tenth of an inch. And inside these parts are a multitude more parts that are very close to each other, some just thousandths of an inch apart. And the shorter the distance the electricity has to travel, the sooner it gets there.

There is no point in saying how many things today’s computers do in a single second, because that would date this book. Computer manufacturers continue to produce new computers that go twice as fast as the fastest computers of only two or three years past. There is a theoretical limit to how fast they can go, but engineers keep finding practical ways to get around the theories and make machines that go faster and faster.

During all of this time that computers have been getting faster, smaller and cheaper, the things that computers do, really have not changed since they were first invented in the 1940's. They still do the same few simple things, just faster, cheaper, more reliably and in a smaller package.

There are only a few sections to a computer, and they are all made out of the same kinds of parts. Each section has a specific mission, and the combination of these parts into a machine was a truly marvelous invention. But it is not difficult to understand.

be able to call it a bit. It could sit on a table with either the “yes” or “no” showing. Then it would only have two states.

You have probably heard of bits before in relation to computers, and now you know what they are. In a computer, the bits are not like the coin or the lock, they are most like the light. That is, the bits in a computer are places that either have electricity or they do not. In a computer, the bits are very, very small and there are a very large number of bits, but that’s all that is in there.

Like the light in the living room, the bit is either on or off. In the living room, there is electricity in the wall coming into the switch. When you turn the switch on, the electricity goes from the switch, through the wires in the wall and ceiling, into the light socket and then into the light bulb. So this bit in the living room is several feet long, it includes the switch, the wires, the socket and the light bulb. In a computer, bits are mostly tiny, actually microscopic. Also, the computer bit doesn’t have a mechanical switch at one end or a light bulb at the other. If you removed the light bulb from the socket in the living room, the switch would still send electricity to the socket when it was on, and it would still be a bit – you just wouldn’t be able to see whether it was on or off by looking at a light bulb. Your computer has something resembling switches, like the keys on the keyboard, and something resembling light bulbs, like the tiny dots on the screen, but most of the bits are inside and unseen.

This is basically all there is in a computer – bits. There are lots and lots of them, and they are arranged and connected up in various ways, which we will examine in detail as the book progresses, but this is what is inside all computers – bits. A bit is always in one of its two possible states, either off or on, and they change between on and off when they are told to do so. Computer bits aren’t like the coin that has to physically flip over to change from one state to the other. Bits don’t change shape or location, they don’t look any different, they don’t move or rotate or get bigger or smaller. A computer bit is just a place, if there is no electricity in that place, then the bit is

What the…?

Imagine it is a bright sunny day, and you walk into a room with lots of open windows. You notice that the ceiling light is on. You decide that this is a waste, and you are going to turn the light off. You look at the wall next to the door and see a switch plate with two switches. So you assume that the one closer to the door is for the ceiling light. But then you notice that the switch is already off. And the other switch is off too. So then you think "well, maybe someone installed the switch upside down," so you decide to flip the switch anyway. You flip it on and off but nothing happens, the ceiling light stays lit. So then you decide that it must be the other switch, and you flip it on, off, on, off. Again nothing happens, that ceiling light continues to shine at you. You look around, there is no other door, there are no other switches, no apparent way to turn off this darned light. It just has to be one of these two switches, who built this crazy house anyway? So you grab one switch with each hand and start flipping them wildly. Then suddenly you notice the ceiling light flicker off briefly. So you slow down your switch flipping and stop when the ceiling light is off. Both switches say "on", and the light is now off. You turn one switch off, then on, and the light goes on, then back off. This is backwards. One switch off equals light on? So then you turn the other switch off, then on, the same thing, the light goes on, then back off. What the heck? Anyway, you finally figure out how it works. If both switches are on, the light goes off. If one or the other or both switches are off, then the ceiling light is on. Kind of goofy, but you accomplish what you intended, you turn both switches on, the light goes off, and you get the heck out of this crazy room.

Now what is the purpose of this little story about the odd light switches? The answer is, that in this chapter we are going to present the most basic part that computers are made of. This part works exactly like the lighting system in that strange room.

This computer part is a simple device that has three connections where there may or may not be some

Off	**Off**	**On**
Off	**On**	**On**
On	**Off**	**On**
On	**On**	**Off**

Each line shows one possible combination of the inputs, and what the output will be under those circumstances.

Compare this little chart with the experience with the odd room with the two light switches. If one switch is called ‘a,’ the other switch is called ‘b,’ and the ceiling light is called ‘c,’ then this little chart describes completely and exactly how the equipment in that room operates. The only way to get that light off is to have both switch ‘a’ and switch ‘b’ on.

Diagrams

If you want to see how a mechanical machine works, the best way to do it is to look inside of it, watch the parts move as it operates, disassemble it, etc. The second best way is to study it from a book that has a lot of pictures showing the parts and how they interact.

A computer is also a machine, but the only thing that moves inside of it is the invisible and silent electricity. It is very boring to watch the inside of a computer, it doesn't look like anything is happening at all.

The actual construction of the individual parts of a computer is a very interesting subject, but we are not going to cover it any further than to say the following: The technique starts with a thin crystal wafer, and in a series of steps, it is subjected to various chemicals, photographic processes, heat and vaporized metal. The result is something called a 'chip,' which has millions of electronic parts constructed on its surface. The process includes connecting the parts into gates, and connecting the gates into complete computer sections. The chip is then encased in a piece of plastic that has pins coming out of it. Several of these are plugged into a board, and there you have a computer. The computer we are going to 'build' in this book could easily fit on one chip less than a quarter of an inch square.

But the point is, that unlike a mechanical machine, the actual structure of a chip is very cluttered and hard to follow, and you can't see the electricity anyway. The diagrams we saw in the previous chapter are the best way to show how a computer works, so we'd better get pretty good at reading them.

Throughout the rest of this book, we are going to build new parts by connecting several gates together. We will describe what the new part does, and then give it a name and its own symbol. Then we may connect several of those new parts into something else that also gets a name and a symbol. Before you know it, we will have assembled a complete computer.

