scale abstractions, and then, having done this, we'll dig down into those abstractions and repeat the process until at the end we get to something physical and concrete that we can actually build.

A computer, by definition, is a physical machine that can accept commands from the ISA and perform the corresponding operations. Typically, these days, such a machine is made out of circuits guiding electron flows, but rather than jumping to this directly, let us take a more gradual descent (much as we didn't immediately start talking about details like oven temperatures and GPS navigation when we wanted to explain the implementation of WaffleCo).

While ultimately the computer will be made of circuits, we start with a high-level organization of these circuits according to their various jobs. This is called the computer's **microarchitecture**. For instance, there should be a circuit that decodes the operations and figures out whether they are performing an addition, a subtraction, a memory retrieval, or something else entirely. Then there should be circuitry for performing additions, and other bits of circuitry for storing and retrieving data. The layout and specification of these high-level components is the microarchitecture. The actual implementation of these components lies one layer lower.

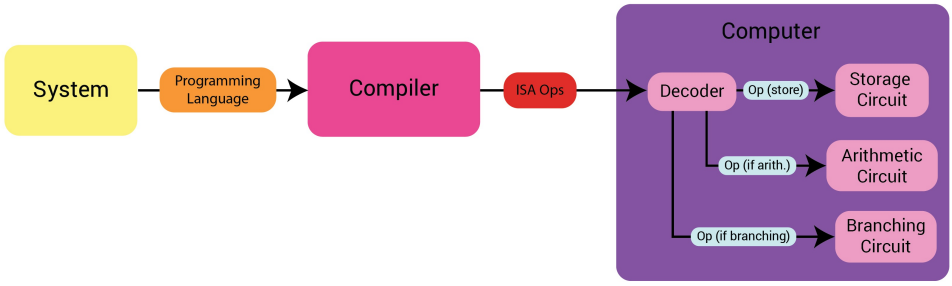

Figure 1.3.2.3: The implementation of the computer involves first figuring out what kind of operation the ISA command corresponds to and passing it along to the appropriate bit of circuitry (I/O operations not pictured for simplicity).

Having set up the appropriate microarchitecture, we will then need to inspect each of the black-box functional units involved and actually put together some circuitry that implements them. We do this, still not in terms of bare wires, but in terms of objects called **logic gates**. Logic gates will still be abstract notions, which in turn will be realized by a single concept--that of a **transistor** which, finally, will be a basic physical device we can actually build using silicon. And since transistors can be used to make logic gates, and logic gates can build the various components of the microarchitecture, we will then be able to actually build a computer.

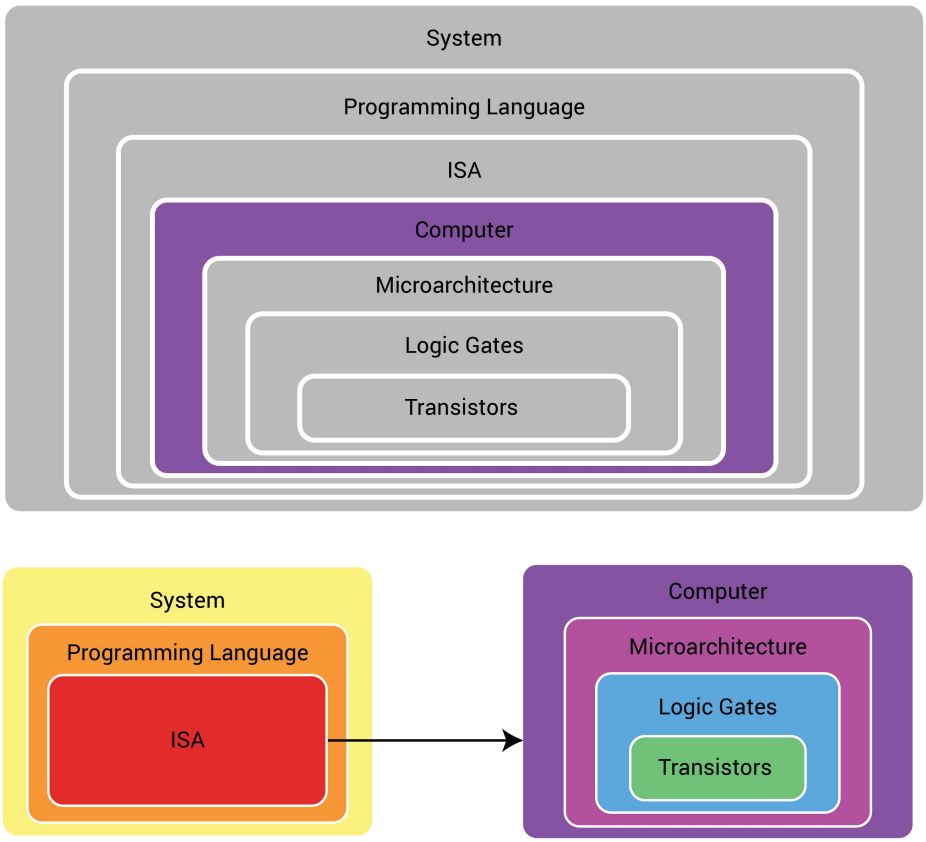

Figure 1.3.2.4: All the layers involved with answering our two questions: To build a system, we program it in the programming language and then this gets translated into the ISA. To build a computer, we start with a microarchitecture, implement that using logic gates, which are in turn made of physical devices called transistors.

1.4: Up the tower: How to program computers

Let us now examine a little more closely the process of going up the tower--that is, the process of, once we have of computer, building a system on top of it.

1.4.1: The ISA layer

Summary: The ISA defines three things:

1. It specifies the basic objects, or "primitives" that the computer contains--often including "registers" (small memories) and a larger data "random-access" memory--collectively known as the "architecture".
2. It specifies the instruction set--that is, the set of things that can be done to these primitives (e.g., "add the numbers in these two registers and store the result in this third register" might be a valid instruction in the instruction sets of some types of computer.
3. It specifies a technique for storing instructions as physical objects. Computers are, after all, machines. They don't deal in abstract commands, but in moving around physical objects (electrons, photons, pineapples, whatever).

Definitions: architecture (basic), registers (basic), instruction set (basic)

The ISA layer comes in two parts: The computer's **architecture** and the **instruction set**. The idea is that before we can specify what instructions the computer can perform, we have to specify what sorts of basic physical components (or "primitives") it has at its disposal. Specifying this gives us the computer's architecture. For example, a part of the architecture of an AVR computer is that it has 32 memory slots, called **registers**, that can only store numbers 0-255. The registers are called r0, r1, ..., r31. The architecture includes other elements as well, which we will discuss later.

Once we have described the architecture, then we can describe the actual instructions that the computer understands--its **instruction set**--by explaining what the instructions are and what they do to the various components that make up the computer's architecture. For example, there will be an instruction to actually store numbers in the registers, called ldi. So to get the value 90 stored in

register r18, we use the instruction:

```
ldi r18,90
```

(Read as 'load into register r18 the value 90')

Another part of the instruction set is an instruction that can add one register to another. This instruction is called (perspicuously enough) add. So, to add r19 to r18, we can just do

```
add r18,r19
```

(Read as 'add to register r18 the value in register r19')

You should now be able to follow our previous example of ISA operations to compute 7+2:

```
ldi r30,2
ldi r31,7
add r30,r31
```

Of course, there are many arbitrary-looking features of this description: Why 32 registers? Why can they only store numbers 0-255? What if I want to add or deal with larger numbers? What if I want to store more than 32 values? These questions will be answered in chapters 4 and 5 when we talk about the ISA layer in detail.

1.4.2: The programming language layer

Summary: Because the ISA deals in very basic primitives, programmers often use a programming language, which allows them to write more human-friendly code which will then be translated into instructions from the actual ISA by a compiler. A programming language has less rigid rules for what valid code can be than the ISA, but there are still rules--called the language's syntax. The meaning of various valid constructions using the programming language--what those lines of code actually do--is also a part of the programming language's definition--called the language's semantics.

Definitions: syntax (basic), semantics (basic), library (basic)

As we mentioned, the ISA layer is not the one that programmers actually deal with most frequently, not least because it takes three lines of code just to compute 7+2! So a programmer will instead use a programming language, which allows them to perform this operation by writing literally "7+2". In general, the programming language is more human-friendly than the ISA. Since the computer only understands the ISA, a compiler will take the more human-readable code and will turn it into ISA operations that the computer can actually understand.

But a compiler doesn't just understand any human-readable text and turn it into ISA operations. For instance, if instead of writing "7+2" a programmer writes "Add seven and two" or "cook some waffles", the compiler will not know what this means. A programming language always comes with specific rules governing exactly what makes valid code. They may be less rigid than the rules governing the ISA (for instance, numbers above 255 will be allowed), but they are no less precisely defined. These rules saying how to write valid code are collectively called the programming language's **syntax**. The rules explaining what the various operations in the syntax actually do are called the language's **semantics**.

For instance, a programming language's syntax may specify that [number] + [number] is a valid piece of code. The specification that, further, this piece of code computes the sum of the two numbers is the semantic definition of this syntax element.

The language's syntax and semantics together define the interface of the programming language layer--they say what inputs the programmer is allowed to give it, and what those inputs will actually do. So the use of layering in this case means that the programmer needs only to understand the syntax and semantics of the programming language, and can ignore the crunchy details of the ISA.

Further, on top of a programming language, we will often deal with **libraries**, i.e., pre-written code that performs commonly-used operations. For instance, at some point, many programs have to draw a square on the screen, whether for a window, a button, icon, spaceship, whatever. Every programmer who ever wanted to draw a square could write the program for `turn all the pixels with coordinates 10,154; 10,155; 11,154, and 11,155 all black'. Or, there could be a library for this, and now they can invoke the library function whose semantic meaning is `draw a 2x2 black square with top-left corner at coordinates 10,154', which runs the pre-written library code that does exactly that.

