Chapter 4: Computer Organization

Computers are useless. They can only give you answers.

—Pablo Picasso

4.1 Introduction to Computer Organization

Hey!

When we run a Python program, what's actually going on inside the computer? While we hope that recursion is feeling less like magic to you now, the fact that an electronic device can actually interpret and execute something as complicated as a recursive program may seem - what's the right word here? - *alien.* The goal of this chapter is to metaphorically pry the lid off a computer and peer inside to understand what's really going on there.

As you can imagine, a computer is a complicated thing. A modern computer has on the order of billions of transistors. It would be impossible to keep track of how all of those components interact. Consequently, computer scientists and engineers design and think about computers using what are called *multiple levels of abstraction* . At the lowest level, are components like transistors - the fundamental building blocks of modern electronic devices. Using transistors, we can build higher level devices called *logic gates* - they are the next level of abstraction. From logic gates we can build electronic devices that add, multiply, and do other basic operations - that's yet another level of abstraction. We keep moving up levels of abstraction, building more complex devices from more basic ones.

As a result, a computer can be designed by multiple people, each thinking about their specific level of abstraction. One type of expert might work on designing smaller, faster, and more efficient transistors. Another might work on using those transistors - never mind precisely how they work - to design better components that are based on transistors. Yet another expert will work on deciding how to organize these components into even more complex units that perform key computational functions. Each expert is grateful to be standing (metaphorically) on the shoulders of another person's work at the next lower level of abstraction.

By analogy, a builder thinks about building walls out of wood, nails, and sheet rock. An architect designs houses using walls without worrying too much about how they are built. A city planner thinks about designing cities out of houses, without thinking too much how they are built, and so forth. This idea is called “abstraction” because it allows us to think about a particular level of design using the lower level ideas abstractly; that is, without having to keep in mind all of the specific details of what happens at that lower level.

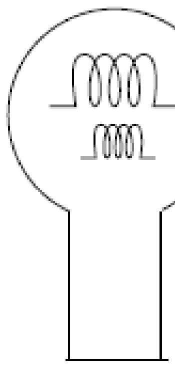

Figure 4.1: A three-way light bulb.

This seemingly simple idea of abstraction is one of the most important ideas in computer science. Not only are computers designed this way, but software is as well. Once we have basic workhorse functions like `map`, `reduce`, and others, we can use those to build more complex functions. We can use those more complex functions in many places to build even more complex software. In essence, we modularize so we that we can reuse good things to build bigger good things.

In this spirit, we’ll start by looking at how data is represented in a computer. Next, we’ll move up the levels of abstraction from transistors all the way up to a full-blown computer. We’ll program that computer in its own “native language” and talk about how your Python program ultimately gets translated to that language. By the end of this chapter, we’ll have a view of what happens when we run a program on our computer.

4.2 Representing Information

At the most fundamental level, a computer doesn’t really know math or have any notion of what it means to compute. Even when a machine adds 1+1 to get 2, it isn’t *really* dealing with numbers. Instead, it’s manipulating electricity according to specific rules.

To make those rules produce something that is useful to us, we need to associate the electrical signals inside the machine with the numbers and symbols that we, as humans, like to use.

4.2.1 Integers

The obvious way to relate electricity to numbers would be to assign a direct correspondence between voltage (or current) and numbers. For example, we could let zero volts represent the number 0, 1 volt be 1, 10 volts be 10, and so forth. There was a time when things were done this way, in so-called analog computers. But there are several problems with this approach, not the least of which would be the need to have a million-volt computer!

That's a shocking idea!

Here's another approach. Imagine that we use a light bulb to represent numbers. If the bulb is off, the number is 0. If the bulb is on, the number is 1. That's fine, but it only allows us to represent two numbers.

Whose bright idea was this?

All right then, let's upgrade to a "three-way" lamp. A three-way lamp really has four switch positions: off, and three increasing brightness levels. Internally, a three-way bulb has two filaments (Figure 4.1), one dim and one bright. For example, one might be 50 watts, and the other 100. By choosing neither of them, one, the other, or both, we can get 0, 50, 100, or 150 watts worth of light. We could use that bulb to represent the numbers 0, 50, 100, and 150 or we could decide that the four levels represent the numbers 0, 1, 2, and 3.

Internally, a computer uses this same idea to represent integers. Instead of using 50, 100, and 200, as in our illuminating example above, computers use combinations of numbers that are powers of 2. Let's imagine that we have a bulb with a 20-watt filament, a 21-watt filament, and a 22-watt filament. Then we could make 0 watts by turning none of the filaments on, we could make 1 watt by turning on only the 20-watt filament, we could make 2 watts by turning on the 21-watt filament, and so forth, up to $2^0 + 2^1 + 2^2 = 7$ watts using all three filaments.

Imagine now that we had the following four consecutive powers of 2 available to us: $2^0, 2^1, 2^2, 2^3$. Take a moment to try to write the numbers 0, 1, 2, and so forth as high as you can go by using zero or one of each of these powers of 2. Stop reading. We'll wait while you try this.

If all went well, you discovered that you could make all of the integers from 0 to 15 using 0 or 1 of each of these four powers of 2. For example, 13 can be represented as $2^0 + 2^2 + 2^3$. Written another way, this is:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Notice that we've written the larger powers of 2 on the left and the smaller powers of two on the right. This convention is useful, as we'll see shortly. The 0's and 1's in the equation above - the *coefficients* on the powers of 2 - indicate whether or not the particular power of 2 is being used. These 0 and 1 coefficients are called *bits*, which stands for *bi*nary digi*ts*. *Binary* means "using two values"- here, the 0 and the 1 are the two values.

There are 10 kinds of people in this world: Those who understand binary and those who don't.

It's convenient to use the sequence of bits to represent a number, without explicitly showing the powers of two. For example, we would use the bit sequence 1101 to represent the number 13 since that is the order in which the bits appear in the equation above. Similarly 0011 represents the number 3. We often leave off the leading (that is, the leftmost) 0's, so we might just write this as 11 instead.

The representation that we've been using here is called base 2 because it is built on powers of 2. Are other bases possible? Sure! You use base 10 every day. In base 10, numbers are made out of powers of 10 and rather than just using 0 and 1 as the coefficients, we use 0 through 9. For example, the sequence 603 really means

$$6 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$$

Other bases are also useful. For example, the Yuki Native American tribe, who lived in Northern California, used base 8. In base 8 we use powers of 8 and the coefficients that we use are 0 through 7. So, for example, the sequence 207 in base 8 means

$$2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0$$

which is 135 (in base 10). It is believed that the Yukis used base 8 because they counted using the eight slots *between* their fingers.

Notice that when we choose some base b (where b is some integer greater than or equal to 2), the digits that we use as coefficients are 0 through $b - 1$. Why? It's not hard to prove mathematically that when we use this convention, every positive integer between 0 and $bd - 1$ can be represented using d digits. Moreover, every integer in this range has a *unique* representation, which is handy since it avoids the headache of having multiple representations of the same number. For example, just as 42 has no other representation in base 10, the number 1101 in base 2 (which we saw a moment ago is 13 in base 10) has no other representation in base 2.

In Star Wars, the Hutts have 8 fingers and therefore also count in base 8.

Many older or smaller computers use 32 bits to represent a number in base 2; sensibly enough, we call them "32-bit computers." Therefore, we can uniquely represent all of the positive integers between 0 and 2^32 - 1 , or 4, 294, 967, 295. A powerful modern computer that uses 64 bits to represent each number can represent integers up to 2^64 - 1, which is roughly 18 *quadrillion*.

4.2.2 Arithmetic

Arithmetic in base 2, base 8, or base 42 is analogous to arithmetic in base 10. For example, let's consider addition. In base 10, we simply start at the rightmost column, where we add those digits and "carry" to the next column if necessary. For example, when adding $17 + 25$, $5 + 7 = 12$ so we write down a 2 in the rightmost position and carry the 1. That 1 represents a 10 and is therefore propagated, or carried, to the next column, which represents the "10's place."

We'll have sum fun with addition.

Addition in base 2 is nearly identical. Let's add 111 (which, you'll recall is 7 in base 10) and 110 (which is 6 in base 10). We start at the rightmost (or "least significant") column and add 1 to 0, giving 1. Now we move to the next column, which is the 2^1 position or "2's place". Each of our two numbers has a 1 in this position. 1 + 1 = 2 but we only use 0's and 1's in base 2, so in base 2 we would have 1 + 1 = 10. This is analogous to adding 7 + 3 in base 10: rather than writing 10, we write a 0 and carry the 1 to the next column. Similarly, in base 2, for 1 + 1 we write 0 and carry the 1 to the next column. Do you see why this works? Having a 2 in the "2's place" is the same thing as having a 1 in the "4's place". In general having a 2 in the column corresponding to 2^i is the same as having a 1 in the next column, the one corresponding to 2^{i+1} since $2 \cdot 2^i = 2^{i+1}$.

Although I find coming to agreement with myself to be easier in general.

Takeaway message: *Addition, subtraction, multiplication, and division in your favorite base are all analogous to those operations in base 10!*

4.2.3 Letters and Strings

We'll try not to get carried away with these examples, but you should try adding a few

As you know, computers don't only manipulate numbers; they also work with symbols, words, and documents. Now that we have ways to represent numbers as bits, we can use those numbers to represent other symbols.

numbers in base 2 to make sure that it makes sense to you.