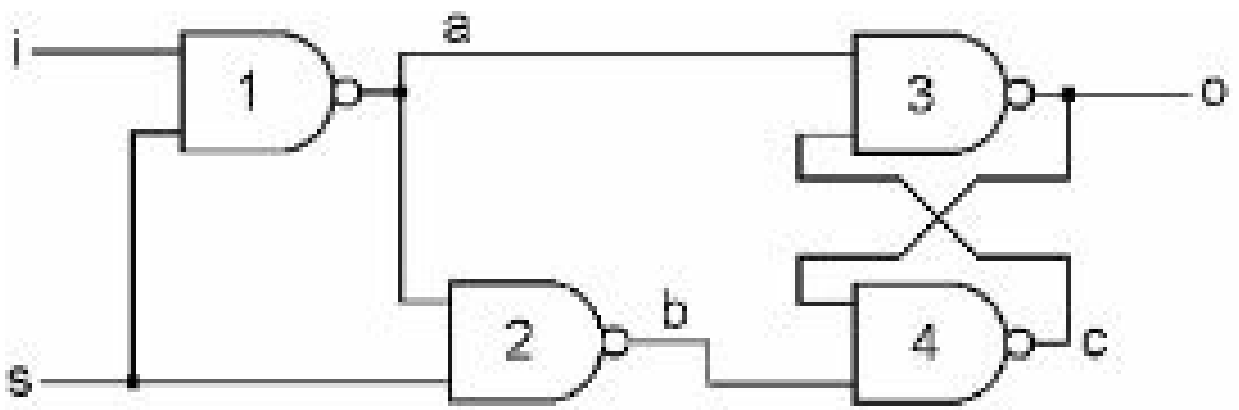
Every time there is a new diagram, the text will explain

Remember When

You have probably heard of computer memory, and now we are going to see exactly what that is. Since the only thing inside of computers is bits, and the only thing that happens to bits is that they either turn on or turn off, then it follows that the only thing a computer can 'remember' is whether a bit was on or off. We will now see how that is accomplished.

The following diagram shows one bit of computer memory. It happens to be one of the neatest tricks you can do with a few gates. We will examine how it works here at great length, and after we understand it, we will replace it with its own symbol, and use it as a building block for bigger and better things.

It is made of only four NAND gates, but its wiring is kind of special. Here it is:



This combination as a whole has two inputs and one output. 'I' is where we input the bit that we want to remember, and 'o' is the output of the remembered bit. 'S' is an input that tells these gates when to 'set' the memory. There are also three internal wires labeled 'a', 'b' and 'c' that we will have to look at to see how these parts work together. Try to follow this carefully, once you see that it works, you will understand one of the most important and most commonly used things in a computer.

To see how this works, start with 's' on and 'i' off. Since 'i' and 's' go into gate 1, one input is off, so 'a' will be on. Since 'a' and 's' go to gate 2, both

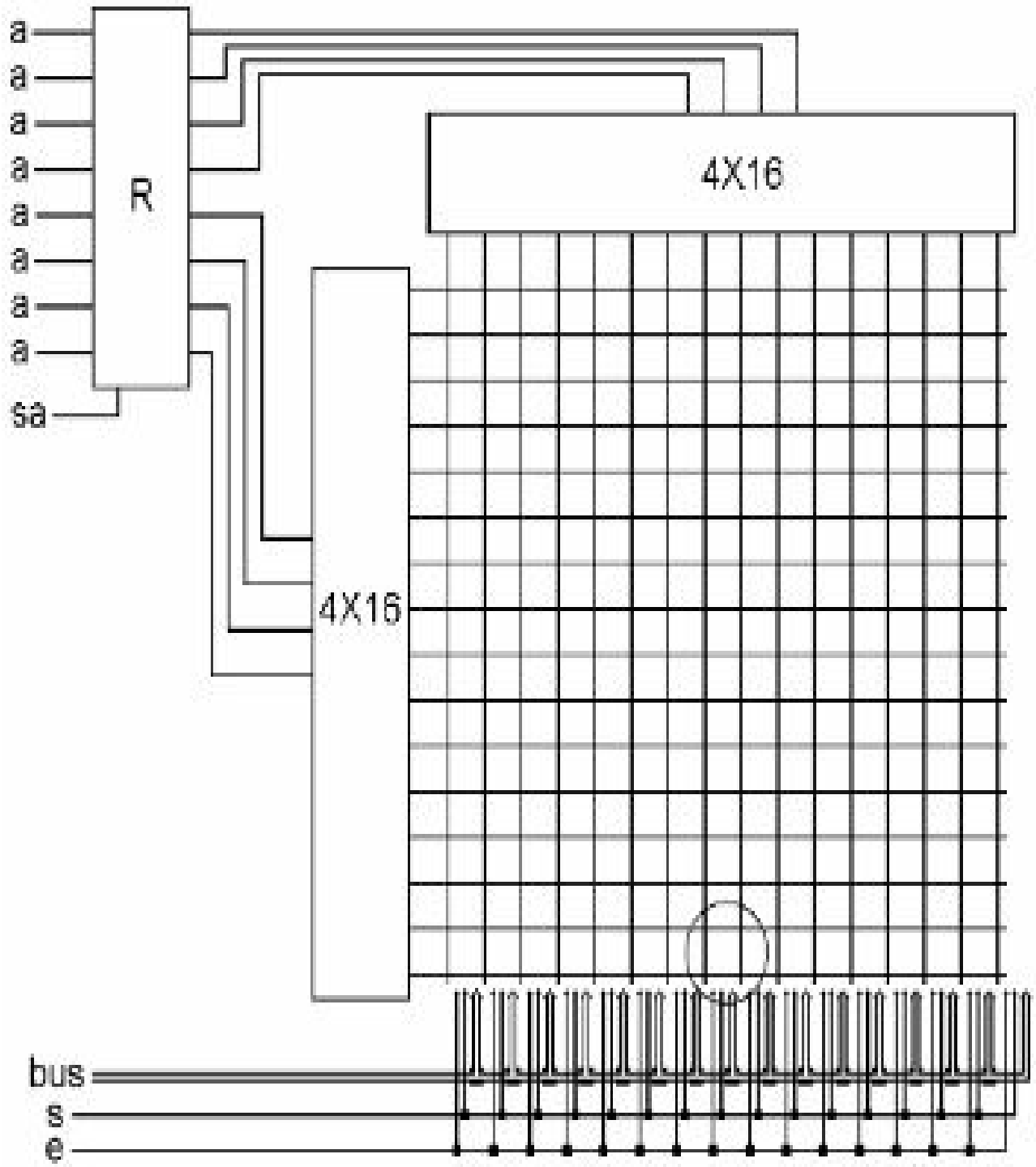
1	**1**	**0**	**0**
1	**1**	**1**	**1**

Imagine replacing input ‘c’ with another AND gate, then you would have a four input AND gate. You could then replace any of the four inputs with another AND gate, and have a five input AND gate. This can be done as many times as necessary for what you are doing.