1.4.3: A word about programming in practice

When we sit down and we have a high-level task set in mind like, "Create a world-class search engine" or even something simpler like "Create a small platformer game," there are a few questions we need to

answer, and a few things that can happen along the way that we will take a moment to address now. Specifically:

- How do we start designing the program?
- What programming language, out of the many hundreds that exist, do we use?
- What things can go wrong when writing a program?

1.4.3.1: How do we decide what program to write?

Summary: Suppose we have a problem. Before writing a program to solve the problem, we write an algorithm--a description of the steps the program should take, but not written in the programming language's syntax. Then we can assess how expensive this algorithm is by considering various notions of algorithmic complexity, and whether a different algorithm might be better-suited to the problem given our resources.

Definitions: algorithm, complexity, trade-off

Say we have some problem we are trying to solve by programming a computer. A computer works by executing a sequence of arithmetic and storage operations, so we need to express any solution in terms of these things--storing numbers and performing arithmetic on them. However, we do not necessarily start by writing code, since this requires us to think about the specific way that the programming language requires us to write the various operations (i.e., its syntax). To begin with, it is often helpful to try to think of the solution in terms of what the programming language can do--basic arithmetic and storage--but to phrase it simply in English (or other human language of your choice). A sufficiently precise such phrasing is called an **algorithm**. Specifically, An algorithm must satisfy two properties:

1. Every step in it must be precise and unambiguous, to the point that it could be rephrased as a combination of the basic types of operations that computers perform. By having this property we can ensure that a computer can execute every step in an algorithm.
2. It must terminate. That is, we will only consider algorithms that have a definite ending after finishing some number of steps.

We discussed this in general earlier when we described the problem of computing the square root of a number: Our algorithm for computing the square root of a given number x was:

1. Start by storing x/2 in a storage slot called s.
2. Take the current value of s and add x/s to it.
3. Divide s by 2
4. If s*s is not yet as close as we want it to be to x, return to step 2

This last step might look a little fishy--everything else in the algorithm was essentially basic arithmetic, but "as close as we want it to be" is not likely a thing built in to our program language.

But what we actually mean by this is something quite precise: Say we want s*s to be within .001 of x. Then we are just asking that s*s - x be between -0.001 and 0.001. So we can rephrase:

1. Start by storing x/2 in a storage slot called s.
2. Take the current value of s and add x/s to it.
3. Divide s by 2
4. If s*s - x is bigger than 0.001 or smaller than -0.001, return to step 2.

Now truly the only things we do in this algorithm are basic arithmetic with numbers and storing the results, and choosing what instructions to execute next based on the result. So when we want to actually make a computer do this, we simply have to rephrase it using the particular syntax of our chosen programming language.

This is not the only algorithm that solves the problem. Another algorithm that solves it is based on the following idea: Suppose we are looking for the square root of 38. I know the answer has to between 0 and 38, so I'll try the midpoint--19. 19*19 = 361, which is way too big. So the answer's between 0 and 19, so we take the midpoint--9.5. 9.5*9.5 is 90.25--still way bigger than 38--so we know the answer is between 0 and 9.5. The midpoint is 4.75, and 4.75*4.75 is 22.5625. This is now too small, so the answer is between 4.75 and 9.5. The midpoint is 7.125. Repeating this process enough will eventually converge on the actual value of 6.1644....

(This example of computing the square root of 38 should suggest the general procedure, for which the reader is invited to, at leisure, write an actual algorithm.)

Because there are different algorithms that solve the same problem, it is useful to have some notion of how 'good' an algorithm is. More specifically, an algorithm will always have to use resources to solve the problem--sometimes it will have to store lots of intermediate data, sometimes it will take a lot of steps, sometimes it will have to send lots of data to other computers, sometimes it will tie up lots of auxiliary special-purpose circuits, whatever. We can analyze how much of each of these a given algorithm will use. These will give various estimates of the **complexity** of the algorithm--that is, an answer to the question 'how much memory, time, or other resources does the algorithm require to come to its answer?'

If we have several algorithms that all accomplish a given task and must select one to use, the first thing we can do is to compute the complexity of each algorithm. Maybe some algorithms will simply be worse in all measurements, but often one algorithm will be faster but will, for example, use more space in memory, whereas another will be slower but use less memory. Such situations are called **trade-offs**, where we cannot have the best of all worlds and must exchange efficiency in one area for efficiency in another.

To then make a decision about which algorithm we want, we have to consider what it is in our specific situation that we care the most about: Maybe we want to get the job done in as little time as possible and have a massive budget, so can buy whatever resources we need to run a very memory-hungry algorithm. In that case, we want to understand and minimize the time complexity of the algorithm and perhaps can ignore, within reason, other notions of complexity. Or maybe our algorithm is running on a small chip inside a cheap camera and we have very limited memory to work with, but we are not worried about taking a little extra time. Now our choice of algorithm will reflect our different needs.

1.4.3.2: What programming language do we use?

Summary: There are two broad classes of programming languages--low-level languages that allow great control over exactly what happens during their execution, and high-level languages which provide less control but correspondingly do not require the programmer to think about all the details of what's happening under the hood.

Definitions: low-level language, high-level language

We will spend a decent amount of time in this book talking about writing programs, so it will do here to mention some ideas associated with various programming languages and the process of creating programs using them.

As we mentioned, the ISA may be able, with enough work, to solve any problem you have, but it is so low-level that it is cumbersome to actually write code using it directly. If we use the ISA directly, however, we get much greater control over precisely what steps the computer will actually perform, which can allow us to optimize our program very aggressively, for example. A **low-level programming language**, more generally, is any programming language that is close enough to the ISA to afford the programmer substantial control over what ISA operations the computer will end up doing. Examples of such languages include Assembly (which is actually programming in the ISA directly) and C.

For example, if we are using a low-level language and want to figure out which of two given numbers is larger, we may have to figure out exactly where in memory the numbers are stored, load them into the special memory where they can be operated on, subtract them, check if the result is negative, and if so store the second number as the result and if not store the first number as the result. This is a lot of work, but it means for example that we can choose precisely where in memory the numbers are stored, which can provide extra flexibility that is sometimes useful.

By contrast, a **high-level programming language** is one where to program in it, we think in terms of often more human-friendly abstractions that the compiler will, behind the scenes, turn into ISA operations that we the programmers need not concern ourselves with. For instance, generally storing a number using a high-level language is as simple as:

```
x = 2
y = 199
```

We don't have to think about how you need to find a free slot in memory and store the numbers in those slots and remember which slots you used--we just give the values names and the compiler takes care of organizing the actual underlying memory.

In a high-level programming language, then, computing the maximum is as easy as writing:

```
max(x,y)
```

1.4.3.3: What can go wrong when writing a program?

Summary:

If a program breaks the rules of the programming language's syntax, this is called a syntax error. The compiler will know that the program is invalid and will generally give some indication of what specifically in the program was problematic, so these errors are often easy enough to fix.

If, on the other hand, the program was written with correct syntax, but had a logical mistake whereby it does not actually do what the programmer intended, this is called a runtime error and is much harder to fix.

Definitions: syntax error, logic error, runtime error, debugging

When you've written a program and are ready to run it on the computer, it will very rarely behave correctly the first time (this is as true for 20-year programming veterans as for neophytes). There are two sorts of things that can go wrong. Remember that a programming language is defined by its syntax and semantics: what sorts of constructs are allowed as valid programs, and what the various constructs actually do. Corresponding to these two are three types of problems you may encounter when running your code: **syntax errors** (or **compile-time errors**), **logic errors**, and **runtime errors**.

A **syntax error** is where your program was written with invalid structure that fails to conform to the syntax of the programming language. For instance, you might have a programming language whose syntax says that

```
max(?,?)
```

is a valid piece of code to write, where the blanks can be filled with any numbers. If you made a typo in your program, you might accidentally write something like:

```
max(2,3(
```

This would be a syntax error. Your program will never be run because it will never even be converted into ISA operations because, in turn, the compiler (which does this job) only understands code written that obeys the programming language's syntax, which the above does not.

A **logic error** is when your program is written with valid syntax, so the compiler can turn it into ISA code that the computer can actually run, but where what gets run is not what you intended. For instance, if we again intended to compute the maximum of 2 and 3 but accidentally typed instead:

```
max(2,2)
```

then this is now perfectly valid code--it does follow the syntax of max(?, ?), so will be run without complaint, but will give back 2 as the result, whereas the desired result was 3.

A **runtime error** is a slightly more subtle one that we shall not discuss at length. In brief, it is when some external circumstance to the program causes the program to fail. For example, if the program attempts to save some file to a disk, but the disk is full.

When your code has an error of any kind, obviously you'll want to fix it. The process of fixing errors is called **debugging** (there was an incident in the very early days of computers in which a system was behaving oddly and the problem was traced back to a bug that had decided to nest in the equipment; thanks to this, errors--though mainly logic errors--are also called 'bugs'). With compile-time errors, debugging is usually easier--if the compiler finds a piece of your code that it cannot understand, it can just tell you "Hey, I don't understand line 15 in file X," and you can go look at file X and hopefully spot the invalid syntax and correct it.

When our code has a logic error, debugging is much more challenging because the computer doesn't see any problem--the code is understandable and runnable, but it just doesn't do what the programmer intended. So the computer cannot tell where the code messed up because it doesn't know it messed up! (Computers cannot read our minds. Yet...)

In chapters 2 and 3 when we talk about programming, we will talk through some techniques for debugging logic errors, but this is a skill that you will constantly be working to improve throughout your programming life and is simply one of the hard parts of programming.

1.5: Down the tower: How to build computers

Now that we have begun to see that using this limited notion of computer, combined with the power of abstraction, we can possibly build complicated and useful systems out of such a simple device as our notion of "computer," we turn in this section to the question of how to actually build such a device.