It’s fairly easy to represent the alphabet numerically; we just need to come to an agreement, or “convention,” on the encoding. For example, we might decide that 1 should mean “A”, 2 should mean “B”, and so forth. Or we could represent “A” with 42 and “B” with 97; as long as we are working entirely within our own computer system it doesn’t really matter.

*In ASCII, the number 42 represents the asterisk (*)*

But if we want to send a document to a friend, it helps to have an agreement with more than just ourselves. Long ago, the American National Standards Institute (ANSI) published such an agreement, called ASCII (pronounced “as key” and standing for the American Standard Code for Information Interchange). It defined encodings for the upper- and lower-case letters, the numbers, and a selected set of special characters - which, not coincidentally, happen to be precisely the symbols printed on the keys of a standard U.S. keyboard.

Negative Thinking

We’ve successfully represented numbers in base 2 and done arithmetic with them. But all of our numbers were positive. How about representing negative numbers? And what about fractions?

Let’s start with negative numbers. One fairly obvious approach is to reserve one bit to indicate whether the number is positive or negative; for example, in a 32-bit computer we might use the leftmost bit for this purpose: Setting that bit to 0 could mean that the number represented by the remaining 31 bits is positive. If that leftmost bit is a 1 then the remaining number would be considered to be negative. This is called a *sign-magnitude* representation. The price we pay is that we lose half of our range (since we now have only 31 bits, in our example, to represent the magnitude of the number). While we don’t mean to be too negative here, a bigger problem is that it’s tricky to build computer circuits to manipulate sign-magnitude numbers. Instead, we use a system called two’s complement.

The idea behind two’s complement is this: It would be very convenient if the representation of a number plus the representation of its negation added up to 0. For example, since 3 added to -3 is 0, it would be nice if the binary representation of 3 plus the binary representation of -3 added up to 0. We already know what the binary representation of 3 is 11. Let’s imagine that we have an 8-bit computer (rather than 32- or 64-bit), just to make this example easier. Then, including the leading 0's, 3 would be represented as 00000011. Now, how could we represent -3 so that its representation added to 00000011 would be 0, that is 00000000?

Notice that if we "flip" all of the bits in the representation of 3, we get 11111100. Moreover, 00000011 + 11111100 = 11111111. If we add one more to this we get 11111111 + 00000001 and when we do the addition with carries we get 100000000; a 1 followed by eight 0's. If the computer is only using eight bits to represent each number then that leftmost ninth bit will not be recorded! So, what will be saved is just the lower eight bits, 00000000, which is 0. So, to represent -3, we can simply take the representation of 3, flip the bits, and then add 1 to that. (Try it out to make sure that you see how this works.) In general, the representation of a negative number in the two's complement system involves flipping the bits of the positive number and then adding 1.

You can look up the ASCII encoding on the web. Alternatively, you can use the Python function ord to find the numerical representation of any symbol. For example:

"ord" stands for "ordinal". You can think of this as asking for the "ordering number" of the symbol

```
>>> ord('*')
42
>>> ord('9')
57
```

Why is the ordinal value of '9' reported as 57? Keep in mind that the 9, in quotes, is just a character like the asterisk, a letter, or a punctuation symbol. It appears as character 57 in the ASCII convention. Incidentally, the inverse of ord is chr. Typing chr(42) will return an asterisk symbol and chr(57) will return the symbol '9'.

4 bits are sometimes called a "nybble"; we take no responsibility for this pathetically nerdy pun.

Each character in ASCII can be represented by 8 bits, a chunk commonly referred to as a "*byte*." Unfortunately, with only 8 bits ASCII can only represent 256 different symbols. (You might find it entertaining to pause here and write a short program that counts from 0 to 255 and, for each of these numbers, prints out the ASCII symbol corresponding to that number. You'll find that some of the symbols printed are "weird" or even invisible. Snoop on the Web to learn more about why this is so.)

Piecing It Together

How about representing fractions? One approach (often used in video and music players) is to establish a convention that everything is measured in units of some convenient fraction (just as our three-way bulb works in units of 50 watts). For example, we might decide that everything is in units of 0.01, so that the number 100111010 doesn't represent 314 but rather represents 3.14.

However, scientific computation often requires both more precision and a wider range of numbers than this strategy affords. For example, chemists often work with values as on the order of $10^2 3$ or more (Avogadro's number is approximately 6.02×10^{23}), while a nuclear physicist might use values as small as 10^{-12} or even smaller.

Imagine that we are operating in base 10 and we have only eight digits to represent our numbers. We might use the first six digits to represent a number, with the convention that there is an implicit 0 followed by a decimal point, just before the first digit. For example, the six digits 123456 would represent the number 0.123456. Then, the last two digits could be used to represent the exponent on the power of 10. So, 12345678 would represent 0.123456×10^{78} . Computers use a similar idea to represent fractional numbers, except that base 2 is used instead of base 10.

It may seem that 256 symbols is a lot, but it doesn't provide for accented characters used in languages like French (Français), let alone the Cyrillic or Sanskrit alphabets or the many thousands of symbols used in Chinese and Japanese.

To address that oversight, the International Standards Organization (ISO) eventually devised a system called Unicode, which can represent every character in every known language, with room for future growth. Because Unicode is somewhat wasteful of space for English documents, ISO also defined several "Unicode Transformation Formats" (UTF), the most popular of which is *UTF-8*. You may already be using UTF-8 on your computer, but we won't go into the gory details here.

There are even unofficial Unicode symbols for Klingon!

Of course, individual letters aren't very interesting. Humans normally like to string letters together, and we've seen that Python uses a data type called "strings" to do this. It's easy to do that with a sequence of numbers; for example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to "chocolate". The only detail missing is that when you are given a long string of numbers, you have to know when to stop; a common convention is to include a "length field" at the very beginning of the sequence. This number tells us how many characters are in the string. (Python uses a length field, but hides it from us to keep the string from appearing cluttered.)

4.2.4 Structured Information

Using the same concepts, we can represent almost any information as a sequence of numbers. For example, a picture can be represented as a sequence of colored dots, arranged in rows. Each colored dot (also known as a "picture element" or pixel) can be represented as three numbers giving the amount of red, green, and blue at that pixel. Similarly, a sound is a time sequence of "sound pressure levels" in the air. A movie is a more complex time sequence of single pictures, usually 24 or 30 per second, along with a matching sound sequence.

That's the level of abstraction thing again! Bits make up numbers, numbers make up pixels, pixels make up pictures, pictures make up movies. A two-hour movie can require several billion bits, but nobody who is making or watching a movie wants to think about all of those bits!

That would be more than a bit annoying!

4.3 Logic Circuitry

Now that we have adopted some conventions on the representation of data it's time to build devices that manipulate data. We'll start at a low level of abstraction of transistors and move up the metaphorical "food chain" to more complex devices, then units that can perform addition and other basic operations, and finally to a full-blown computer.

My favorite food chain sells donuts.

4.3.1 Boolean Algebra

In Chapter 2 we talked about Boolean variables - variables that take the value `True` or `False`. It turns out that Booleans are at the very heart of how a computer works.

As we noted in the last section, it's convenient to represent our data in base 2, also known as binary. The binary system has two digits, 0 and 1 just as Boolean variables have two values, `False` and `True`. In fact, we can think of 0 as corresponding to `False` and 1 as `True` as corresponding to `True`. The truth is, that Python thinks about it this way too. One funny way to see this is as follows:

```
>>> False + 42
42
>>> True + 2
3
```

Weird - but there it is: in Python `False` is really 0 and `True` is really 1. By the way, in many programming languages this is not the case. In fact, programming language designers have interesting debates about whether it's a good idea or not to have `False` and `True` be so directly associated with the numbers 0 and 1. On the one hand, that's how we often think about `False` and `True`. On the other hand, it can result in confusing expressions like `False + 42` which are hard to read and prone to introducing programmer errors.

We are writing these in upper-case letters to indicate that we are talking about operations on bits-0's and 1's - rather than Python's build in `and`, `or`, *and* `not`.

With the Booleans `True` and `False` we saw that we could use the operations and, or, and not to build up more interesting Boolean expressions. For example, `True` and `True` is the same as `True` while `True` or `False` is `True` and not `True` is `False`. Of course, we can now emulate these three operations for 0 and 1. 1 AND 1 = 1, 1 OR 0 = 1, and NOT 1 = 0.

Although your intuition of AND, OR, and NOT is probably fine, we can be very precise about these three operations by defining them with a *truth table* : a listing of all possible combinations of values of the input variables, together with the result produced by the function. For example, the truth table for AND is:

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

In Boolean notation, AND is normally represented as multiplication; an examination of the above table shows that as long as *x* and *y* are either 0 or 1, *x* AND *y* is in fact identical to multiplication. Therefore, we will often write *xy* to represent *x* AND *y*.

Takeaway message: AND *is 1 if and only if both of its arguments are 1.*

OR is a two-argument function that is 1 if either of its arguments are 1. The truth table for OR is:

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

OR is normally written using the plus sign: $x + y$. The first three lines of the above table are indeed identical to addition, but note that the fourth is different.