As you add inputs, the chart will need more and more lines. Every time you add another input, you double the number of combinations that the inputs can have. The chart we saw for the original two input AND gate had four lines, one for each possibility. The three input, directly above, has eight lines. A four input AND gate will have 16 lines, a five input will have 32, etc. In all cases though, for an AND gate, only one combination will result in the output turning on, that being the line where all inputs are on.

Here is the last combination we need to make the first half of a computer. This combination is different from anything we have looked at so far, in that it has more outputs than inputs. Our first example has two inputs and four outputs. It is not very complicated, it just has two NOT gates and four AND gates.

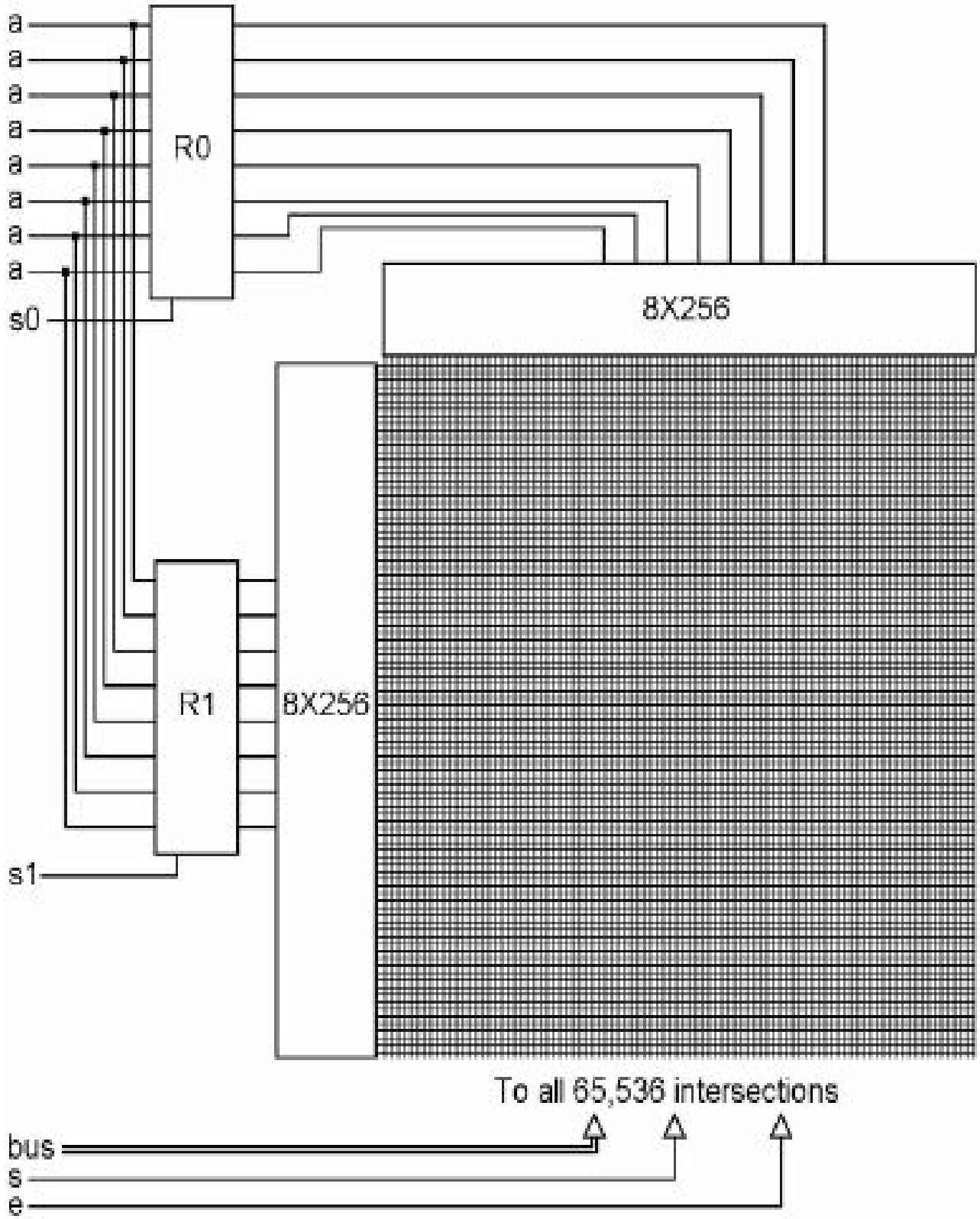
In the diagram below, ‘a’ and ‘b’ are the inputs coming in from the left. Both of them are connected to NOT gates. The NOT gates generate the opposite of their inputs. There are four vertical wires going down the page that come from ‘a’ and ‘b’ and the opposites of ‘a’ and ‘b.’ Thus, for each ‘a’ and ‘b,’ there are two wires going down the page, where one of them will be on if its input is on, and the other will be on if its input is off. Now we put four AND gates on the right, and connect each one to a different pair of the vertical wires such that each AND gate will turn on for a different one of the four possible combinations of ‘a’ and ‘b.’ The top



At the bottom of this diagram is one bus and an ‘s’ and ‘e’ bit, just the same as the connections that go to a register. As you can see, they go upwards and into the grid. The diagram doesn’t show it, but they go up under

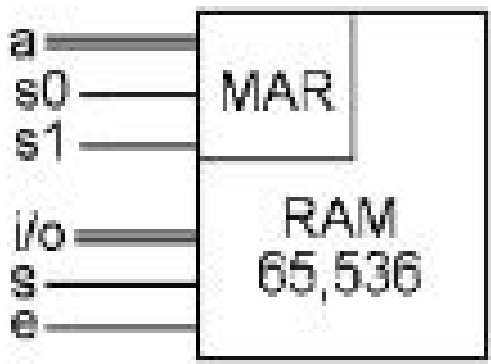
the grid all the way to the top, so that each of the 256 intersections has a bus and an ‘s’ and ‘e’ bit nearby.