1.5.1: State machines

Summary: The basic operations that we want to be able to perform will be organized using an abstract scheme called a state machine. This will break each instruction down into a sequence of basic operations.

Definitions: state machine

While our ultimate goal will be to build a real machine, we start with an organizational tool called a "state machine." A **state machine** is an abstract "machine" that consists of a collection of "states" that the machine can be in. The machine accepts inputs one at a time, and each state also specifies,

for each possible input, which state to change to if that input comes in.

For example, imagine we are trying to design a simple robotic cat. We can describe our design using a state machine, which will have states called "Purring," "Neutral," and "Grumpy." The possible inputs to this state machine might be, for example, "Feed," "Pet," and "Ignore." So if the cat is in the "Grumpy" state, then if it gets the "Pet" input it will remain in "Grumpy", whereas if it gets the "Feed" input it will move to "Neutral".

This information is often represented as a drawing with the states represented as labeled circles and the transitions as labeled arrows between the states, like so:

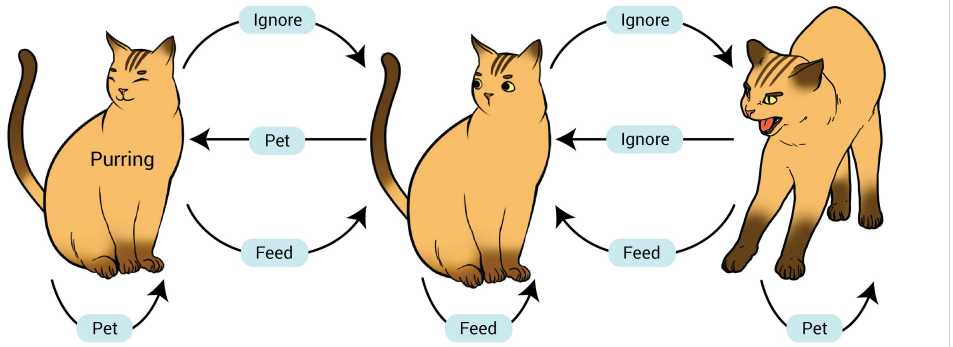

Figure 1.5.1.1: Model of a cat's behavior using a simple state machine

1.5.2: Microarchitecture

Summary: To actually build the computer, then, we will create black-box functional units that perform each of these basic operations--storing a number, adding two numbers, etc. The arrangement of these basic functional units is called the computer's microarchitecture.

Definitions: microarchitecture

A real cat is a lot more complex than our above example state machine gives it credit for. To do a cat justice, we would have to have states also for things like exercising its hunting instinct and for whatever it is that this cat[3] is doing. And there are a lot more ways of interacting with a cat than we listed above.

Computers, on the other hand, are relatively simple: The things a programmer can tell it to do are listed exhaustively in the ISA. Then the states the computer can be in correspond to the various things it will do in response to receiving a particular ISA command.

For example, suppose it gets the instruction "Add 5 and 8 and store the result in storage space number 51". The first thing it should do is to actually add these two numbers. Then it should store the sum somewhere. Finally, at this point it is finished with the add instruction, so it should go look at the next instruction whatever that is. We can summarize this sequence of events with a simple state machine describing only the addition-capable part of the processor:

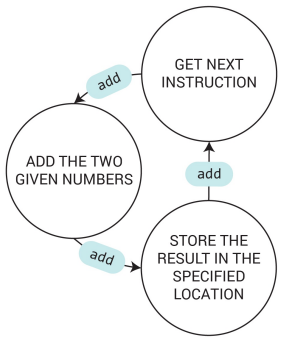

Figure 1.5.2.1: A state machine for a simple computer that can only add numbers. This machine advances state automatically at regular intervals. It chooses its next state based on the current instruction, which does not change as a result of the "Add" or "Store" states, so any single "add" intruction goes through both of these states before the machine gets the next instruction.

This state machine is slightly different from the cat example: The processor has a memory bank where

it stores all its instructions (the stored program we described earlier). At all times, this memory bank is sending out the current instruction. This instruction is the input to our state machine. The state machine simply advances on its own at regular intervals, and which state it advances to is determined by the current instruction. The instruction itself doesn't change unless we get to the state "Get the next instruction."

For example, the state machine starts at "Get next instruction". Suppose the memory bank storing the instructions looks like:

Add 5 and 8 and store it in the 11th memory slot
Add -1 and 90 and store it in the 5th memory slot

Then to start out, "Get next instruction" will make the current instruction be "Add 5 and 8 and store it in the first slot." The state machine will advance on its own. Because its input is the current instruction, which is an adding instruction, it will advance to "Add the two numbers" and perform the sum 5+8. This doesn't affect which instruction is the current instruction, so the next time it advances, the state machine's input is still the same adding instruction, so it advances to "Store the result", so it takes the result that it computed, namely 13, and stores this as directed. This still doesn't change the current instruction, so it advances to the state "Get next instruction". This changes the curent instruction to "Add -1 and 90 and store it in the 5th memory slot." Now that is the current instruction, and the cycle continues.

This state machine is of course just a sequence of steps to execute for each instruction. It only really becomes interesting when it involves choosing different steps to execute depending on the type of instruction. The ISA doesn't only have arithmetic instructions, but also some for storage and some for branching and some for I/O. So let us add states for the instruction "Jump ahead X instructions":

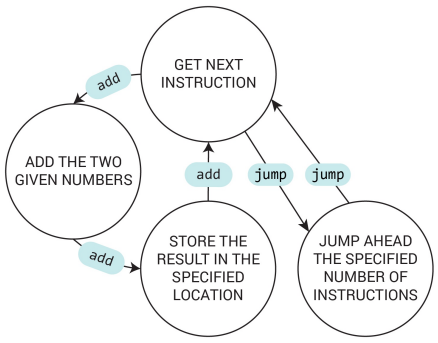

Figure 1.5.2.2: A state machine for a computer that can both add numbers and branch to different instructions.

As you can imagine, we would continue adding to this picture for every kind of ISA operation imaginable--Storage operations, input, output, subtraction, multiplication, etc.--until it became a giant spaghetti of states and arrows, but we'll leave off the spaghetti until we're ready to do it for real in a later chapter.

In this example, the states were a bit vague: One state just said to "add two numbers" without saying anything about how to do it. In reality, the states for our computer will correspond to individual pieces of hardware doing specific things on the way to executing all the steps needed to complete any particular ISA operation.

Having so organized the basic operations that our machine should do, we have broken the problem into two tasks:

- Build physical components that perform the tasks described in the states of the state machine.
- Connect up these components in a way that gives the behavior described by the state machine.

A precise description of the individual components and the connections between them, along with a state machine describing how they interact, is what will be called the **microarchitecture** of the machine.

1.5.3: Binary representation

Summary: Computers are machines. But we said that computers operate on numbers, whereas physical machines can only manipulate physical things! So we have a scheme for representing numbers by something physical. This scheme is called "binary."

Definitions: binary (basic)

Before we can start creating machines as physical devices, we need to solve a more fundamental problem: Computers as we have defined them operate on numbers. But physical machines can only manipulate physical things, so we need a way to represent numbers by something physical.

Actually, when we write down numbers, we are already representing them by something physical: To write the number 483, we took the symbols 4, 8, and 3, and wrote them next to each other. This physical arrangement of ink on a page (or pixels on a screen) somehow corresponds to the number we say as `Four hundred, eighty-three'. This method of representation, where every number can be written as a sequence of symbols 0, 1, 2, ..., or 9, is called 'decimal'. We use 10 symbols in this representation simply because we have 10 fingers, so it turns out to be well-adapted for human-to-human communication.

For the purposes of making a computer, there is another way to represent numbers that will be much more convenient, called **binary**. In this system numbers are represented as sequences now just of the symbols '0' and '1'. But a sequence of '0's and '1's, now, we can imagine representing with some simple physical thing--e.g. a sequence of switches where a switch being off represents a '0' and on represents a '1', or a sequence of wires, where a wire with electrons flowing through it will represent a '1', and a wire with no electrons flowing will represent a '0'.

Then when we come to build the part of the computer that adds numbers, it will take in two binary representations--that is, two sequences of wires whose patterns of electron flow will represent the two numbers we want to add, and we will have to construct some circuitry so that on the outgoing wires (representing the output), the sequence of '0's and '1's going out represents the sum of the two input numbers. We will get more into the details of this in chapter 5.

Building this component happens in two steps: First, we have to understand abstractly what is the procedure for computing sums of binary numbers. E.g., What is the procedure that computes '001101 + 101001'?. This is addressed in chapter 4 in detail. Then we need to construct actual circuitry that performs this procedure. This will be discussed in chapter 8.

1.5.4: Logic gates and transistors

Summary: The functional units comprising the microarchitecture, finally, are themselves constructed using basic components called logic gates, which are in turn constructed out of transistors.

Definitions: logic gate, and gate, truth table, xnor gate

Once we have decided to represent our numbers in binary, we need a way to take two numbers and perform operations on them. The building blocks we will use to do this are called **logic gates**. Most basically, a logic gate is a device with two input wires and one output wire. Referring back to the previous section, where a wire represented a binary digit (current flowing = '1', no current = '0'), the gate will either send or not send current on the output wire depending on what's happening at the input wires.

Different types of logic gates will have different rules for how to determine the output based on the inputs. For instance, we can have a gate whose output will be 0 unless both inputs are 1. This is called an **"AND gate"**. There is a standard symbol that is used when an AND gate appears in a circuit diagram, which looks like like this:

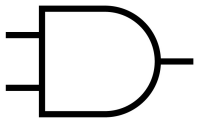

Figure 1.5.4.1: The AND gate

The two lines coming off the left side represent wires, that in a larger diagram would be connected to some other circuitry that provides the input for the gate, and the line coming off the right represents the output of the gate, which would also be connected somehow to some other part of the circuit.