Takeaway message: OR *is 1 if either of its arguments is 1.*

Finally, NOT is a one-argument function that produces the opposite of its argument. The truth table is:

x	NOT x
0	1
1	0

NOT is normally written using an overbar, e.g. $\bar{x}$

4.3.2 Making Other Boolean Functions

Amazingly, any function of Boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT. No other operations are required because, as we'll see, any other operation could be made out of AND, OR, and NOT s. In this section we'll show how to do that. This fundamental result will allow us to build circuits to do things like arithmetic and, ultimately, a computer. For example, consider a function described by the truth table below. This function is known as "implication" and is written $x \Rightarrow y$.

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

This function can be expressed as $\bar{x} + xy$. To see why, try building the truth table for $\bar{x} + xy$. That is, for each of the four possible combinations of *x* and *y*, evaluate $\bar{x} + xy$. For example, when $x = 0$ and $y = 0$, notice that $\bar{x}$ is 1. Since the OR of 1 and anything else is always 1, we see that $\bar{x} + xy$ evaluates to 1 in this case. Aha! This is exactly the value that we got in the truth table above for $x = 0$ and $y = 0$. If you continue doing this for the next three rows, you'll see that values of $x \Rightarrow y$ and $\bar{x} + xy$ always match. In other words, they are identical. This method of enumerating the output for each possible input is a fool-proof way of proving that two functions are identical, even if it is a bit laborious.

For simple Boolean functions, it's often possible to invent an expression for the function by just inspecting the truth table. However, it's not always so easy to do this, particularly when we have Boolean functions with more than two inputs. So, it would be nice to have a systematic approach for building expressions from truth tables. The *minterm expansion principle* provides us with just such an approach.

Perhaps "minterms" should be called "happyterms".

We'll see how the minterm expansion principle works through an example. Specifically, let's try it out for the truth table for the implication function above. Notice that when the inputs are $x = 1$ and $y = 0$, the truth table tells us that the output is 0. However, for all three of the other rows (that is pairs of inputs), the output is more "interesting" - it's 1. We'll build a custom-made logical expression for each of these rows with a 1 as the output. First, consider the row $x = 0$; $y = 0$. Note that the expression $\bar{x}\bar{y}$ evaluates to 1 for this particular pair of inputs because the NOT of 0 is 1 and the AND of 1 and 1 is 1. Moreover, notice that for every other possible pair of values for x and y this term $\bar{x}\bar{y}$ evaluates to 0. Do you see why? The only way that $\bar{x}\bar{y}$ can evaluate to 1 is for $\bar{x}$ to be 1 (and thus for x to be 0) and for $\bar{y}$ to be 1 (since we are computing the AND here and AND outputs 1 only if both of its inputs are 1). The term $\bar{x}\bar{y}$ is called a minterm. You can think of it as being custom-made to make the inputs $x = 0$; $y = 0$ "happy" (evaluate to 1) and does nothing for every other pair of inputs.

We're not done yet! We now want a minterm that is custom-made to evaluate to 1 for the input $x = 0; y = 1$ and evaluates to 0 for every other pair of input values. Take a moment to try to write such a minterm. It's $\bar{x}y$. This term evaluates to 1 if and only if $x = 0$ and $y = 1$. Similarly, a minterm for $x = 1; y = 1$ is math:*xy*. Now that we have these minterms, one for each row in the truth table that contains a 1 as output, what do we do next? Notice that in our example, our function should output a 1 if the first minterm evaluates to 1 or the second minterm evaluates to 1 or the third minterm evaluates to 1. Also notice the words “or” in that sentence. We want to `OR` the values of these three minterms together. This gives us the expression $\bar{x}\bar{y} + \bar{x}y + xy$. This expression evaluates to 1 for the first, second, and fourth rows of the truth table as it should. How about the third row, the “uninteresting” case where $x = 1; y = 1$ should output 0. Recall that each of the minterms in our expression was custom-made to make exactly one pattern “happy”. So, none of these terms will make the $x = 1; y = 1$ “happy” and thus, for that pair of inputs, our newly minted expression outputs a 0 as it should!

It’s not hard to see that this minterm expansion principle works for every truth table. Here is the precise description of the process:

1. Write down the truth table for the Boolean function that you are considering.
2. Delete all rows from the truth table where the value of the function is 0.
3. For each remaining row we will create something called a “minterm” as follows:
    a. For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to `NOT` it.
    b. Now `AND` all of these variables together.
4. Combine all of the minterms for the rows using `OR` .

You might have noticed that this general algorithm for converting truth tables to logic expressions only uses `AND`, `OR`, and `NOT` operations. It uses `NOT` and `AND` to construct each minterm and then it uses `OR` to “glue” these minterms together. This effectively proves that `AND`, `OR`, and `NOT` suffice to represent any Boolean function!

The minterm expansion principle is a recipe - it’s an *algorithm*. In fact, it can be implemented on a computer to automatically construct a logical expression for any truth table. In practice, this process is generally done by computers. Notice, however that this algorithm doesn’t necessarily give us the simplest expression possible. For example, for the implication function, we saw that the expression $\bar{x} + xy$ is correct. However, the minterm expansion principle produced the expression $\bar{x}\bar{y} + \bar{x}y + xy$. These expressions are logically equivalent, but the first one is undeniably shorter. Regrettably, the so-called “minimum equivalent expressions” problem of finding the shortest expression for a Boolean function is very hard. In fact, a Harvey Mudd College graduate, David Buchfuhrer, recently showed that the minimum equivalent expressions problem is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science. Amazing but true!

4.3.3 Logic Using Electrical Circuits

That means that computers are helping design other computers! That seems profoundly amazing to me.

Next, we'd like to be able to perform Boolean functions in hardware. Let's imagine that our basic "building block" is an electromagnetic switch as shown in Figure 4.2. There is *always* power supplied to the switch (as shown in the upper left) and there is a spring that holds a movable "arm" in the up position. So, normally, there is no power going to the wire labeled "output". The "user's" input is indicated by the wire labeled "input." When the input power is off (or "low"), the electromagnet is not activated and the movable arm remains up, and output is 0. When the input is on (or "high"), the electromagnet is activated, causing the movable arm to swing downwards and power to flow to the output wire. Let's agree that a "low" electrical signal corresponds to the number 0 and a "high" electrical signal corresponds to the number 1. Now, let's build a device for computing the AND function using switches. We can do this as shown in Figure 4.3 where there are two switches in series. In this figure, we use a simple representation of the switch without the power or the ground. The inputs are x and y, so when x is 1, the arm of the first switch swings down and closes the switch, allowing power to flow from the left to the right. Similarly, when y is 1, that arm of the second switch swings down, allowing power to flow from left to right. Notice that when either or both of the input x, y are 0, at least one switch remains open and there is no electrical signal flowing from the power source to the output. However, when both x *and* y are 1 both switches close and there is a signal, that is a 1, flowing to the output. This is a device for computing x AND y . We call this an AND gate.

Similarly, the circuit in Figure 4.4 computes x OR y and is called an OR gate. The function NOT x can be implemented by constructing a switch that conducts if and only x is 0.

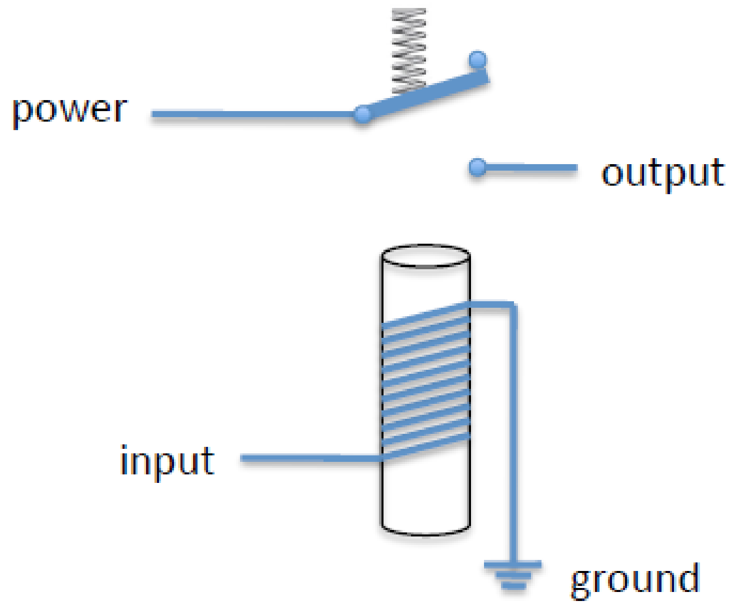

Figure 4.2: An electromagnetic switch.

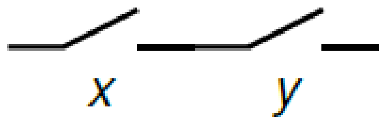

Figure 4.3: An AND gate constructed with switches.

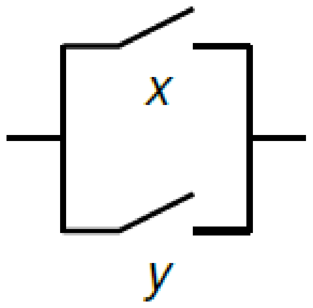

Figure 4.4: An OR gate constructed with switches.