There is a circle on the diagram above, around one of the intersections of the grid. What is in this circle is magnified in the diagram below, showing that there are three AND gates and one register at each of the 256 intersections. As we can see, there is an AND gate ‘x,’ connected to the one vertical grid wire and the one horizontal grid wire at this intersection. These ‘x’ gates are the only things connected to the grid. The rest of the connections go down to the bus and ‘s’ and ‘e’ bits at the bottom of the diagram. Remember that there is only one intersection where both grid wires are on. Therefore, there are 256 of these ‘x’ gates, but only one of them has its output on at any given time. The output of that ‘x’ gate goes to one side each of two more AND gates. These two gates control access to the set and enable inputs of the register at that intersection. So when an ‘x’ gate is off, the ‘s’ and ‘e’ bits of that register cannot be turned on. That will be the case for 255 of these registers, the ones where the ‘x’ gate is off. But one intersection has its ‘x’ gate on, and its register can be set from the bus, or its contents can be enabled onto the bus and sent elsewhere by using the ‘s’ and ‘e’ bits at the bottom of the diagram.


a
a
a
a
a
a
a
a
s0
R0
8X256
R1
8X256
s1
To all 65,536 intersections
bus
s
e

A bus carries one byte at a time, so selecting one of the 65,536 memory locations of this RAM would be a two-step process. First, one byte would have to be placed on the ‘a’ bus and set into R0, then the second byte would have to be placed onto the ‘a’ bus and set into R1. Now you could access the desired memory location with the bus and the ‘s’ and ‘e’ bits at the bottom of the drawing.

Simplifying again, we have something that looks very much like our 256 byte RAM, it just has one more input bit.



For the rest of this book, we will be using the 256 byte RAM just to keep things simple. If you want to imagine a computer with a larger RAM, every time we send a byte to the Memory Address Register, all you have to do is imagine sending two bytes instead.

In this code, 0000 0001 means one, 0001 0000 means sixteen, 0001 0001 means seventeen (sixteen plus one,) 1111 1111 means 255, etc. In an eight-bit byte, we can represent a number anywhere from 0 to 255. This code is called the “binary number code.”

The computer works just fine with this arrangement, but it is annoying for people to use. Just saying what is in a byte is a problem. If you have 0000 0010, you can call it “zero zero zero zero zero zero one zero binary” or you can mentally translate it to decimal and call it “two,” and that is usually what is done. In this book when a number is spelled out, such as ‘twelve,’ it means 12 in our decimal system. A binary 0000 0100 would be called ‘four,’ because that is what it works out to be in decimal.

Actually, in the computer industry, people often use hexadecimal, (and they just call it ‘hex’.) If you look at the chart above, you can see that four digits of binary can be expressed by one digit of hex. If you have a byte containing 0011 1100, you can translate it to 60 decimal, or just call it “3C hex.” Now don’t worry, we’re not going to use hex in this book, but you may have seen these types of numbers somewhere, and now you know what that was all about.

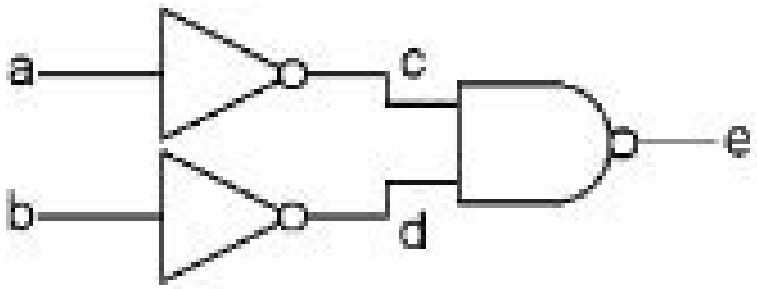
Addresses

Now that we have the binary number code, we can use it for various purposes in our computer. One of the first places we will use it, is in the Memory Address Register. The pattern of bits that we put into this register will use the binary number code. The bits of this number in MAR then select one of the 256 RAM storage locations. The number in MAR is considered to be a number somewhere between 0 and 255, and thus each of the 256 RAM bytes can be considered to have an address.

This is fairly simple, but a point needs to be made here about exactly what is meant by an address inside of a computer. In a neighborhood of homes, each house has an address, like 125 Maple Street. There is a sign at the corner that says "Maple St." and written on the house are the numerals "125." This is the way we normally think of addresses. The point to be made here is that the houses and streets have numbers or names written on them. In the computer, the byte does not have any identifying information on it or contained in it. It is simply the byte that gets selected when you put that number in the Memory Address Register. The byte gets selected by virtue of where it is, not by any other factor that is contained at that location. Imagine a neighborhood of houses that had sixteen streets, and sixteen houses on each street. Imagine that the streets do not have signs and the houses do not have numbers written on them. You would still be able to find any specific house if you were told, for example, to go to 'the fourth house on the seventh street.' That house still has an address, that is, a method of locating it, it just doesn't have any identifying information at the location. So a computer address is just a number that causes a certain byte to be selected when that address is placed into the Memory Address Register.

More Gates

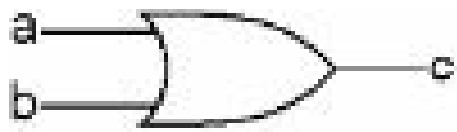
We have used NAND, AND and NOT gates so far. There are two more combination gates that we need to define. The first is built like this:



All it does is to NOT the two inputs to one of our good old NAND gates. Here is the chart for it, showing the intermediate wires so it is easy to follow.

a	b	c	d	e
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

In this case, when both inputs are off, the output is off, but if either 'a' OR 'b' is on, or both, then the output will be on. So it has another very simple name, it is called the "OR gate." Instead of drawing all the parts, it has its own diagram shaped something like a shield. The diagram and chart look like this:

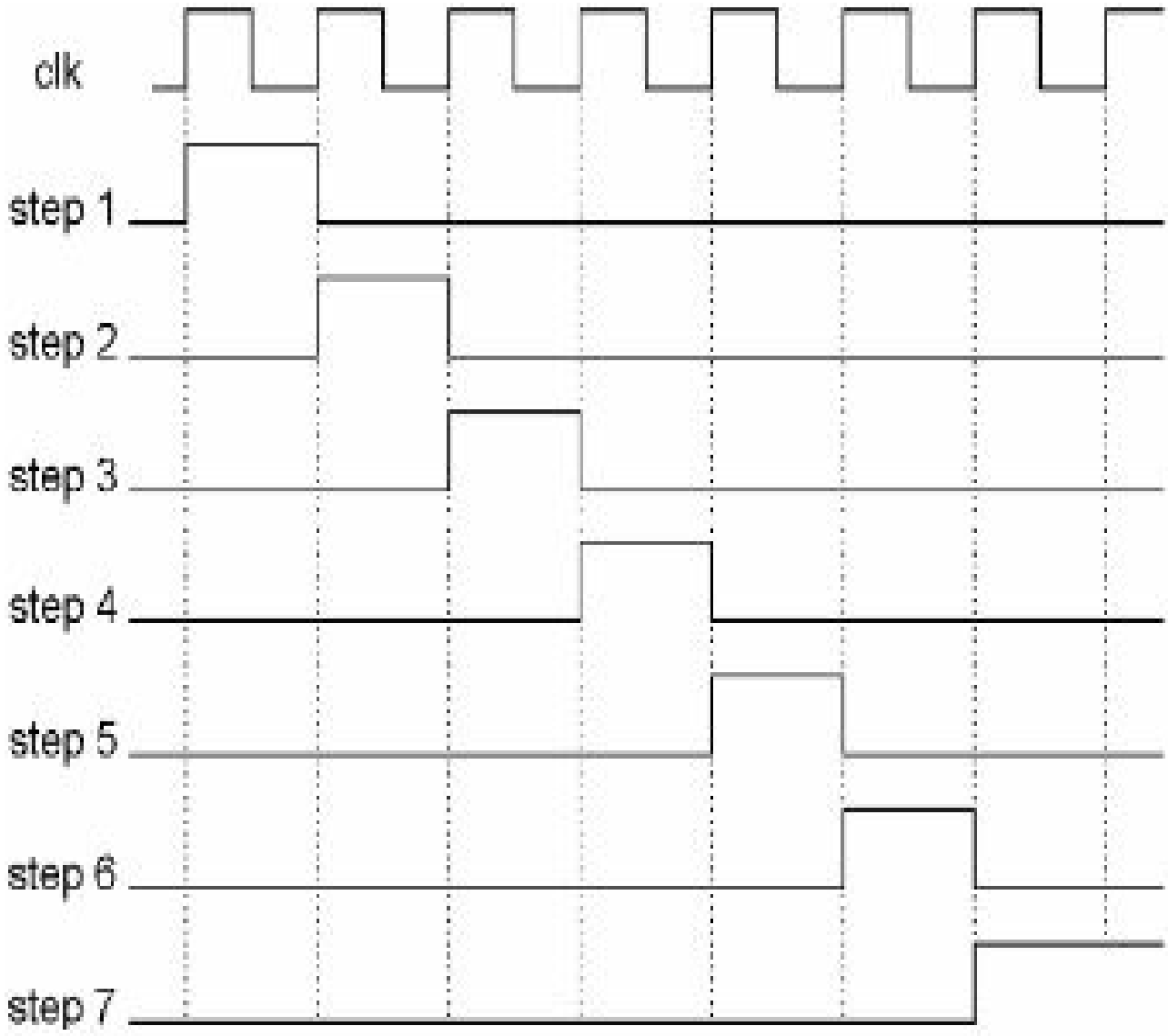


a	b	c
0	0	0
0	1	1

Messing with Bytes

Individual gates operate on bits. Two bits in, one bit out. But the RAM stores and retrieves a byte at a time. And the bus moves a byte at a time. Here in the CPU, we want to be able to work on a whole byte at one time. We want some 'gates' that affect an entire byte. In the chapter on the bus, we saw how the contents of a byte can be copied from one register to another. This is usually referred to as moving a byte. Now we are going to see some variations on this.

First we will see three ways that we can change the contents of a byte as it moves from one register to another. Second, we will see four ways that we can take the contents of two bytes, and have them interact with each other to create the contents for a third byte. These are all of the things that computers actually do to bytes. All things ultimately come down to these seven operations.



Here is how the stepper is built. It is done using some of the same memory bits that we used to make registers, but they are arranged very differently. We are not going to store anything in these bits, we are going to use them to create a series of steps.

The stepper consists of several memory bits connected in a string, with the output of one connected to the input of the next. Here is a diagram that shows most of the stepper:

because its ‘set’ bit is connected to ‘clk,’ which is now off. When ‘clk’ comes back on, the second ‘M’ will now come on. As the clock ticks, the ‘on’ that enters the first memory bit will step down the line, one bit for each time the clock goes on, and one bit for each time the clock goes off. Thus two bits come on for each clock cycle.

Now, turning to the full stepper diagram below, step 1 comes from a NOT gate connected to the output of the second ‘M.’ Since all ‘M’s start off, step 1 will be on until the second ‘M’ comes on, at which time step 1 will be over. For the remaining steps, each one will last from the time its left side ‘M’ turns on until the time its right side ‘M’ turns on. The AND gates for steps 2-6 have both inputs on when the left ‘M’ is on, and the right ‘M’ is off. If we connect the output of one ‘M’ and the NOT of the output of an ‘M’ two spaces farther on to an AND gate, its output will be on for one complete clock cycle. Each one comes on when its left input has come on, but its right input has not yet come on. This gives us a series of bits that each come on for one clock cycle and then turn off.

The only thing missing here is that the ‘M’ bits come on and stay on. Once they are all on, there is no more action despite the clock’s continued ticking. So we need a way to reset them all off so we can start over again. We have to have a way to turn off the input to the first ‘M,’ and then turn on all of the set bits at the same time. When that happens, the ‘off’ at the input to the first ‘M’ will travel through all of the ‘M’s as fast as it can go. We will add a new input called ‘reset,’ which will accomplish these things.

itself off as soon as the zero can get through the string of ‘M’s. This means that step 7 will not last long enough to be used for one of our data transfers over the bus. All of the things we want to accomplish will take place in steps 1 through 6.