We can summarize the behavior of an AND gate rather succinctly in a table, called the gate's **truth table**:

Input 1	Input 2	Output
0	0	0
1	0	0
0	1	0

1	1	1

Of course, we can make any gate we want by writing down a different truth table--for instance, the following gate (called an **XNOR gate**) will output 1 if the inputs are the same and 0 otherwise:

Input 1	Input 2	Output
0	0	1
1	0	0
0	1	0
1	1	1

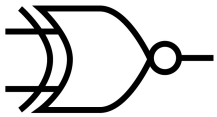

Figure 1.5.4.2: The XNOR gate and its truth table

The true power of logic gates will come from connecting them together, as we will see in chapter 8. There, we will learn to build things like a circuit that adds binary numbers out of nothing more than the various kinds of logic gates.

4-input AND circuit

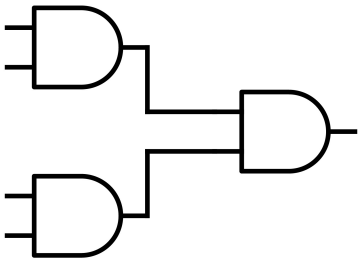

Figure 1.5.4.3: A four-input AND circuit

Finally, once we know how to build interesting things using gates, it will behoove us to figure out how to actually build physical logic gates. We do this using, finally, actual physical devices you can make (or buy) called transistors, which will also be discussed toward the end of chapter 8.

1.6: Theoretical underpinnings--why are computers so powerful?

Summary: Just because we can write an algorithm to solve a problem, why does that mean we will be able to write a program that expresses this algorithm, thereby using a machine to solve the problem? Alan Turing devised an abstract notion of a machine, called a 'Turing machine' and noted that there are certain Turing machines--the so-called 'Turing complete' ones--that can simulate the behavior of any other Turing machine. Since we can make a machine that follows any given algorithm, and since our computer will be Turing complete, we can conclude that our computer can itself simulate the behavior of the other machine, i.e., our computer can run the specified algorithm.

Definitions: Turing machine, Church-Turing thesis, Turing complete, computational complexity

We now turn to a fundamental question we've been blithely presupposing throughout the preceding discussion: Why do we expect to be able to implement any algorithm using just a computer?

As we have defined it, 'computer' refers to an apparently quite limited device, not obviously capable of running games or talking to other computers or any of the thing you already think of as doable with a computer. There are two steps to building a system on top of a computer: Since as we have defined it, a computer deals only with numbers and can only do arithmetic/logic on these numbers and store them, we need to encode the objects of our system. For instance, if we want to think about searching through text files, we need a way to represent text as numbers.

Secondly, once everything is numbers, anything we want to do then consists of turning a bunch of numbers into another bunch of numbers (e.g. the numbers encoding a query to the search engine

together with the numbers encoding the database of search able documents into the numbers encoding the search results). So we need some guarantee that any numerical computation can be done by a computer that, by definition, only has the capability of basic arithmetic.

In 1936, Alan Turing proposed an even more apparently limited sort of device, called a **Turing machine**. This was a device with three parts: a tape consisting of slots in which numbers may be written, a read/write head that can move along the tape reading the numbers written there or writing numbers onto the tape (or possibly overwriting numbers already written on the tape), and a state machine by which it will decide what numbers to write and where on the tape to write them.

The Church-Turing thesis (called a "thesis" because it is more a philosophy more than a precise axiom) says that if you can write an algorithm for performing a computation, then a Turing machine can be built that performs that same computation.

A Turing machine is said to be **Turing complete** if it is capable of simulating any other Turing machine (and therefore of running any algorithm, if you believe Church's Thesis). Since our computers have memory (which serves the same function as the tape) and have access to memory operations and arithmetic operations, it is not hard to any Turing machine might be representable as a program in our computer. And since we can write any program we want, our computer can therefore simulate any Turing machine, and should therefore be Turing complete. In fact, if one performs a more detailed study of the matter, this argument can be made rigorous. So for this purely theoretical reason, if we have a problem and can write down an algorithm for it, we can know that there should be an ISA-level program that realizes this algorithm. (It may take an substantial amount of skill to concoct this program, but at least in theory, we can be assured that it will be possible.)

We already spoke about the various notions of complexity for a given algorithm. As we think about all possible Turing machines, one notion that arises is the computational complexity of a problem itself (as opposed to a specific algorithm that solves the problem). The **computational complexity** of a problem is defined to be the minimum complexity among that of all algorithms that solve the problem. This notion allows us to distinguish the problems where we use a resource-intensive algorithm because we simply couldn't think of a simpler one and the problems that are actually inherently difficult and require at least that minimum complexity regardless of cleverness.

Actually figuring out the computational complexity for a given problem is quite hard. Indeed, looking at the definition, to know it appears to require us to analyze all possible algorithms for solving a problem, and there are infinitely many such algorithms! The area of computer science known as computability theory, using some quite clever techniques, manages to get answers to these sorts of questions. In your study of computer science you will almost surely encounter this area at some point, but we will venture no further in that direction in this text.[4]

1.7: Some modern computing systems

At this point, we've seen how it can be somewhat easier for programmers to supply computers with their arithmetic and storage operations. But we have yet to bridge the gap between these basic numeric operations and the exciting applications we associate with modern computers.

The actual bridging of this gap will only be completed by the end of chapter 6, but to begin to gesture in this direction, the next few sections shall introduce three such applications, provide a high-level overview of each, and start breaking these down into components until we reach pieces that consist of concrete programming problems.

This breakdown, as well as the programming of the individual pieces, will happen in full throughout the course of this book, but we'll start it now, and then flesh the components out as we go along.

1.8: Search engine

1.8.1: High-level problem

Summary: The high-level goal of a search engine is to accept user input and find, among a vast space of documents, those documents that best match the user's input.

We want a system into which any user anywhere who wants to know something can visit our website, input a sequence of words, and be presented quickly with a list of websites that contain the information he was looking for.

The scope of the problem is dizzying--Google themseves estimate the size of their index at over a billion GB [5]--i.e. a quintillion bytes--with about 2 million search queries coming in every minute [6]. The engineering problem associated even with just getting a webpage that displays the search box sent out 2 million times a minute are substantial, let alone solving the high-level problem stated above at least as many times every minute. This massive challenge has engendered a highly complex solution, to the point where no individual engineer at Google has a hope of understanding the entire system from top to bottom. But then how can it work?

The key, once again, is abstraction--the problem is broken up into distinct sub-problems, and separate units are engineered to solve each of these individually. A team is tasked with crawling through all the websites on the internet and collecting data and keeping it current. Another team might manage all the physical storage units needed to keep all that data, ensuring proper redundancy in case of hardware failure, and ensuring uniform access to the data even if some of it is in Canada and some of it is in Japan. A further team will have to find a way to automatically determine the most relevant results for a given query and rank them for display to the end-user. And if abstraction is employed properly, the search team will not need to know how specifically the data management team stores the data--they just know they can ask "Give me a bunch of data to search" and it will happen. And the storage team doesn't have to know how the crawler team gets the data--they just have to tell the crawler team how to feed them new data and store the submissions.

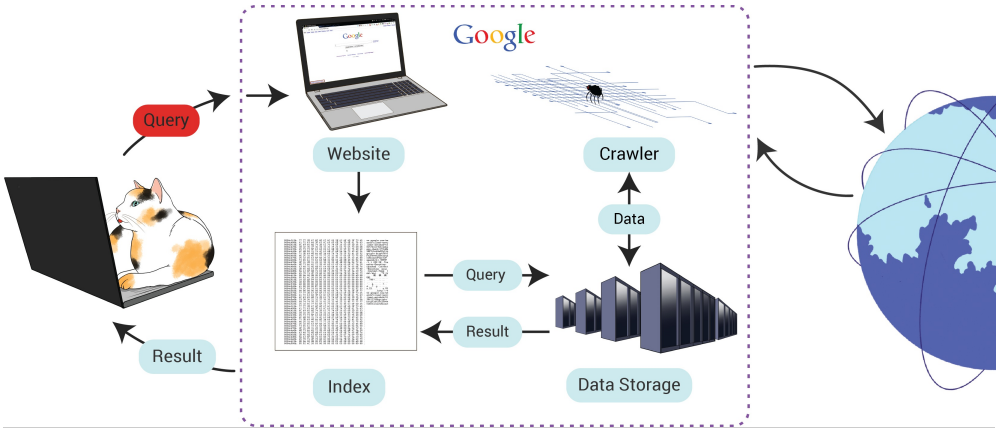

Figure 1.8.1.1: A level-2 view of the search engine system.

1.8.2: Zooming in: Crawling the web

Now say you're specifically on the team responsible for crawling the internet and collecting the data. You can receive user queries from the website team, and once you decide how to organize the data for quick searching, you can then send the data to the datacenter for safe, redundant, distributed storage.

1.8.2.1: Software side

Summary: Collecting the data of all pages on the internet is hard enough, but if we simply store all these documents naively, then searching through them will take forever. We want a more clever scheme for storing this data, then, that makes it easier to search quickly.

Once we have the data, we need to store it, but with an eye toward what we want to actually do with it.

First, if we find a match for a user query within some part of the data, we must be able to tell which specific place on the internet this data came from so that we can point the user there.

Secondly, in view of exactly how much data we will be collecting, we need to store it in a way that will make searching through it fast. As a basic example, suppose we are searching the works of Shakespeare for the word 'bestride'. If we have all the works of Shakespeare stored, we can simply open each document and compare each word in turn to the query, and when we find a match, we report the name of the document we're currently searching through.

But wouldn't it be nicer if instead we had an alphabetized list of all the words in Shakespeare's collected works, say in a book much like a dictionary. Except instead of listing the definition of the word next to the actual word, this book lists each location (e.g. which play, which act, which scene) where the word may be found.

We are storing the same information in each case, but because we organized it more cleverly in the second example, the lookups will be faster. So too with our search engine--if we find a clever way to store the data, we can make searching through the data much faster than if we store the data in the naive way.