While computers based on electromechanical switches were state-of-the-art in the 1930's, computers today are built with transistorized switches that use the same principles but are much smaller, much faster, more reliable, and more efficient. Since the details of the switches aren't terribly important at this level of abstraction, we

Those gate shapes

represent, or “abstract”, the gates using symbols as shown in Figure 4.5

are weird. I’m on the fence about whether I like them or not.

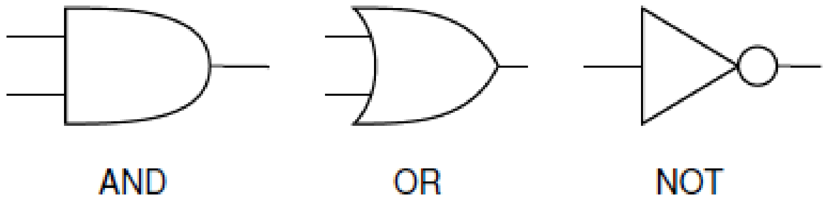

Figure 4.5: Symbols used for `AND`, `OR`, and `NOT` gates.

We can now build circuits for any Boolean function! Starting with the truth table, we use the minterm expansion principle to find an expression for the function. For example, we used the minterm expansion principle to construct the expression $\bar{x}\bar{y} + \bar{x}y + xy$ for the implication function. We can convert this into a circuit using `AND`, `OR`, and `NOT` gates as shown in Figure 4.6.

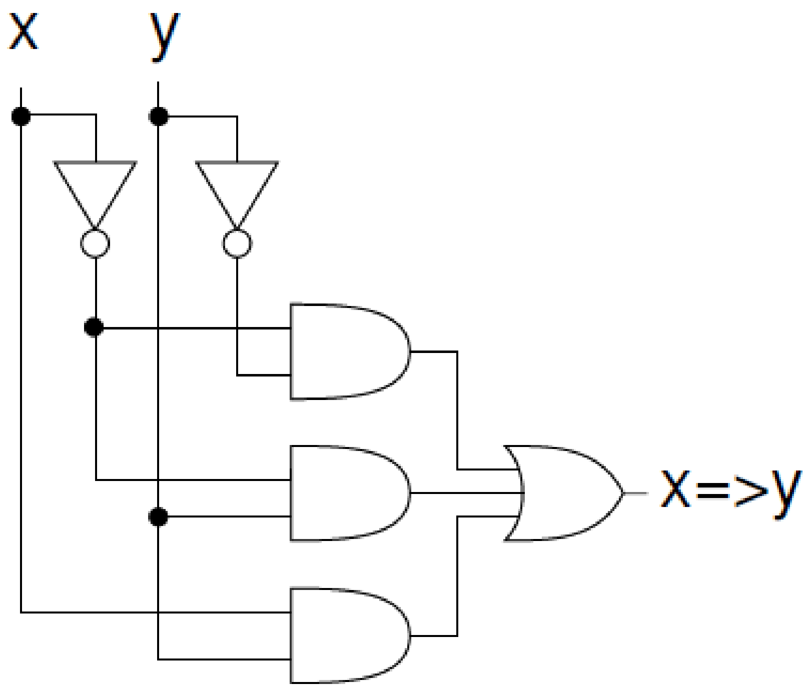

Figure 4.6: A circuit for the implication function.

4.3.4 Computing With Logic

Now that we know how to implement functions of Boolean variables, let’s move up one more level of abstraction and try to build some units that do arithmetic. In binary, the numbers from

0 to 3 are represented as 00, 01, 10, and 11. We can use a simple truth table to describe how to add two two-bit numbers to get a three-bit result:

x	y	$x + y$
00	00	000
00	01	001
00	10	010
⋮		⋮
01	10	011
01	11	100
⋮		⋮
11	11	110

In all, this truth table contains sixteen rows. But how can we apply the minterm expansion principle to it? The trick is to view it as three tables, one for each bit of the output. We can write down the table for the rightmost output bit separately, and then create a circuit to compute that output bit. Next, we can repeat this process for the middle output bit. Finally, we can do this one more time for the leftmost output bit. While this works, it is much more complicated than we would like! If we use this technique to add two 16-bit numbers, there will be 2^{32} rows in our truth table resulting in several *billion* gates.

Ouch!

Fortunately, there is a much better way of doing business. Remember that two numbers are added (in any base) by first adding the digits in the rightmost column. Then, we add the digits in the next column and so forth, proceeding from one column to the next, until we're done. Of course, we may also need to carry a digit from one column to the next as we add.

We can exploit this addition algorithm by building a relatively simple circuit that does just one column of addition. Such a device is called a *full adder*, (admittedly a funny name given that it's only doing one column of addition!). Then we can "chain" 16 full adders together to add two 16-bit numbers or chain 64 copies of this device together if we want to add two 64-bit numbers. The resulting circuit, called a *ripple-carry adder*, will be much simpler and smaller than the first approach that we suggested above. This modular approach allows us to first design a component of intermediate complexity (e.g. the full adder) and use that design to design a more complex device (e.g. a 16-bit adder). Aha! Abstraction again!

The full adder takes three inputs: The two digits being added in this column (we'll call them x and y) and the "carry in" value that was propagated from the previous column (we'll call that c_{in}). There will be two outputs: The sum (we'll call that z) and the "carry out" to be propagated to the next column (we'll call that c_{out}). We suggest that you pause here and build the truth table for this function. Since there are three inputs, there will be $2^3 = 8$ rows in the truth table. There will be two columns of output. Now, treat each of these two output columns as a separate function. Starting with the first column of output, the sum z, use the minterm expansion principle to write a logic expression for z. Then, convert this expression into a circuit using `AND`, `OR`, and `NOT` gates. Then, repeat this process for the second column of output, c_{out}. Now you have a full adder! Count the gates in this adder - it's not a very big number.

Finally, we can represent this full adder abstractly with a box that has the three inputs on top and the two outputs on the bottom. We now chain these together to build our ripple-carry adder. A 2-bit ripple-carry adder is shown in Figure 4.7. How many gates would be used in total for a 16-bit ripple-carry adder? It's in the hundreds rather than the billions required in our first approach!

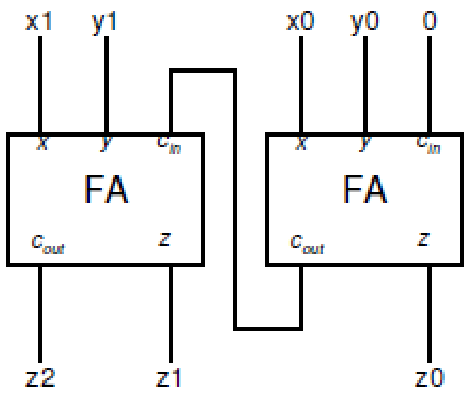

Figure 4.7: A 2-bit ripple-carry adder. Each box labeled "FA" is a full adder that accepts two input bits plus a carry, and produces a single output bit along with a carry into the next FA.

Now that we've built a ripple-carry adder, it's not a big stretch to build up many of the other kinds of basic features of a computer. For example, consider building a multiplication circuit. We can observe that multiplication involves a number of addition steps. Now that we have an addition module, that is an abstraction that we can use to build a multiplier!

Take Away *Using the minterm expansion principle and modular design, we can now build virtually all of the major parts of a computer.*

4.3.5 Memory

There is one important aspect of a computer that we haven't seen how to design: memory! A computer can store data and then fetch those data for later use.("Data" is the plural of "datum". Therefore, we say "those data" rather than "that data".) In this section we'll see how to build a circuit that stores a single bit (a 0 or 1). This device is called a *latch* because it allows us to "lock" a bit and retrieve it later. Once we've built a latch, we can abstract that into a "black box" and use the principle of modular design to assemble many latches into a device that stores a lot of data.

I'm glad that you didn't forget this part!

A latch can be created from two interconnected NOR gates: NOR is just OR followed by NOT. That is, its truth table is exactly the opposite of the truth table for OR as shown below.

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

A NOR gate is represented symbolically as an OR gate with a little circle at its output (representing negation).

Now, a latch can be constructed from two NOR gates as shown in Figure 4.8. The input S is known as "set" while the input R is known as "reset". The appropriateness of these names will become evident in a moment.

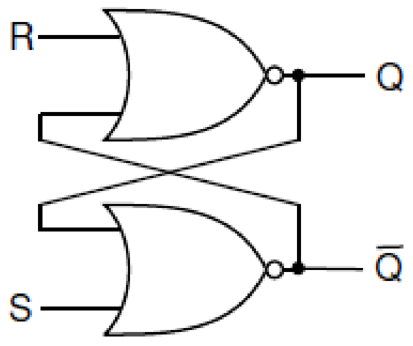

Figure 4.8: A latch built from two NOR gates.

What in the world is going on in this circuit!? To begin with, suppose that all of R, S, and Q are 0. Since both Q and S are 0, the output of the bottom NOR gate, $\bar{Q}$, is 1. But since $\bar{Q}$ is 1, the top NOR gate is forced to produce a 0. Thus, the latch is in a stable state: the fact that $\bar{Q}$ is 1 holds Q at 0.