The First Great Invention

What we need is some way to do different operations from one stepper sequence to the next. How could we have it wired up one way for one sequence, and then a different way for the next sequence? The answer, of course, is to use more gates. The wiring for one operation can be connected or disconnected with AND gates, and the wiring for a different operation can be connected or disconnected with some more AND gates. And there could be a third and fourth possibility or more. As long as only one of those operations is connected at one time, this will work fine. Now we have several different operations that can be done, but how do you select which one will be done?

The title of this chapter is "The First Great Invention," so what is the invention? The invention is that we will have a series of instructions in RAM that will tell the CPU what to do. We need three things to make this work.

The first part of the invention is, that we are going to add another register to the CPU. This register will be called the "Instruction Register," or "IR" for short. The bits from this register will "instruct" the CPU what to do. The IR gets its input from the bus, and its output goes into the control section of the CPU where the bits select one of several possible operations.

The second part of the invention is another register in the CPU called the "Instruction Address Register," or "IAR" for short. This register has its input and output connected to the bus just like the general purpose registers, but this one only has one purpose, and that is to store the RAM address of the next instruction that we want to move into the IR. If the IAR contains 0000 1010 (10 decimal,) then the next instruction that will be moved to the IR is the byte residing at RAM address ten.

The third part of the invention is some wiring in the control section that uses the stepper to move the desired "instruction" from RAM to the IR, add 1 to the address in the IAR and do the action called for by the

The Arithmetic or Logic Instruction

This first type of instruction is the type that uses the ALU like our ADD operation earlier. As you recall, the ALU has eight things it can do, and for some of those things it uses two bytes of input, for other things it only uses one byte of input. And in seven of those cases, it has one byte of output.

This type of instruction will choose one of the ALU operations, and two registers. This is the most versatile instruction that the computer can do. It actually has 128 variations, since there are eight operations, and four registers, and you get to choose twice from the four registers. That is eight times four times four, or 128 possible ways to use this instruction. Thus this is not just one instruction, but rather it is a whole class of instructions that all use the same wiring to get the job done.

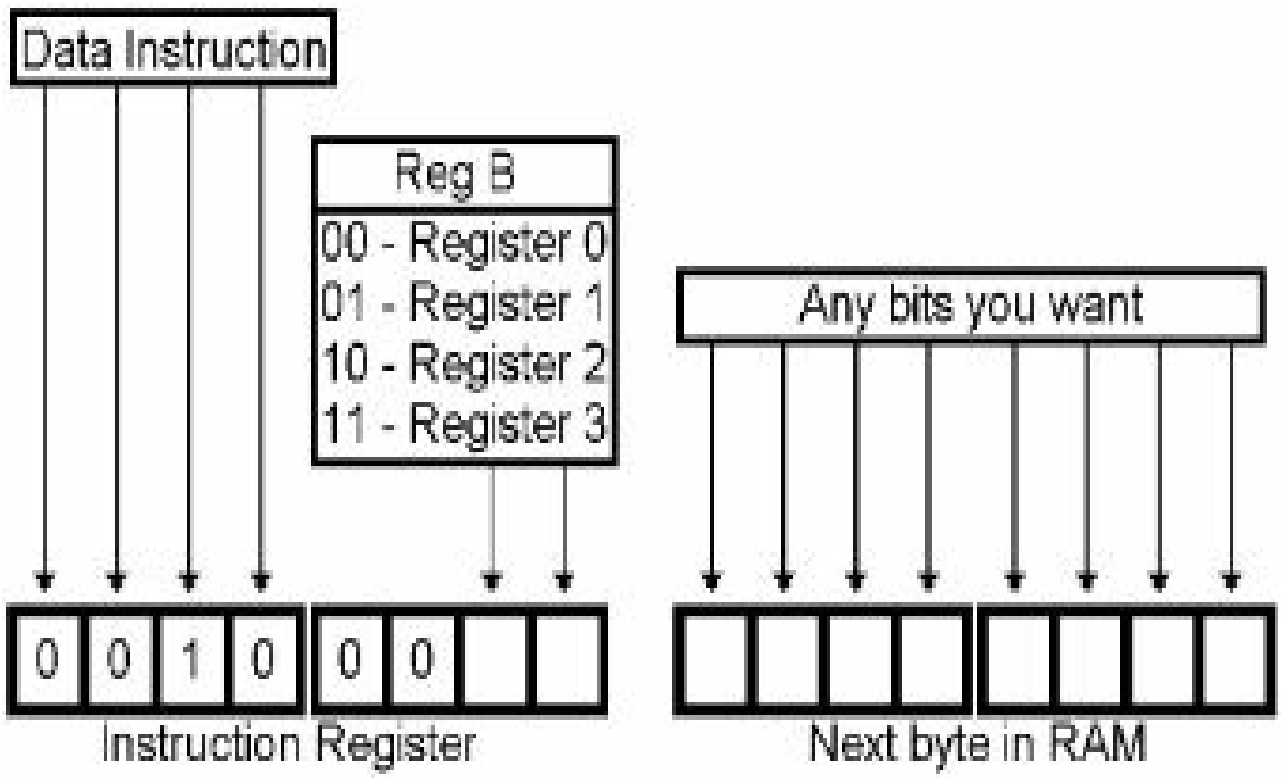
Here is the Instruction Code for the ALU instruction. If the first bit in the Instruction Register is a 1, then this is an ALU instruction. That's the simplicity of it. If the first bit is on, then the next three bits in the instruction get sent to the ALU to tell it what to do, the next two bits choose one of the registers that will be used, and the last two bits choose the other register that will be used.