So we have designed an algorithm for searching, namely: store the data in a lexicon, and then, when you get a query, take each word in the query, look it up in the lexicon, and see what websites contain that word. We feed that list of websites, along with the original query, to the ranking team to put the results in order of relevance to the query, and then to forward on to the user for display.

Further, in designing our algorithm, we have ensured that its time-complexity is not too great. For

instance, if a website becomes twice as big, it contains more words, so we get more entries in our lexicon, but looking up words in a lexicon is really rather fast, even if it is somewhat large. If we add more websites with the same words, this doesn't affect our lookup time at all. We'll have more results, so the ranking team who have to order the results will have more work, but the number of entries we search through in the lexicon remains the same.

1.8.2.2: Hardware side

When describing the algorithm above, and even when actually coding it up, we take a somewhat idealized view of memory--we talk about taking a query, breaking it up into words, and searching a stored index of words for each of the query words.

But none of this is actually literally doable, at least not in the way the instruction `take two hot dogs and throw them out a second-story window into a lake' is doable. The `text' you see on your screen is just a grid of lights in some pattern. The real objects inside your computer are electrons. And when we say we have some text stored, the way to `realize' this statement is the following: Text is be comprised of characters. Thus if we can store a sequence of characters, we can store `text'. There are only finitely many characters in the world. If we restrict to the English alphabet and English-friendly symbols, there are fewer than 128 of them.

We assign each character a number between 0 and 255

9	[tab]	63	?	96	`
10	[newline]	64	@	97	a
32	[space]	65	A	98	b
33	!	66	B	99	c
34	"	67	C	100	d
35	#	68	D	101	e
36	$	69	E	102	f
37	%	70	F	103	g
38	&	71	G	104	h
39	'	72	H	105	i
40	(	73	I	106	j
41	)	74	J	107	k
42	*	75	K	108	l
43	+	76	L	109	m
44	,	77	M	110	n
45	-	78	N	111	o
46	.	79	O	112	p
47	/	80	P	113	q
48	0	81	Q	114	r
49	1	82	R	115	s
50	2	83	S	116	t
51	3	84	T	117	u
52	4	85	U	118	v
53	5	86	V	119	w
54	6	87	W	120	x
55	7	88	X	121	y
56	8	89	Y	122	z
57	9	90	Z	123	{
58	:	91	[	124	|

59	;	92	\	125	}
60	<	93	]	126	~
61	=	94	^		
62	>	95	_		

Then when I press the A key on the keyboard and that gets stored somewhere in memory, it gets stored as the number 65. And how do I store numbers? I can represent numbers in binary--for instance, 65 = 100001. And this, finally, I can realize: Any number 0-255, when written in binary, has 8 digits or fewer. So I can have a sequence of 8 slots where I can store electrons, and if a slot contains electrons I will think of it as a 1, and if not, I think of it as a 0. That is, I can store a whole byte (8 bits) using actual hardware.

And if I have a whole array of these byte storage units, I can store now a sequence of characters, i.e., I can store text.

(Note that at no point thus far is the connection made between the number 00100001 and the shape 'A'--this connection only happens when we send this number to the screen for display, whereupon it reads this number and turns on the appropriate grid of lights to the right pattern so that it displays the 'A' shape. But until you get to the display, everything is treated as numbers, or more precisely, sequences of bits.)

OK, so we can store text. Great. But we also have our program that we wrote. We'll look at only a little piece of this, where we have picked out a query word and one of the indexed words, and we have both in memory somewhere, and we want to see whether they are the same. To do this, we'll have to start by taking the first character of each and comparing those. So at some point in the program, this operation will be performed.

But the program was an abstract thing too! It needs to be something real as well! OK, yes. Fine. Good. Remember that the computer only accepts instructions in its ISA. One of those instructions might abstractly mean `compare these two bytes'. But inside a computer, this instruction is just some specific number. So the computer reads out the next number from wherever it stores its instructions. Maybe it reads out 10011110. This sequence of electrons comes through some wires, toggles some things, and causes the electrons representing the bytes for the first characters of the two words to flow down some more wires into some comparison circuit that will output more electrons representing whether it found a match.

All these electron-moving operations happen using wires and gates. For instance, somewhere in the compare circuit, we'll have two bits coming in--the first bit of the first char of query word, and the first bit of the first char of the index word. And we'll want to compare these. So we feed them into an '=' gate, exactly as in our example above. As we have 8 bits in each character, we need 8 of these '=' gates

But now we have 8 bits coming out of these 8 gates--each bit representing whether the respective bits of the inputs matched. On the other hand, we want only a 1-bit answer: 1 if every single bit matched, and 0 if not. So we can feed these 8 outputs into an 8-bit AND-gate, much like the 4-bit one from earlier, which will output 1 if all the inputs are 1, and 0 otherwise:

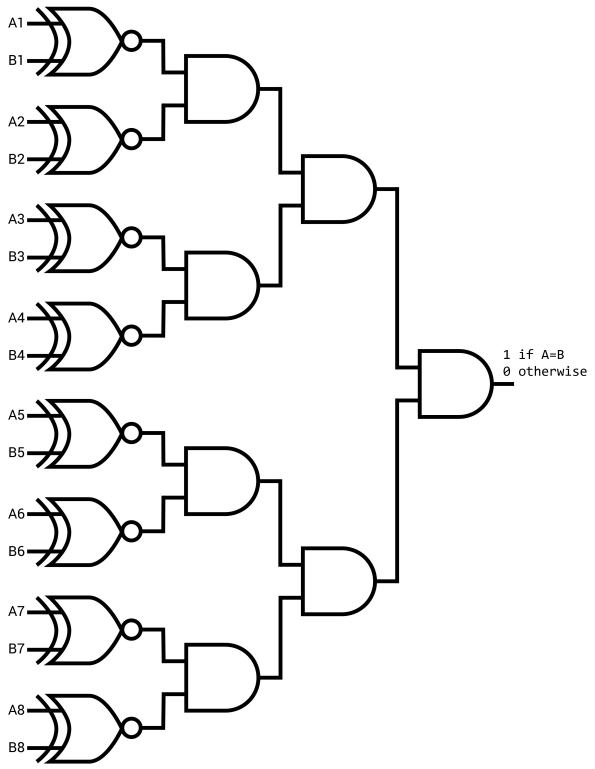

Figure 1.8.2.2.1: Circuit to compare two bytes

1.9: Game console

1.9.1: High-level problem

A game console is a system specially designed to allow the user to play high-quality 3D games. Depending on what is meant by "high quality", this presents us with another numerically impressive problem. However, to get at the fundamental difficulty involved here, we must understand what concrete problem we are actually solving. There are, by and large, two main pieces of work that a game console must do, which we might label as "modeling" and "rendering".

Modeling is the business of computing the locations of all objects subject to whatever forces are acting. A common sort of modeling is physics modeling, where maybe an object falls into a stack of crates and the crates themselves fall in response and some of them perhaps splinter, and the game console must compute all of this behavior in real time.

Rendering is the business of actually displaying everything that we have computed onto the screen. This is a distinct and independently difficult task: The model tells us that we have 10 crates and has perhaps computed their positions and orientations in 3D space. But now we need to take that information along with information about what the crates actually should look like, not to mention information about existing light sources, maybe whether the atmosphere is dusty, and from what position we should appear to be looking at the crates, and turn this all into a 2D image suitable for displaying on the screen that will appear to the end-user as representing a 3D virtual reality.

If you are OK with a simple game, then neither of these problems is too terrible. Old games like Pong had simple physics modeling for the bouncing ball, and simple rendering of rectangles for the paddles and a circle for the ball. But at the level of modern games, the objects of the game are most often stored as a very large number of triangles, all connected in some way to make the shape of the desired object. For example, here is a 3D model of a teapot, together with a visualization to indicate how what looks like a smooth surface is actually many small triangles fused together:

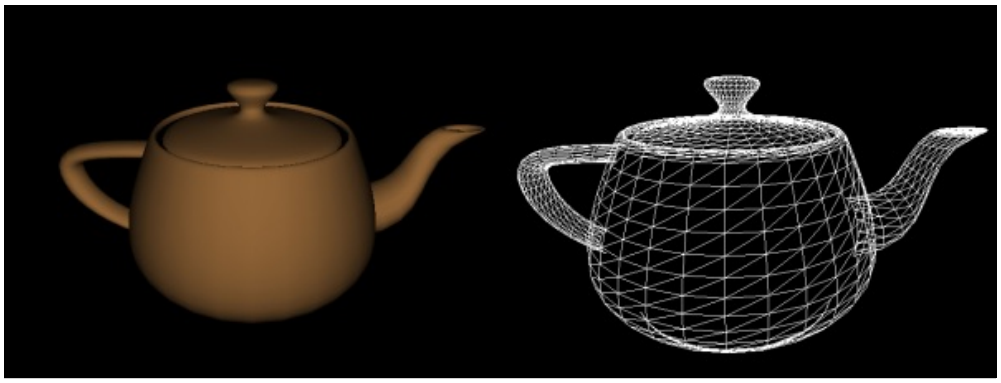

Figure 1.9.1.1: A 3D rendering of a teapot using a large of triangles.

In order to make this look smooth, we simply used a large enough number of triangles. If we use fewer, then we get something like:

Figure 1.9.1.2: A 3D rendering of a teapot using fewer triangles.

So in fact, when we do a physics calculation like "did the falling teapot actually collide with the crate?", what we actually do is take each triangle in the teapot individually, and check whether it interacted with any of the triangles in the crate. And when we want to render the teapot, we are actually taking every triangle of the teapot and computing how to display it on the screen (given ambient lighting/atmosphere/etc. conditions).

Both of those steps involve potentially millions of calculations to do just once, but in order to present the player with a reasonably fluid gaming experience all the modeling calculations have to be done and the results rendered a bare minimum of 30 times every second. (A single rendering of the scene is called a "frame", and so this rendering rate is sometimes called "frames per second".)