Now, consider what happens if we change S (remember, it's called "set") to 1, while holding R at 0. This change forces $\bar{Q}$ to 0; for a moment, both Q and $\bar{Q}$ are zero. But the fact that

$\bar{Q}$ is zero means that both inputs to the top NOR gate are zero, so its output, Q, must become 1. After that happens, we can return S to 0, and the latch will remain stable. We can think of the effect of changing S to 1 for a moment as "setting" the latch to store the value 1. The value is stored at Q. ($\bar{Q}$ is just used to make this circuit do its job, but it's the value of Q that we will be interested in.)

An identical argument will show that R (remember, it's called "reset") will cause Q to return to zero, and thus $\bar{Q}$ will become 1 again. That is, the value of Q is reset to 0! This circuit is commonly called the *S-R latch*.

What happens if both S and R become 1 at the same time? Try this out through a thought experiment. We'll pause here and wait for you.

Did you notice how this latch will misbehave? When S and R are both set to 1 (this is trying to set and reset the circuit simultaneously—very naughty) both Q and $\bar{Q}$ will become 0. Now, if we let S and R return back to 0, the inputs to both NOR gates are 0 and their outputs both become 1. Now each NOR gate gets a 1 back as input and its output becomes 0. In other words, the NOR gates are rapidly "flickering" between 0 and 1 and not storing anything! In fact, other weird and unpredictable things can happen if the two NOR gates compute their outputs at slightly different speeds. Circuit designers have found ways to avoid this problem by building some "protective" circuitry that ensures that S and R can never be simultaneously set to 1.

So, a latch is a one-bit memory. If you want to remember a 1, turn S to 1 for a moment; if you want to remember a 0, turn R to 1 for a moment. If you aggregate 8 latches together, you can remember an 8-bit byte. If you aggregate millions of bits, organized in groups of 8, you have the *Random Access Memory (RAM)* that forms the memory of a computer.

I'm sheepish about sharing my RAM puns because you've probably herd them already.

4.4 Building a Complete Computer

Imagine that you've been elected treasurer of your school's unicycling club and its time to do the books and balance the budget. You have a large notebook with all of the club's finances. As you work, you copy a few numbers from the binder onto a scratch sheet, do some computations using a calculator, and jot those results down on your scratch sheet. Occasionally, you might copy some of

"Balancing" the club's budget?! We promise that wheel have no more unicycle jokes!

those results back into the big notebook to save for the future and then jot some more numbers from the notebook onto your scratch sheet.

A modern computer operates on the same principle. Your calculator and the scratch sheet correspond to the *CPU* of the computer. The CPU is where computation is performed but there's not enough memory there to store all of the data that you will need. The big notebook corresponds to the computer's memory. These two parts of the computer are physically separate components that are connected by wires on your computer's circuit board.

We spoke too soon about no more unicycle jokes.

What's in the CPU? There are devices like ripple-carry adders, multipliers, and their ilk for doing arithmetic. These devices can all be built using the concepts that we saw earlier in this chapter, namely the minterm expansion principle and modular design. The CPU also has a small amount of memory, corresponding to the scratch sheet. This memory comprises a small number of *registers* where we can store data. These registers could be built out of latches or other related devices. Computers typically have on the order of 16 to 32 of these registers, each of which can store 32 or 64 bits. All of the CPU's arithmetic is done using values in the registers. That is, the adders, multipliers, and so forth expect to get their inputs from registers and to save the results to a register, just as you would expect to use your scratch pad for the input and output of your computations.

The memory, corresponding to the big notebook, can store a lot of data - probably billions of bits! When the CPU needs data that is not currently stored in a register (scratch pad), it requests that data from memory. Similarly, when the CPU needs to store the contents of a register (perhaps because it needs to use that register to store some other values), it can ship it off to be stored in memory.

See! We didn't tire you with any more unicycle puns.

What's the point of having separate registers and memory? Why not just have all the memory in the CPU? The answer is multifaceted, but here is part of it: The CPU needs to be small in order to be fast. Transmitting a bit of data along a millimeter of wire slows the computer down considerably! On the other hand, the memory needs to be large in order to store lots of data. Putting a large memory in the CPU would make the CPU slow. In addition, there are other considerations that necessitate separating the CPU from the memory. For example, CPUs are built using different (and generally much more expensive) manufacturing processes than memories.

Now, let's complicate the picture slightly. Imagine that the process

Memory is slow. If

of balancing the unicycle club's budget is complicated. The steps required to do the finances involve making decisions along the way (e.g. "If we spent more than $500 on unicycle seats this year, we are eligible for a rebate.") and other complications. So, there is a long sequence of instructions that is written in the first few pages of our club notebook. This set of instructions is a program! Since the program is too long and complicated for you to remember, you copy the instructions one-by-one from the notebook onto your scratch sheet. You follow that instruction, which might tell you, for example, to add some numbers and store them some place. Then, you fetch the next instruction from the notebook.

the CPU can read from or write to a register in one unit of time, it will take approximately 100 units of time to read from or write to memory!

How do you remember which instruction to fetch next and how do you remember the instruction itself? The CPU of a computer has two special registers just for this purpose. A register called the *program counter* keeps track of the location in memory where it will find the next instruction. That instruction is then fetched from memory and stored in a special register called the *instruction register*. The computer examines the contents of the instruction register, executes that instruction, and then increments the program counter so that it will now fetch the next instruction.

John von Neumann (1903-1957)

One of the great pioneers of computing was John von Neumann(pronounced "NOY-mahn"), a Hungarian-born mathematician who contributed to fields as diverse as set theory and nuclear physics. He invented the Monte Carlo method (which we used to calculate π in Section 1.1.2), cellular automata, the *merge sort* method for sorting, and of course the von Neumann architecture for computers.

Although von Neumann was famous for wearing a three-piece suit everywhere—including on a mule trip in the Grand Canyon and even on the tennis court—he was not a boring person. His parties were always popular (although he sometimes sneaked away from his guests to work) and he loved to quote from his voluminous memory of off-color limericks. Despite his brilliance, he was a notoriously bad driver, which might explain why he bought a new car every year.

Von Neumann died of cancer, perhaps caused by radiation from atomic-bomb tests. But his legacy lives on in every computer built today.

This way of organizing computation was invented by the famous mathematician and physicist, Dr. John von Neumann and is known as the *von Neumann architecture*. While computers differ in all kinds of ways, they all use this fundamental principle. In the next subsection we'll look more closely at how this principle is used in a real computer.

4.4.1 The von Neumann Architecture

One of von Neumann's colleagues was Dr. Claude Shannon—inventor of the minterm expansion principle. Shannon, it turns out, was also a very good unicyclist.

We mentioned that the computer's memory stores both instructions and data. We know that data can be encoded as numbers and numbers can be encoded in binary. But what about the instructions? Good news! Instructions too can be stored as numbers by simply adopting some convention that maps instructions to numbers.

For example, let's assume that our computer is based on 8-bit numbers and let's assume that our computer only has four instructions: add, subtract, multiply, and divide. (That's very few instructions, but let's start there for now and then expand later). Each of these instructions will need a number, called an *operation code* (or *opcode*), to represent it. Since there are four opcodes, we'll need four numbers, which means two bits per number. For example, we might choose the following opcodes for our instructions:

Operation	Meaning
00	Add
01	Subtract
10	Multiply
11	Divide

Next, let's assume that our computer has four registers number 0 through 3. Imagine that we want to add two numbers. We must specify the two registers whose values we wish to add and the register where we wish to store the result. If we want to add the contents of register 2 with the contents of register 0 and store the result in register 3, we could adopt the convention that we'll write "add 3, 0, 2". The last two numbers are the registers where we'll get our input and the first number is the register where we'll store our result. In binary, "add 3, 0, 2" would be represented as "00 11 00 10". We've added the spaces to help you see the numbers 00 (indicating "add"), 11 (indicating the register where the result will be stored), 00 (indicating register 0 as the first register that we will add), and 10 (indicating register 2 as the second register that we will add).

In general, we can establish the convention that an instruction will be

Computer scientists

encoded using 8 bits as follows: The first two bits (which we'll call I0 and I1) represent the instruction, the next two bits (D0 and D1) encode the "destination register" where our result will be stored, the next two bits (S0 and S1) encode the first register that we will add, and the last two bits (T0 and T1) encode the second register that we will add. This representation is shown below.

often start counting from 0.

I0 I1	D0 D1	S0 S1	T0 T1

Recall that we assume that our computer is based on 8-bit numbers. That is, each register stores 8 bits and each number in memory is 8 bits long. Figure 4.9 shows what our computer might look like. Notice the program counter at the top of the CPU. Recall that this register contains a number that tells us where in memory to fetch the next instruction. At the moment, this program counter is 00000000, indicating address 0 in memory.

The computer begins by going to this memory location and fetching the data that resides there. Now look at the memory, shown on the right side of the figure. The memory addresses are given both in binary (base 2) and in base 10. Memory location 0 contains the data 00100010. This 8-bit sequence is now brought into the CPU and stored in the instruction register. The CPU's logic gates decode this instruction. The leading 00 indicates it's an addition instruction. The following 10 indicates that the result of the addition that we're about to perform will be stored in register 2. The next 00 and 10 mean that we'll get the data to add from registers 0 and 2, respectively. These values are then sent to the CPU's ripple-carry adder where they are added. Since registers 0 and 2 contain 00000101 and 00001010, respectively, before the operation, after the operation register 2 will contain the value 00001111.