The Data Instruction

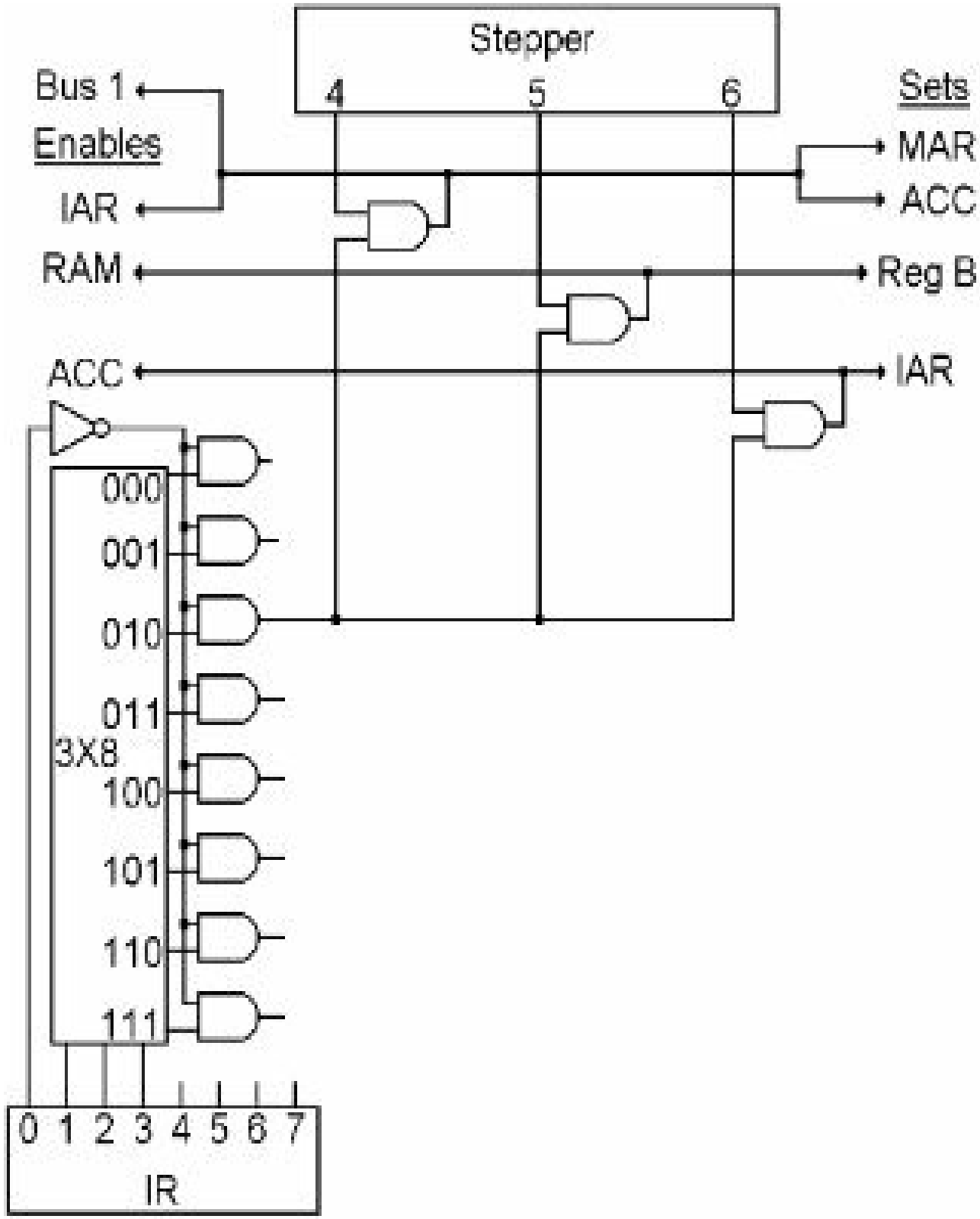
Now here is an interesting instruction. All it does is load a byte from RAM into a Register like the Load instruction, above. The thing that is different about it though, is where in RAM it will get that byte.

In the Data instruction, the data comes from where the next instruction ought to be. So you could consider that this instruction is actually two bytes long! The first byte is the instruction, and the next byte is some data that will be placed into a register. This data is easy to find, because by the time we have the instruction in the IR, the IAR has already been updated, and so it points right to this byte.

Here is the Instruction Code for the Data instruction. Bits 0 to 3 are 0010. Bits 4 and 5 are not used. Bits 6 and 7 select the register that will be loaded with the data that is in the second byte.



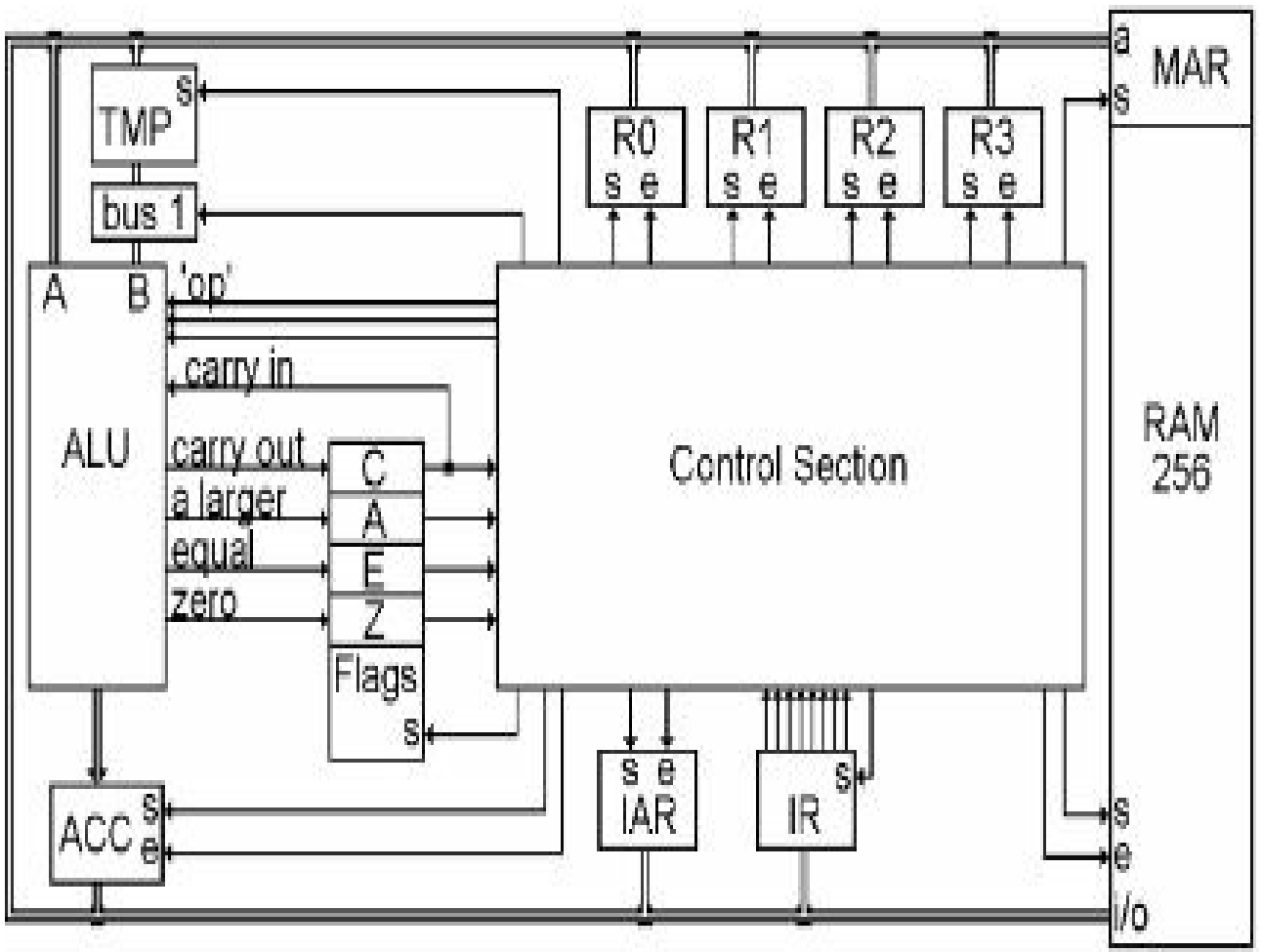
All this instruction needs to do is, in step 4, send IAR to MAR, and in step 5, send RAM to the desired CPU