So once again, the problem quickly presents us with a numerically massive task, and the infrastructure involved with solving the it is substantial. As before, it is likely beyond the scope of any individual's skill level. But, as ever, it can still be broken down into manageable sub-problems, and success comes from solving each of these and making the smaller solutions work together to attack the whole.

To break the problem down, we need to think carefully about what components are involved and which is responsible for what: At the highest level, there is the console itself, and the game we are playing. The console needs to be able to perform all the calculations and display all the graphics that the game needs, and the game needs to manage which things it wants displayed and which calculations it wants to do.

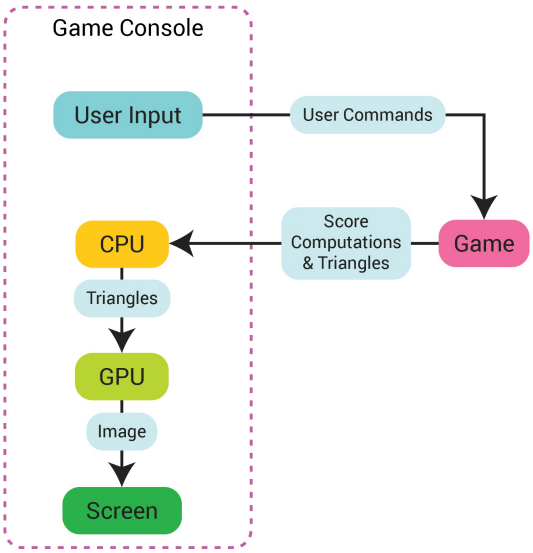

Figure 1.9.1.3: High-level view of the game console system.

1.9.2: Hardware side: The console

As we mentioned, the console itself needs to be able to handle two different sorts of computations: modeling and rendering.

Rendering computations are relatively predictable, in that if you have the capability to do one thing extremely well and quickly, you can render most things to satisfaction. That one thing is to take in a single triangle (with all its position and texture and lighting information) and some information about the camera's location and orientation and figure out how to draw it on screen. If we can do this, say, billions of times a second, then even if our teapot had a million triangles, we could still render all the teapot's triangles 1000 times every second--well above our target of 30.

Rather than design an algorithm to do this on any normal computer, then, it turns out to be best to design special hardware just for performing this one task really really well. This is what is called the "graphics processing unit", or GPU, of a gaming system. So the GPU is a kind of special-purpose computer like the ABC from earlier--it is designed for performing a particular task very quickly.

The game will also have some logic that is better suited to general-purpose chips, like tracking points or health or how many camels are present and calculating the paths of flying teapots and testing whether two crates collided. (Somewhere along the line this example got a little surreal. Oh well.) So our game console will have a built-in general-purpose processor as well for performing the modeling calculations and that can control the GPU and get it to render whatever is needed.

1.9.3: Software side: The game

Once we have all the hardware as described above, then it comes time to use it. This is the responsibility of the actual game. The game's code is divided into three principal components:

- Rendering engine: This is the bit responsible for getting the graphics rendered. The game console already has some hardware ready-made to do this. So we don't have to have our program compute how to display the triangles we want. Our program does however have to actually pass all our triangles to the GPU to get them rendered. There is some small amount of work beyond this also: If we can figure out, for instance, that a certain teapot is actually behind the camera and doesn't need to be rendered right now, then we can skip those triangles and be more efficient.
- Physics engine: The general-purpose processor in the game console can be programmed to do all the physics calculations we need, but we have to actually write the program to do this as part of the game! This portion of the game is called the physics engine--the code that manages how fast the camels are going and which teapots are colliding with what else.
- Game mechanics: We also need something responsible basic game mechanics: Counting points, how many crates we've smashed, etc.

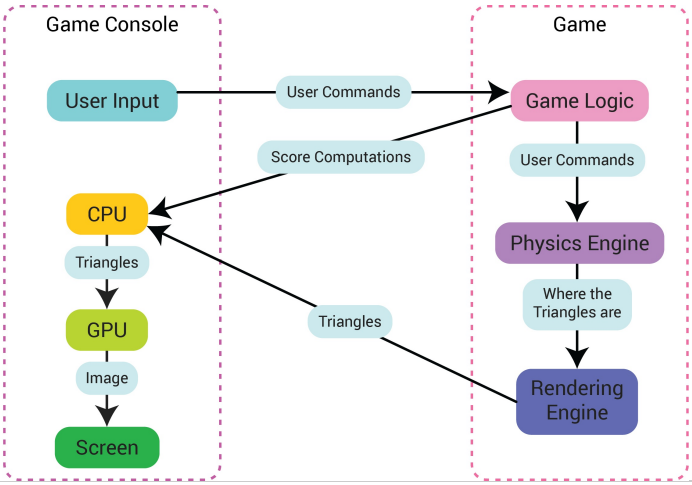

Figure 1.9.3.1: Peeling back a layer of abstraction: The components of the game and console, and their interactions.

1.9.4: Zooming in: Rendering a triangle

Let us look more closely at the business of rendering a single triangle. As we've seen, the physics engine calculates the location of all the triangles, and the rendering engine figures out which ones to pass onto the GPU, along with the camera angle and lighting.

For a start, these instructions physics modeling are largely arithmetic instructions run on the CPU. Then, the instructions for rendering a triangle, once we've computed all the data, will largely be I/O instructions, since they tell the CPU to output the triangles somewhere else--in this case the GPU.

Let us zoom in, now, on what the GPU does once it receives this data--specifically on what it does about lighting. It has been told where the light sources are relative to the triangle, so we know what direction the light should be coming from. How do we combine this direction and location information to get the intensity of light on the triangle?

To understand this, imagine the photons emanating from the light source as paint particles being sprayed out of a can. If we spray paint on a sheet of paper 1m away, the paper gets covered in some amount of paint. If we move the paper to 2m away, now, the same paint spreads out over an area 4 times as large, so we actually get 1/4 the paint we originally got on the paper.

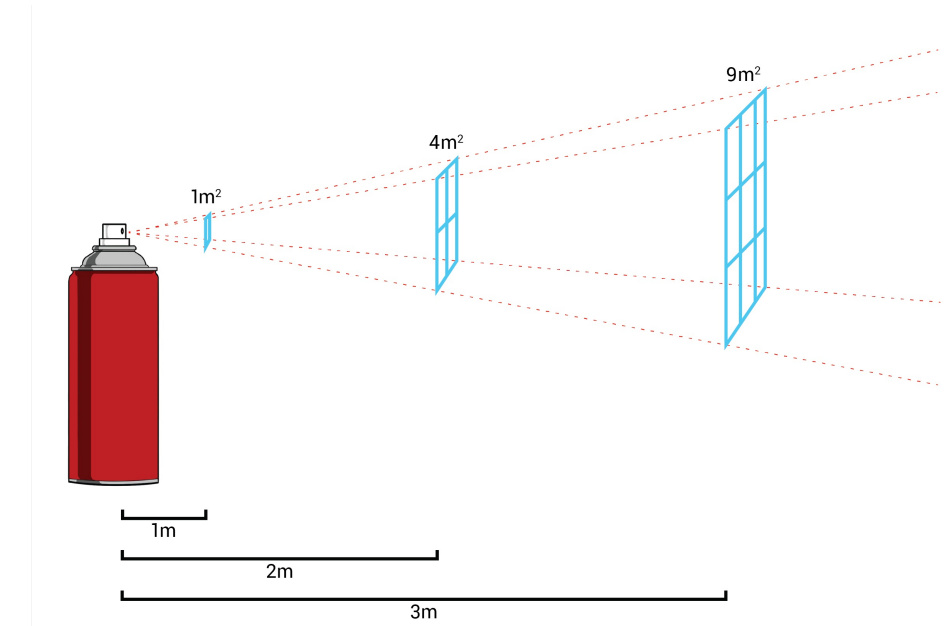

Figure 1.9.4.1: Spray paint intensity obeys an inverse square law.

If instead of paint emanating from a point, we have particles of light, the same thing is observed. So then we get that the intensity is inversely proportional to the square of the distance between the light source and the object. So, for example, if we move 7 times farther from a light source, we will get 1/49th the intensity.

So to compute intensity of light, we will have to compute distances. We know from our maths courses a formula for the distance between two points (x0, y0, z0) and (x1, y1, z1) as:

```
distance = √((x1-x0)^2 + (y1-y0)^2 + (z1-z0)^2)
```

So in particular, along with all that mess on the inside, have to compute square roots. Aha! But we already know how to do that from earlier in the chapter! We had this algorithm for computing the square root of a given number x:

1. Start by storing s to be s = x/2.
2. Take whatever s is currently and add the number x/s to it, making that sum the new value of s.
3. Divide s by 2, making the result the new value of s.
4. If s*s is not yet as close as we want it to be to x, return to step 2 and continue from there.

But now, instead of writing ISA operations to do this, the GPU needs to do this. That is, the GPU needs to have special hardware designed for computing square roots quickly. Recall we sketched what such a piece of hardware might look like at a high level:

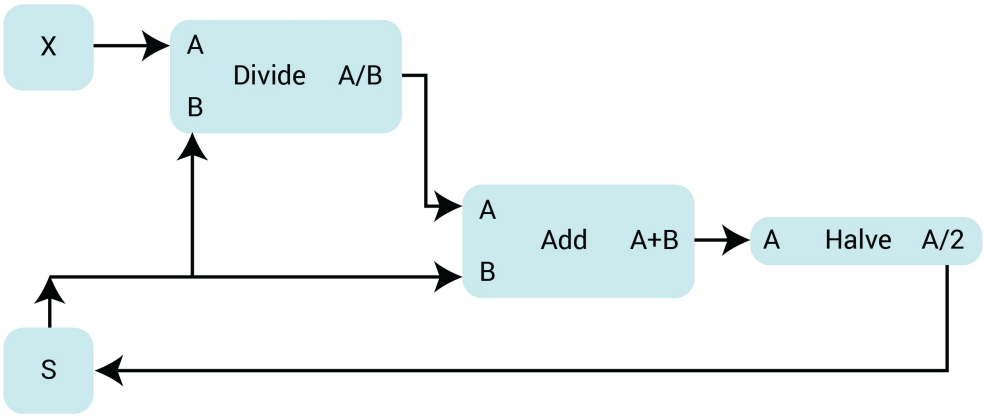

Figure 1.9.4.2: The square-root-inator--now a part of our GPU.