So a computer is kind of like a dog?

In general, our computer operates by repeatedly performing the following procedure:

1. Send the address in the program counter (commonly called the *PC*) to the memory, asking it to read that location.
2. Load the value from memory into the instruction register.
3. *Decode* the instruction register to determine what instruction to execute and which registers to use.
4. *Execute* the requested instruction. This step often involves reading operands from registers, performing arithmetic, and sending the results back to the destination register. Doing so usually involves several sub-steps.
5. Increment the PC (Program Counter) so that it contains the address of the next

instruction in memory. (It is this step that gives the PC its name, because it *counts* its way through the addresses in the program.)

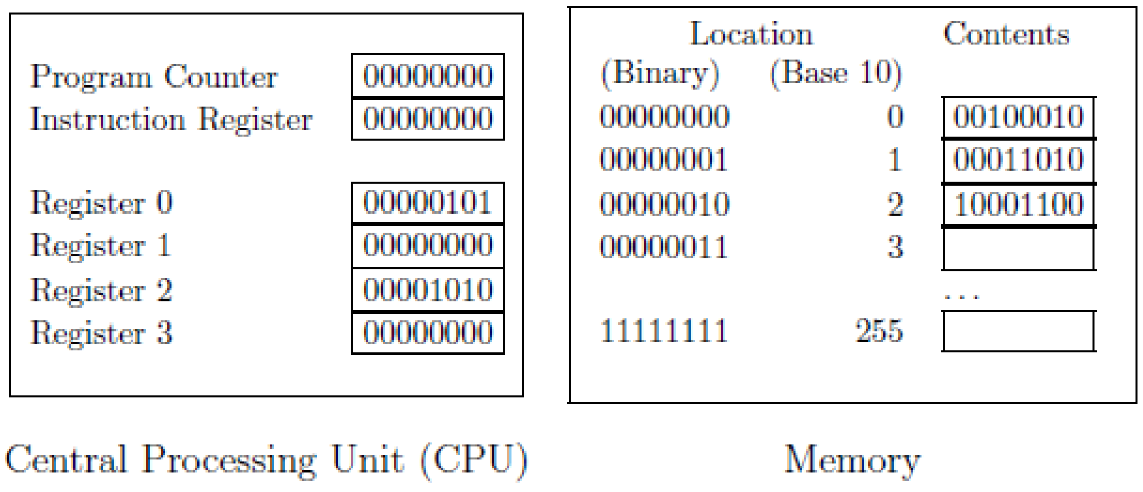

Figure 4.9: A computer with instructions stored in memory. The program counter tells the computer where to get the next instruction.

“Wait!” we hear you scream. “The memory is storing *both* instructions *and* data! How can it tell which is which?!” That’s a great question and we’re glad you asked. The truth is that the computer *can’t tell* which is which. If we’re not careful, the computer might fetch something from memory into its instruction register and try to execute it when, in fact, that 8-bit number represents the number of pizzas that the unicycle club purchased and not an instruction! One way to deal with this is to have an additional special instruction called “halt” that tells the computer to stop fetching instructions. In the next subsections we’ll expand our computer to have more instructions (including “halt”) and more registers and we’ll write some real programs in the language of the computer.

Takeaway message: *A computer uses a simple process of repeatedly fetching its next instruction from memory, decoding that instruction, executing that instruction, and incrementing the program counter. All of these steps are implemented with digital circuits that, ultimately, can be built using the processes that we’ve described earlier in this chapter.*

4.5 Hmmm

The computer we discussed in Section 4.4 is simple to understand, but its simplicity means it’s not very useful. In particular, a real computer also needs (at a minimum) ways to:

1. Move information between the registers and a large memory,

2. Get data from the outside world,
3. Print results, and
4. Make decisions.

To illustrate how these features can be included, we have designed the Harvey Mudd Miniature Machine, or Hmmm. Just like our earlier 4-instruction computer, Hmmm has a program counter, an instruction register, a set of data registers, and a memory. These are organized as follows:

1. While our simple computer used 8-bit instructions, in Hmmm, both instructions and data are 16 bits wide. The set of bits representing an instruction is called a *word*. That allows us to represent a reasonable range of numbers, and lets the instructions be more complicated.
2. In addition to the program counter and instruction register, Hmmm has 16 registers, named R0 through R15. R0 is special: it always contains zero, and anything you try to store there is thrown away.
3. Hmmm's memory contains 256 locations. Although the memory can be used for either instructions or data, programs are prevented from reading or writing the instruction section of memory. (Some modern computers offer a similar feature.)

Hmmm is music to my ears!

Different computers have different word sizes. Most machines sold today use 64-bit words; older ones use 32 bits. 16-bit, 8-bit, and even 4-bit computers are still used for special applications. Your digital watch probably contains a 4-bit computer.

Since the Hmmm instruction format has it's inconvenient to try to program Hmmm by writing down the bits corresponding to the instructions. Instead, we will use *assembly language*, which is a programming language where each machine instruction receives a friendlier symbolic representation. For example, to compute R3 = R1 + R2, we would write:

```
add r3, r1, r2
```

Hmmm's instruction set is large, but not hmmmungous.

A very simple process is used to convert this assembly language into the 0's and 1's - the "machine language" - that the computer can execute.

A complete list of Hmmm instructions, including their binary encoding, is given in Figure 4.10.

Is there a good mnemonic for

remembering how to spell "mnemonic"?

Not the least of which is that writing even a short program in assembly language can be a hmmmbling experience!

4.5.1 A Simple Hmmm Program

To begin with, let's look at a program that will calculate the approximate area of a triangle. It's admittedly mundane, but it will illustrate (We suggest that you follow along by downloading Hmmm from http://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html and trying these examples out with us.)

Let's begin by using our favorite editor to create a file named `triangle1.hmmm` with the following contents:

```
#
# Calculate the approximate area of a triangle.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1       # Get base
1       read    r2       # Get height
2       mul     r1 r1 r2 # b times h into r1
3       setn    r2 2
4       div     r1 r1 r2 # Divide by 2
5       write   r1
6       halt
```

How Does It Work?

What does all of this mean? First, anything starting with a pound sign ("#") is a comment; Hmmm ignores it. Second, you'll note that each line is numbered, starting with zero. This number indicates the memory location where the instruction will be stored.

You may have also noticed that this program doesn't use any commas, unlike the example `add` instruction above. Hmmm is very lenient about notation; all of the following instructions mean exactly the same thing:

```
add r1,r2,r3

ADD R1 R2 R3

ADD R1,r2, R3

aDd R1,,R2,       ,R3
```

Needless to say, we don’t recommend the last two options!

So what do all of these lines actually do? The first two (0 and 1) read in the base and height of the triangle. When Hmmm executes a `read` instruction, it pauses and prompts the user to enter a number, converts the user’s number into binary, and stores it into the named register. So the first-typed number will go into register R1, and the second into R2.

The `MUL`tiply instruction then finds $b \times h$ by calculating R1 = R1 $\times$ R2. This instruction illustrates three important principles of hmmm programming: 1. Most arithmetic instructions accept three registers: two *sources* and a *destination*. 2. The destination register is always listed first, so that the instruction can be read somewhat like a Python assignment statement. 3. A source and destination can be the same.

After multiplying, we need to divide $b \times h$ by 2. But where can we get the constant 2? One option would be to ask the user to provide it via a `read` instruction, but that seems clumsy. Instead, a special instruction, `setn` (*set to number*), lets us insert a small constant into a register. As with `mul`, the destination is given first; this is equivalent to the Python statement R2 = 2.

The `DIV`ide instruction finishes the calculation, and `write` displays the result on the screen. There is one thing left to do, though: after the `write` is finished, the computer will happily try to execute the instruction at the following memory location. Since there isn’t a valid instruction there, the computer will fetch a collection of bits there that are likely to be invalid as an instruction, causing the computer to crash. So we need to tell it to `halt` after it’s done with its work.

That’s it! But will our program work

Trying It Out

It would be hmmmurous if we

We can *assemble* the program by running `hmmmAssembler.py` from the command line: [1] User-typed input is shown in blue and the prompt is shown using the symbol `%`. The prompt on your computer may look different.

got this wrong

```
% ./hmmmAssembler.py
Enter input file name: triangle1.hmmm
Enter output file name: triangle1.hb

----------------------
| ASSEMBLY SUCCESSFUL |
----------------------

0 : 0000 0001 0000 0001        0        read    r1        # Get base
1 : 0000 0010 0000 0001        1        read    r2        # Get height
2 : 1000 0001 0001 0010        2        mul     r1 r1 r2  # b times h into r1
3 : 0001 0010 0000 0010        3        setn    r2 2
4 : 1001 0001 0001 0010        4        div     r1 r1 r2  # Divide by 2
5 : 0000 0001 0000 0010        5        write   r1
6 : 0000 0000 0000 0000        6        halt
```

If you have errors in the program, `hmmmAssembler.py` will tell you; otherwise it will produce the output file `triangle1.hb` ("hb" stands for "Hmmm binary"). We can then test our program by running the Hmmm simulator:

```
% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 5
Enter number: 5
12
```

We can see that the program produced the correct answer for the first test case, but not for the second. That's because Hmmm only works with integers; division rounds fractional values down to the next smaller integer just as it does in Python.

4.5.2 Looping

If you want to calculate the areas of a lot of triangles, it's a nuisance to have to run the program over and over. Hmmm offers a solution in the *unconditional jump* instruction, which says "instead of executing the next sequential instruction, start reading instructions beginning at address *n* ." If we simply replace the `halt` with a jump:

```
#
# Calculate the approximate areas of many triangles.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1       # Get base
1       read    r2       # Get height
2       mul     r1 r1 r2 # b times h into r1
3       setn    r2 2
4       div     r1 r1 r2 # Divide by 2
5       write   r1
6       jumpn   0
```

then our program will calculate triangle areas forever.

What's that `jumpn 0` instruction doing? The short explanation is that it's telling the computer to jump back to location 0 and continue executing the program there. The better explanation is that this instruction simply puts the number 0 in the program counter. Remember, the computer mindlessly checks its program counter to determine where to fetch its next instruction from memory. By placing a 0 in the program counter, we are ensuring that the next time the computer goes to fetch an instruction it will fetch it from memory location 0.

Since there will come a time when we want to stop, we can *force* the program to end by holding down the Ctrl ("Control") key and typing C (this is commonly written "Ctrl-C" or just "^C""):

```
% ./hmmSimulator.py
Enter binary input file name: triangle2.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
Enter number: 5
Enter number: 5
12
Enter number: ^C
```

```
Interrupted by user, halting program execution...
```

That works, but it produces a somewhat ugly message at the end. A nicer approach might be to automatically halt if the user inputs a zero for either the base or the height. We can do that with a *conditional jump* instruction, which works like `jumpn` if some *condition* is true, and otherwise does nothing.

There are several varieties of conditional jump statements and the one we'll use here is called `jeqzn` which is pronounced "jump to *n* if equal to zero" or just "jump if equal to zero." This conditional jump takes a register and a number as input. If the specified register contains the value zero then we will jump to the instruction specified by the number in the second argument. That is, if the register contains zero then we will place the number in the second argument in the program counter so that the computer will continue computing using that number as its next instruction.

I believe that I learned that when someone jeqzn's I should say "gesundheit"

```
#
# Calculate the approximate areas of many triangles.
# Stop when a base or height of zero is given.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1        # Get base
1       jeqzn   r1 9      # Jump to halt if base is zero
2       read    r2        # Get height
3       jeqzn   r2 9      # Jump to halt if height is zero
4       mul     r1 r1 r2 # b times h into r1
5       setn    r2 2
6       div     r1 r1 r2 # Divide by 2
7       write   r1
8       jumpn   0
9       halt
```

Now, our program behaves politely:

```
% ./hmmmSimulator.py
Enter binary input file name: triangle3.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
```

```
10
Enter number: 5
Enter number: 5
12
Enter number: 0
```

The nice thing about conditional jumps is that you aren't limited to just terminating loops; you can also use them to make decisions. For example, you should now be able to write a Hmmm program that prints the absolute value of a number. The other conditional jump statements in Hmmm are included in the listing of all Hmmm instructions at the end of this chapter.

4.5.3 Functions

Here is a program that computes factorials:

```
#
# Calculate N factorial.
#
# Input: N
# Output: N!
#
# Register usage:
#
#       r1      N
#       r2      Running product
#

0       read    r1       # Get N
1       setn    r2,1
2       jeqzn   r1,6     # Quit if N has reached zero
3       mul     r2,r1,r2 # Update product
4       addn    r1,-1    # Decrement N
5       jumpn   2        # Back for more

6       write   r2
7       halt
```

(If you give this program a negative number, it will crash unpleasantly; how could you fix that problem?) The `addn` instruction at line 4 simply adds a constant to a register, replacing its contents with the result. We could achieve the same effect by using `setn` and a normal `add`, but computer programs add constants so frequently that Hmmm provides a special instruction to make the job easier.

But suppose you need to write a program that computes $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. (If you haven't seen this formula before, it counts the number of different ways to choose k items from a set of n distinct items and it's pronounced " n choose k ".)

Since we need to compute three different factorials, we would like to avoid having to copy the above loop three different times. Instead, we'd prefer to have a *function* that computes factorials, just like Python has.

I wouldn't choose to write that program.

Creating a function is just a bit tricky. It can't read its input from the user, and it must return the value that it computed to whatever code called that function.

It's convenient to adopt a few simple conventions to make this all work smoothly. One such convention is to decide on special registers to be used for *parameter passing*, i.e., getting information into and out of a function. For example, we could decide that r1 will contain n when the factorial function starts, and r2 will contain the result. (As we'll see later, this approach is problematic in the general case, but it's adequate for now.)

Our new program, with the factorial function built in, is:

```
#
# Calculate C(n,k) = n!/k!(n-k)!.
#
# First input: N
# Second input: K
# Output: C(N,K)
#
# Register usage:
#
#       r1      Input to factorial function
#       r2      r1 factorial
#       r3      N
#       r4      K
#       r5      C(N,K)
#
# Factorial function starts at address 15
#

0       read    r3       # Get N
1       read    r4       # Get K

2       copy    r1,r3    # Calculate N!
3       calln   r14,15   # ...
4       copy    r5,r2    # Save N! as C(N,K)
```

```
5       copy    r1,r4    # Calculate K!
6       calln   r14,15   # ...
7       div     r5,r5,r2 # N!/K!

8       sub     r1,r3,r4 # Calculate (N-K)!
9       calln   r14,15   # ...
10      div     r5,r5,r2 # C(N,K)

11      write   r5       # Write answer
12      halt

13      nop              # Waste some space
14      nop

# Factorial function.  N is in R1. Result is R2.
# Return address is in R14.
15      setn    r2,1     # Initial product
16      jeqzn   r1,20    # Quit if N has reached zero
17      mul     r2,r1,r2 # Update product
18      addn    r1,-1    # Decrement N
19      jumpn   16       # Back for more

20      jumpr   r14      # Done; return to caller
```

As you can see, the program introduces a number of new instructions. The simplest is `nop`, the *no-operation* instruction, at lines 13 and 14. When executed, it does absolutely nothing.

Why would we want such an instruction? If you've already written some small Hmmm programs, you've probably discovered the inconvenience of renumbering lines. By including a few `nop`s as *padding*, we make it easy to insert a new instruction in the sequence from 0-15 without having to change where the factorial function begins.

Far more interesting is the `calln` instruction, which appears at lines 3, 6, and 9. `Calln` works similarly to `jumpn`: it causes Hmmm to start executing instructions at a given address, in this case 15. But if we had just used a `jumpn`, after the factorial function calculated its result, it wouldn't know whether to jump back to line 4, line 7, or line 10! To solve the problem, the `calln` uses register R14 to save the address of the instruction immediately *following* the call. [2]

One of the hmmmdingers of programming in assembly language!

We're not done, though: the factorial function itself faces the other end of the same dilemma. R14 contains 4, 7, or 10, but it would be clumsy to write code equivalent to "If R14 is 4, jump to 4; otherwise if R14 is 7, jump to 7; ..." Instead, the `jumpr` (jump to address in register) instruction solves the problem neatly, if somewhat confusingly. Rather than jumping to a fixed address given in the instruction (as is done in line 19), `jumpr` goes to a *variable* address taken from a register. In other words, if R14 is 4, `jumpr` will jump to address 4; if it's 7 it will jump to 7, and so forth.

4.5.4 Recursion

In Chapter 2 we learned to appreciate the power and elegance of recursion. But how can you do recursion in Hmmm assembly language? There must be a way; after all, Python implements recursion as a series of machine instructions on your computer. To understand the secret, we need to talk about the *stack*.

Stacks

You may recall that in Chapter 2 we talked about stacks (remember those stacks of storage boxes)? Now we're going to see precisely how they work.

A stack is something we are all familiar with in the physical world: it's just a pile where you can only get at the top thing. Make a tall stack of books; you can only see the cover of the top one, you can't remove books from the middle (at least not without risking a collapse!), and you can't add books anywhere except at the top.

The reason a stack is useful is because it remembers things and gives them back to you in reverse order. Suppose you're reading *Gone With the Wind* and you start wondering whether it presents the American Civil War fairly. You might put *GWTW* on a stack, pick up a history book, and start researching. As you're reading the history book, you run into an unfamiliar word. You place the history book on the stack and pick up a dictionary. When you're done with the dictionary, you return it to the shelf, and the top of the stack has the history book, right where you left off. After you finish your research, you put that book aside, and voilá! *GWTW* is waiting for you.

The sheer weight of that book could crush your stack!

This technique of putting things aside and then returning to them is precisely what we need for recursion. To calculate $5!$, you need to find $4!$ for which you need $3!$ and so forth. A stack can remember where we were in the calculation of $4!$ and help us to return to it later.

To implement a stack on a computer, we need a *stack pointer*, which is a register that holds the memory address of the top item on

Again, this is just a convention.

the stack. Traditionally, the stack pointer is kept in the highest-numbered register, so on Hmmm we use R15.