But let us zoom into this even further--what does this "divide by 2" gadget actually look like on the inside?

It somehow takes in a number and spits out the number, except halved. But what does this even mean for a physical device to "take in a number"? After all, this is supposed to be a machine (electrical, mechanical, or whatever, but a physical device either way), whereas a "number" is something abstract.

For this, we need to have a physical way of representing numbers. Say our halving gadget has 4 physical wires coming into it, each of which can either be carrying electrons or not. If we represent the presence of electron flow on a wire as "1", and the absence as "0", then there are 16 possibilities for what might be coming into our gadget: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. **Binary representation** is a scheme for treating each of these sequences as representing a different number. Without explaining how it works, we can for now simply say which combination corresponds to which number in a table:

```
Number   Representation
0        0000
1        0001
2        0010
3        0011
4        0100
5        0101
6        0110
7        0111
8        1000
9        1001
10       1010
11       1011
12       1100
13       1101
14       1110
15       1111
```

One feature that this particular assignment has, is that dividing by 2 is rather easy: 10 is represented by "1010", and 10/2 is 5, which is represented by "0101". Similarly, 8 is "1000", and 8/2 is "0100". In general, to divide by 2 (rounding down when presented with an odd number), all we do is shift all the wires to the right and stick a 0 on the leftmost wire of the output. Designing this as a physical device is actually quite simple, then:

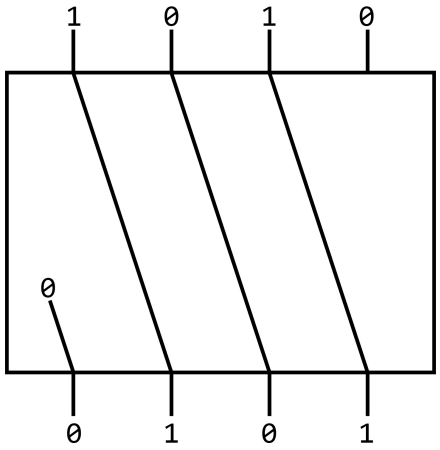

Figure 1.9.4.3: A simple kind of "divide by 2" device that only works on numbers 0-15.

1.10: Mobile phone

1.10.1: High-level problem

The problem that smartphone manufacturers must solve is the following: Provide small computer that uses relatively little power (so it can survive using only a battery for a while) and that interfaces with a rich set of sensors: It has to receive and decode the 4G (or whatever it's at when you're reading this) cellular signal, measure acceleration in various directions (to determine the phone's orientatin in space), take photos, receive input from a touch screen or from voice commands, communicate to other devices using Bluetooth, among many other possibilities.

1.10.2: Zooming in: Smoothing out noise

One problem with sensors is that their output, as fed to the phone's processor, is potentially slightly off. Various factors, from atmospheric pressure, environmental vibrations, gamma rays, and subtle unknown engineering mistakes, lead to the possibility of noise affecting each individual reading.

For example, the phone might be falling to the ground, straight down

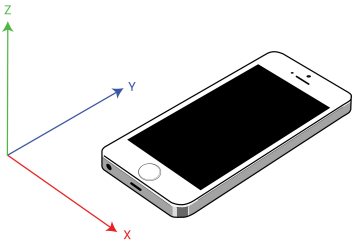

Figure 1.10.2.1: A phone together with the coordinate axes indicating the three directions in which acceleration can be measured by the phone's sensors.

However, because the phone was knocked off the table, it was vibrated a bit, and residual vibration affects causes the x-acceleration readings to vary between 1 and -1, say. Further, there is friction in the accelerometer, so while it is accelerating at g = 9.81 m/s^2 downward, the measurements actually bounce around this number in a random fashion.

Ultimately, the software in the phone needs just three numbers--'what is the current acceleration in each direction?'--so that it can display the right thing. But a single query to the accelerometer may give values that are randomly different from reality--sometimes significantly. So the trick is to take multiple readings in quick succession, say giving the following values:

```
x     y     z
-1    .5    -10.1
.2    -.1   -9.3
-.5   -.1   -10.2
```

```
.9    -.2   -9.5
```

```
x   -1      .2     -.5     .9
y   .5      -.1    -.1     -.2
z   -10.1   -9.3   -10.2   -9.5
```

and rather than taking any of these readings on their own as absolute truth, we suppose that each reading is off from reality by a random amount. For example, the 4 z readings are:

```
g - .3
g + .5
g - .4
g + .3
```

So if we assume they are as often a bit high as a bit low, then the highs and lows will cancel out if we add up all these values:

```
(g - .3) + (g + .5) + (g - .4) + (g + .3) = 4g + (-.3 + .5 - .4 + .3) = 4g + .1
```

But since we added up 4 values, each of which was close to the true acceleration of g, we get something that is instead close to 4*g. Thus if we instead of just adding them all, we average them--that is, we add them all and then divide by 4--we get an answer of g + .025, which is indeed pretty close to g.

So the algorithm for measuring acceleration will be instead of just reading once from the accelerometer, reading 4 times quickly, and then averaging the four readings, to hopefully get most of the noise to cancel.

Zooming in further, how would one compute the sum of two numbers in a computer? Let's say we have two wires coming in to a circuit, and if electricity is flowing through a wire, that will represent a 1, and if not, a 0. Coming out of this circuit should be a wire that will indicate the answer to the sum

```
input1 value + input2 value
```

except that this answer could be 0, 1, or possibly 2 if both input wires have electricity flowing through them. We cannot represent the answer using just one output wire, which could only give answers of 0 and 1, so we'll have two output wires--one of which will represent answers 0 and 1 if that's the answer, and a second wire for `the answer was 2'. So if both input wires are on, representing the sum is 1 + 1, the first output wire will be off, but the second `is the answer 2' output wire will be on.

We can summarize this approach in the following table:

```
input 1        off    off    on     on
input 2        off    on     off    on
actual sum     0      1      1      2
output 1       off    on     on     off
output 2       off    off    off    on
```

Now we have the challenge: make a circuit that, when it receives all the patterns of input wires, sends out the correct combination of output wires.

Once again, the components will be logical gates. output 2 seems relatively straightforward--it is controlled by a gate that will only turn on if both its inputs are on--i.e. an AND gate.

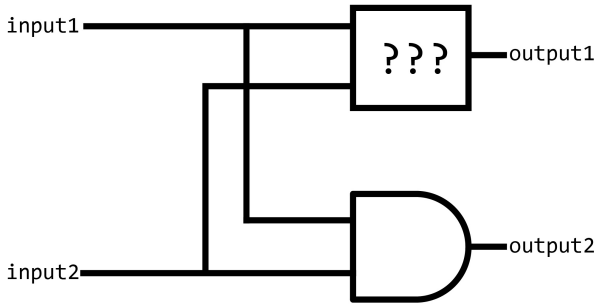

Figure 1.10.2.2: A first attempt at an addition circuit.

output 1 requires a slightly different sort of logic--it appears close to an OR gate, which turns on if either input 1 or input 2 is on. But an OR gate will also turn on its output if both inputs are on, which we don't want. The sort of gate we're after is called an 'exclusive-or' or 'XOR' gate--that is, an or gate that turns on only if input 1 is on, or if input 2 is on, but not if both are on.

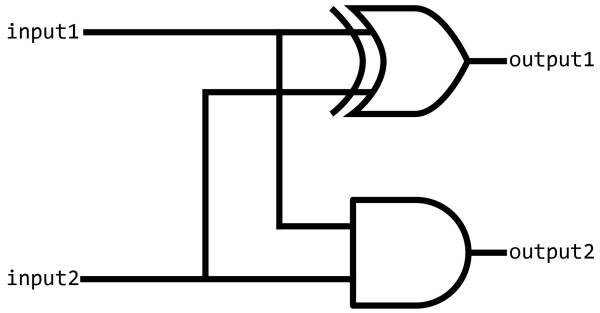

Figure 1.10.2.3: Our adding circuit for real

1.11: Outline of the remainder of the book

This chapter has been a whirlwind tour of some of the concepts of computer science. The big picture, however, remains: A computer is an machine that does arithmetic and can store and retrieve results of these computations from some memory. The organizing questions of this book are:

1. Once we have a computer, how do we make it perform all the functions we associate with computers?
2. Once we know that such a (seemingly) simple machine is sufficient for our purposes, how do we actually construct such a machine?

The distance in complexity between an game of Halo with several players in different countries and the moving around of electrons through wires is a rather intimidating gap, with several levels in between. Nevertheless, this is the gap we intend to span in this book, and we'll accomplish it by starting from the top stepping through the levels individually, at each level, black-boxing all the lower levels entirely.

Chapters 2 and 3 -- Programming a computer: high-level programming language

We'll start supposing we have a computer and that we also have a programming language that is able to translate somewhat human-friendly code into ISA operations, and we'll learn how to operate the programming language without reference to any lower-level details. The programming language of choice for us will be Python, and we will introduce it in two stages in chapters 2 and 3, respectively.

In chapter 2, we'll introduce the basics of the language, and then give some examples of its use in programming the actual applications we've discussed in the previous three sections. Because of the dearth of features introduced in chapter 2, these examples will be slightly cumbersome and unrealistic, but hopefully at least suggestive.