Suppose R15 currently contains 102, [3] and the stack contains 42 (on top), 56, and 12. Then we would draw the stack like this:

	Address	Contents
R15 $\rightarrow$	102	42
	101	56
	100	12

To *push* a value, say 23, onto the stack, we must increment R15 to contain the address of a new location (103) and then store 23 into that spot in memory. The stack will now look like:

Later, to *pop* the top value off, we must ask R15 where the top is [4] (103) and recover the value in that location. Then we decrement R15 so that it points to the new stack top, location 102. Everything is now back where we started.

The code to push something, say the contents of R4, on the stack looks like this:

```
addn r15,1       # Point to a new location
storer r4,r15    # Store r4 on the stack
```

The `storer` instruction stores the contents of R4 into the memory location *addressed* by register 15. In other words, if R15 contains 103, the value in R4 will be copied into memory location 103.

Just as `storer` can put things onto the stack, `loadr` will recover them:

```
loadr r3,r15     # Load r3 from the stack
addn r15,-1      # Point to new top of stack
```

Important note: in the first example above, the stack pointer is incremented *before* the `storer`; in the second, it is decremented *after* the *loadr*`. This ordering is necessary to make the stack work properly; you should be sure you understand it before proceeding further.

Saving Precious Possessions

When we wrote the $\binom{n}{k}$ program in Section 4.5.3, we took advantage of our knowledge of how the factorial function on line 15 worked. In particular, we knew that it only modified registers R1, R2, and R14, so that we could use R3 and R4 for our own purposes. In more complex programs, however, it may not be possible to partition the registers so neatly. This is especially true in recursive functions, which by their nature tend to re-use the same registers that their callers use.

The only way to be sure that a called function won't clobber your data is to save it somewhere, and the stack is a perfect place to use for that purpose. When writing a Hmmm program, the convention is that you must save all of your "precious possessions" before calling a function, and restore them afterwards. Because of how a stack works, you have to restore things in reverse order.

But what's a precious possession? The quick answer is that it's any register that you plan to use, *except* R0 and R15. In particular, if you are calling a function from inside another function, R14 is a precious possession.

Many newcomers to Hmmm try to take shortcuts with stack saving and restoring. That is to reason, "I know that I'm calling two functions in a row, so it's silly to restore my precious possessions and save them again right away." Although you can get away with that trick in certain situations, it's very difficult to get it right, and you are much more likely to cause yourself trouble by trying to save time. We strongly suggest that you follow the suggested *stack discipline* rigorously to avoid problems.

It's a common mistake, but to err is hmmman.

Let's look at a Hmmm program that uses the stack. We'll use the recursive algorithm to calculate factorials:

```
# Calculate N factorial, recursively
#
# Input: N
# Output: N!
#
# Register usage:
#
#       r1      N! (returned by called function)
#       r2      N

0       setn    r15,100  # Initialize stack pointer
1       read    r2       # Get N
2       calln   r14,5    # Recursive function finds N!
3       write   r1       # Write result
4       halt
```

```
# Function to compute N factorial recursively
#
# Inputs:
#
#       r2      N
#
# Outputs:
#
#       r1      N!
#
# Register usage:
#
#       r1      N! (from recursive call)
#       r2      N (for multiplication)

5       jeqzn   r2,18    # Test for base case (0!)

6       addn    r15,1    # Save precious possessions
7       storer  r2,r15   # ...
8       addn    r15,1    # ...
9       storer  r14,r15  # ...

10      addn    r2,-1    # Calculate N-1
11      calln   r14,5    # Call ourselves recursively to get (N-1)!

12      loadr   r14,r15  # Recover precious possessions
13      addn    r15,-1   # ...
14      loadr   r2,r15   # ...
15      addn    r15,-1   # ...

16      mul     r1,r1,r2 # (N-1)! times N
17      jumpr   r14      # Return to caller

# Base case: 0! is always 1
18      setn    r1,1
19      jumpr   r14      # Return to caller
```

The main program is quite simple (lines 0 - 4): we read a number from the user, call the recursive factorial function, and write the answer, and then halt.

The factorial function is only a bit more complex. After testing for the base case of $0!$, we save our "precious possessions" in preparation for the recursive call (lines 6 - 9). But what is precious to us? The function uses registers R1, R2, R14, and R15. We don't need to save R15 because it's the stack pointer, and R1 doesn't yet have anything valuable in it. So we only have to save R2 and R14. We follow the stack discipline, placing R2 on the stack (line 7) and then R14 (line 9).

After saving our precious possessions, we call ourselves recursively to calculate

$(N-1)!$ (lines 10–11) and then recover registers R14 (line 12) and R2 (line 14) in reverse order, again following the stack discipline. Then we multiply $(N-1)!$ by N, and we're done.

It is worth spending a bit of time studying this example to be sure that you understand how it operates. Draw the stack on a piece of paper, and work out how values get pushed onto the stack and popped back off when the program calculates $3!$.

4.5.5 The Complete Hmmm Instruction Set

This finishes our discussion of Hmmm. We have covered all of the instructions except `sub`, `mod`, `jnezn`, `jgtzn`, and `jltzn`; we trust that those don't need separate explanations.

Note that `sub` can be combined with `jltzn` to evaluate expressions like $a < b$.

For convenience, Figure 4.10 at the bottom of the page summarizes the entire instruction set, and also gives the binary encoding of each instruction.

As a final note, you may find it instructive to compare the encodings of certain pairs of instructions. In particular, what is the difference between `add`, `mov`, and `nop`? How does `calln` relate to `jumpn`? We will leave you with these puzzles as we move on to imperative programming.

4.5.6 A Few Last Words

What actually happens when we run a program that we've written in a language such as Python? In some systems, the entire program is first automatically translated or *compiled* into machine language (the binary equivalent of assembly language) using another program called a *compiler*. The resulting compiled program looks like the Hmmm code that we've written and is executed on the computer. Another approach is to use a program called an *interpreter* that translates the instructions one line at a time into something that can be executed by the computer's hardware.

It's important to keep in mind that exactly when and how the program gets translated - all at once as in the case of a compiler or one line at a time in the case of an interpreter - is an issue separate from the language itself. In fact, some languages have both compilers and interpreters, so that we can use one or the other.

Why would someone even care whether their program was compiled or interpreted? In the compiled approach, the entire program is converted into machine language before it is executed. The program runs very fast but if there is an error when the program runs, we probably won't get very much information from the computer other than seeing that the program "crashed". In the interpreted version, the interpreter serves as a sort of "middle-man" between our program and the computer. The interpreter can examine our instruction before translating it into machine language. Many bugs can be detected and reported by the interpreter before the instruction is ever actually executed by the computer. The "middle-man" slows down the execution of the program but provides an opportunity to detect potential problems. In any case, ultimately every program that we write is executed as code in machine language. This machine language code is decoded by digital circuits that execute the code on other circuits.

4.6 Conclusion

Aye caramba! That was a lot. We've climbed the levels of abstraction from transistors, to logic gates, to a ripple-carry adder, and ultimately saw the general idea of a how a computer works. Finally, we've programmed that computer in its native language.

Now that we've seen the foundations of how a computer works, we're going back to programming and problem-solving. In the next couple of chapters, some of the programming concepts that we'll see will be directly tied to the issues that we examined in this chapter. We hope that the understanding that you have for the internal workings of a computer will help the concepts that we're about to see be even more meaningful than they would be otherwise.

I think a mochaccino and a donut would be quite meaningful right about now.

Instruction	Description	Aliases
System instructions		
halt	Stop!	None
read rX	Place user input in register rX	None
write rX	Print contents of register rX	None
nop	Do nothing	None
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	None
addn rX N	Add integer N (-128 to 127) to register rX	None
copy rX rY	Set rX = rY	**mov**

Arithmetic		
add rX rY rZ	Set rX = rY + rZ	None
sub rX rY rZ	Set rX = rY - rZ	None
neg rX rY	Set rX = -rY	None
mul rX rY rZ	Set rX = rY * rZ	None
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)	None
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	None
Jumps!		
jumpn N	Set program counter to address N	None
jumpr rX	Set program counter to address in rX	**jump**
jeqzn rX N	If rX == 0, then jump to line N	**jeqz**
jnezn rX N	If rX != 0, then jump to line N	**jnez**
jgtzn rX N	If rX > 0, then jump to line N	**jgtz**
jltzn rX N	If rX < 0, then jump to line N	**jltz**
calln rX N	Copy the next address into rX and then jump to mem. addr. N	**call**
Interacting with memory (RAM)		
loadn rX N	Load register rX with the contents of memory address N	None
storen rX N	Store contents of register rX into memory address N	None
loadr rX rY	Load register rX with data from the address location held in reg. rY	**loadi**, **load**
storer rX rY	Store contents of register rX into memory address held in reg. rY	**storei**, **store**

Figure 4.10: The Hmmm instruction set.

Footnotes

[1] On Windows, you can navigate to the command line from the Start menu. On a Macintosh, bring up the Terminal application, which lives in Applications/Utilities. On Linux, most GUIs offer a similar terminal application in their main menu. The example above was run on Linux.

[2] We could have chosen any register except R0 for this purpose, but by convention Hmmm programs use R14.

[3] We say that R15 *points to* location 102.

[4] We say we *follow the pointer* in R15.