In chapter 3, we shall build on this foundation, adding in some more advanced language features that make the previous chapter's examples more believable and more functional, at the price of being slightly more complex.

Chapter 4 -- Numbers in computer science

Chapter 4 will be a bit of an interlude--it stands on its own and may be read at leisure any time before

chapters 5 and onward. It tackles the following issue: We will at this point understand that we want a machine that can do arithmetic. This means that a machine is going to have to deal with numbers. But a machine is also physical, whereas numbers are abstract. So we need some way to have an arrangement of physical objects--marbles, electrons, something--to represent numbers, and further how to perform operations like addition on these physical representations of numbers. The method we will use will be that of binary representations.

Chapter 5 -- Programming a computer: low-level programming language

Once we know how to use a programming language, we'll start to see how such a thing can be actually made by introducing a particular real computer's ISA--namely that of the the AVR microprocessor (this is the microprocessor that is used by Arduino chips, for example). This will be done in chapter 5. The ISA, as we have already seen in a few snippets above, will be more esoteric-looking. But in another sense it will be simpler, since in contrast to the whole suite of conceptually different operations available in Python, this language's operations are limited to precisely the three types we alluded to before: arithmetic, memory, and branching operations.

So the language will be simpler, but then it behooves us to see how something as featureful as Python can be constructed on top of this, so we'll revisit our three applications in this lower-level language and do some of the work of translating from Python to assembly to see that a machine that understands even just this low-level language is still capable of doing all the same things.

Chapter 6 -- Everything is numbers

Chapter 6 will be another interlude to discuss how, once we can represent numbers, we can represent anything else the computer might in theory want to store--text, images, and sounds, and low-level programs--using numbers.

Chapter 7 -- Building a computer: A bird's-eye view of a machine that can handle the low-level language

At this point, we'll have an idea of the basic low-level ISA instructions that the machine needs to be able to execute (from chapter 5). We know that these act on numbers, and we know how to actually store/manipulate numbers in a machine--namely, via their binary representations (from chapter 4). In chapter 7, then, we'll go through these instructions and build up a machine that can execute all of them, progressively adding on functionality for each new instruction as we go. At this stage, we won't be designing a full blueprint for the machine, but we'll black-box certain components like 'this is a module that can store numbers--here is its interface' or 'this is a module that can add two numbers' and build the machine out of these black-boxes.

Chapter 8 -- Building a computer: The pieces needed to build the machine of chapter 7

In chapter 8, finally, we'll finally get down to the level of the low-level components--called logic gates--that comprise all of the high-level pieces. We'll then see how to fill out the insides of the black boxes of chapter 6 using gates. You can actually purchase chips with just logic gates (4000-series integrated circuits), so at this point, you'll know enough to, in theory, build the machine we've been describing throughout this book.

One can purchase logic gates, so while we might stop there and say we have enough information to build a computer, logic gates are themselves built out of a single type of still lower-level component, namely a transistor. We also explain in chapter 8 what a transistor is, how one can chain transistors together to create logic gates, and how the transistor that go into the computer in front of you now are actually built.

1.12: Exercises

1.1: True or false: Mobile phones and game consoles are both computers.

1.2: What four things is the computer capable of?

1.3: Define a computer.

1.4: For each of the steps 1-5 in the algorithm written below, mention which among the four capabilities of a computer are required (A step may require more than one capability):

1. Take a number from the user and call it n1.
2. Take another number from the user and call it n2.
3. Add n1 and n2. Save the sum in a memory location. We will call this memory location total.
4. Divide total by 2. If you get 0 as a remainder, go to step v, else go to step vi.
5. Display “total is even” on the monitor.

1.5: Write an algorithm to determine whether the result of A / B should be rounded up, down, or has no remainder, if the rules for rounding are to round up if the remainder is .5 or greater and round down if it is less than .5.

Assume you are not able to use the modulus operator (%) and that A > B.

1.6: What is an algorithm?

1.7: What is the difference between a fixed-program device and a stored-program device? Is your basic pocket calculator a fixed-program computer or a stored-program computer?

1.8: According to the text, the ENIAC was:

1. Turing Complete
2. a fixed program device
3. a stored program device
4. A and C
5. A and B
6. none of the above

1.9: (Calculator required) Apply the square-root algorithm to x = 9. Iterate through the 4 steps one time. Keep track of the current s-value by writing it down at the end of each step.

1.10: Design a list of steps similar to the square-root algorithm that computes A*B and stores the result in C, where A and B are positive integers. Assume that your hardware is only capable of adding, comparing (>, <, >=, <=, =), and storing.

1.11: Now verify that your algorithm works by testing the values A=2 and B=3. Iterate through the steps until completion. Keep track of your B and C values by writing them down at the end of each step.

1.12: Define the property Turing Complete.

1.13: What is the fundamental difference between the ABC and the ENIAC?

1.14: Why is the following sequence of steps not a valid computer algorithm?

1. Store 0 in a memory location, and call the memory location sum.
2. Add 1 to sum, and save the result back in sum.
3. If sum is large enough, go to step 4, else go to step 3.
4. Display sum on the console.
5. Stop

1.15: True or false: Systems are more abstract than computers.

1.16: What is the difference between systems and computers?

1.17: Explain the difference between implementation and interface and give an example of both.

1.18: What is an abstraction?

(An abstraction, not just abstraction in general)

1.19: Draw and explain all levels of abstraction.

In figures 1.3.1.1 and 1.3.1.2, what do the large arrows connecting the customer to WaffleCo represent?

1.20: Describe the interface of a compiler.

1.21: What are the benefits of black-boxing?

1.22: Differentiate between the ISA and the microarchitecture.

1.23: Explain all the layers in a computer.

1.24:

What level of abstraction hides the details of the statements which follow? For example, the abstraction level of logic gates allows engineers not to think in terms of transistors.

1. A bunch of loads into memory, stores from memory, arithmetic computational instructions, and a branching instruction at the ISA level, is all contained in one single Python “if” statement.
2. A stored program computer consists of a Central Processing unit, I/O devices, and memory.
3. An ALU consists of loads of AND and OR gates.
4. A very common type of 2-input NAND gate consists of 4 transistors.

1.25: List and explain the significance of the different parts of the ISA.

1.26: Explain the difference between syntax and semantics.

1.27: Explain algorithmic complexity and how trade-offs are involved in choosing an algorithm.

1.28: Complexity accounts for:

1. degree of difficulty of math used in algorithm
2. memory used
3. time taken to write the algorithm
4. time to execute
5. b and d

1.29: How are syntax errors and runtime errors similar and how do they differ?

1.30:

According to the max(?, ?) function defined in this section, would the following incorrect usage be a syntax error or runtime error?

max(4 , “dog”)

1.31: What are programming languages and their syntax and semantics?

1.32: Explain the errors which might occur while programming.

1.33: Differentiate between high-level and low-level programming languages. Give examples of low-level languages.

1.34: Programming using a high-level language is much easier than using ISA instructions directly. But, what is a benefit in programming using ISA instructions directly?

1.35:

In some programming language, to display the natural logarithm of any number n, this is what we need to write: print (log(n))

A group of students were assigned to display log(5).Three different students wrote:

1. print (log(5))
2. print (log((5)
3. print (log(55))

Which student got the right answer? What were the type of errors experienced by

the other two students?

1.36: True or false:

1. If I plan to change the type of transistors I was using, I would have to change my ISA.
2. If I plan to use a different programming language, I would have to develop a new ISA.
3. If I disallow some instructions from my ISA, some Python instructions may stop working.

1.37: Companies tend to stick with very similar or the same ISA across generations of computers, even when there can be some benefit in performance if they changed it. Why do you think they do that?

1.38: What is a state machine?

1.39: Create a truth table for NOR, where the output is 0 when either of the inputs is 1 and the output is 1 otherwise.

1.40: Create a state machine that uses binary inputs of 0 and 1 and looks for four consecutive 1s and outputs “Found four 1s” when this occurs. Use only 0 and 1 for inputs at each state and write outputs of states inside the state “bubble” when creating the diagram.

1.41: Use figure 1.5.1.1 to answer the following: Beginning in the neutral state, which state will the cat be in after the following sequence of inputs is applied?

1. Ignore, ignore, ignore, pet, pet, pet
2. Ignore, ignore, pet, pet, pet
3. Feed, ignore, ignore, ignore, feed, ignore, ignore, pet

1.42: What is an encoding and how is it used by computers?

1.43: What is a Turing Machine?

1.44: Explain how abstraction works in the search engine example, specifying the individual components/teams involved.

1.45: All objects in a game are:

1. squares
2. circles
3. triangles

1.46: What are the two distinct components involved with programming a game and what are their main duties?

1.47: What are the two engines used in the software side of the game?

1.48: The decimal value 6 can be represented in binary as 0110, which is shown in the table in section in 1.9.4. Using the procedure for multiplying by 2 shown in figure 1.9.4.5, show and explain how 6*2 = 12 = 1100 in binary.

1.49: What are some sensors used in mobile phones?

1.50: What is noise, how does it affect phones, and explain how it is compensated for in phones, using both software and hardware input and calculation.

1.51: In section 1.5 we introduced the truth table for the AND gate. In this section, we define an OR gate as one that turns on (output is 1) if only input 1 is on, or if only input 2 is on, or if both are on. Based on this description, fill in the truth table for the OR gate:

Input 1	Input 2	Output

0	0	
0	1	
1	0	
1	1	

1.52: In this section, we define an XOR gate as one that turns on (output is 1) if input 1 or input 2 is on, but not if both are on. Based on this description, fill in the truth table for the XOR gate:

Input 1	Input 2	Output
0	0	
0	1	
1	0	
1	1	

1.53: The XOR name is shorthand for "Exclusive OR". How does this name so accurately represent the functionality of the XOR gate? (Hint: what exactly is the difference between OR and XOR? What is "exclusive" about the XOR functionality?)