Stepper
4
5
6
Bus 1
Enables
IAR
RAM
ACC
Sets
MAR
ACC
Reg B
IAR
000
001
010
011
3X8
100
101
110
111
0 1 2 3 4 5 6 7
IR

Like the Data instruction, the IAR already points to the byte we need. Unlike the Data Instruction, we don't need to add 1 to the IAR a second time because we are going to replace it anyway. So we only need two steps. In step 4, we send IAR to MAR. In step 5 we move the selected RAM byte to the IAR. Step 6 will do nothing.

Here is the wiring that makes it work:

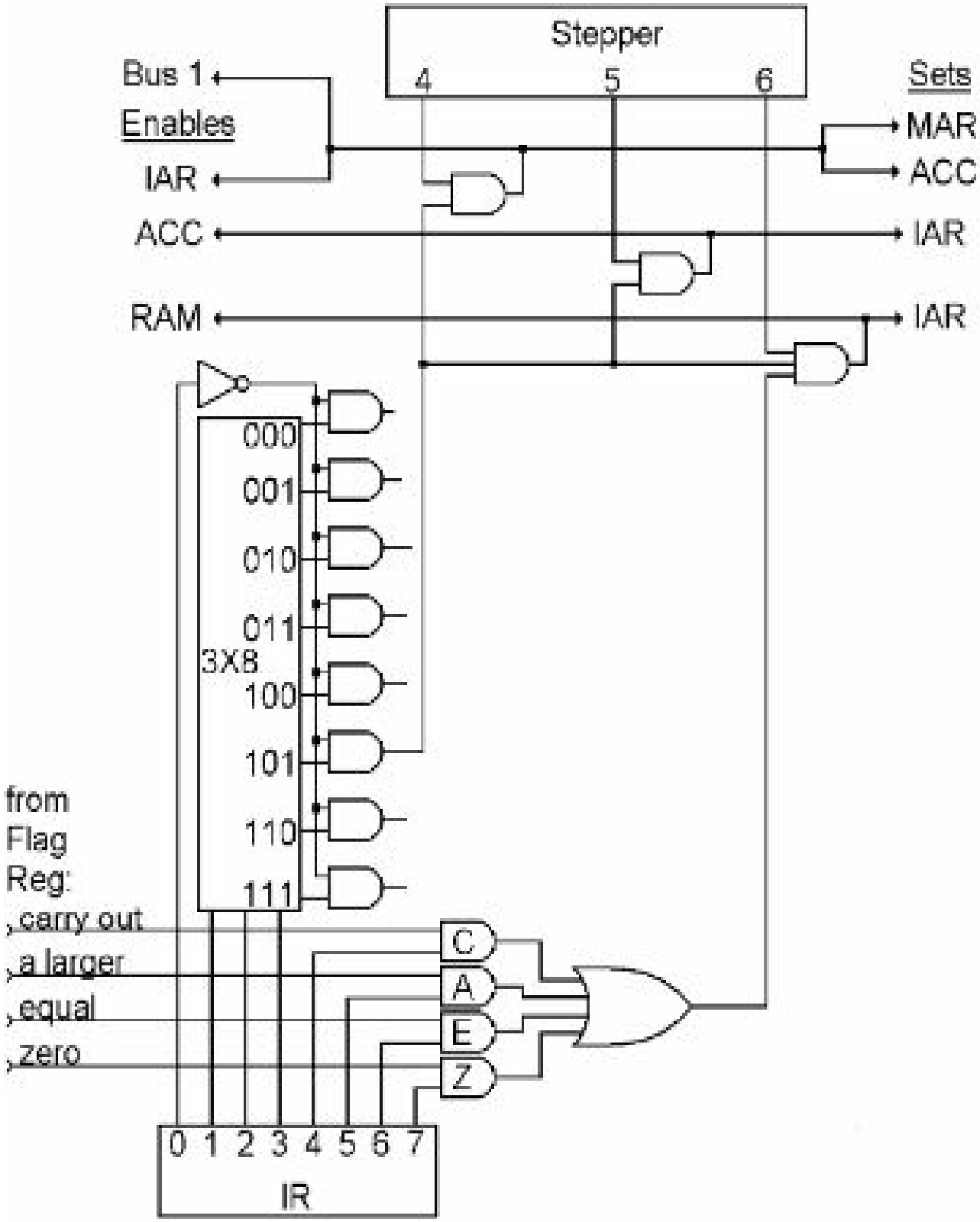
valid during step 5 of the ALU instruction. Therefore, we need a way to save the state of the Flag bits as they were during step 5 of the ALU instruction.



Here is the last register that we are going to add to the CPU. This will be called the FLAG register, and we are only going to use four bits of it, one for each of the flags.

The Flag bits from the ALU are connected to the input of this register, and it will be set during step 5 of the ALU instruction just like ACC and it will stay set that way until the next time an ALU instruction is executed. Thus if you have an ALU instruction followed by a “Jump If” instruction, the “Flag” bits can be used to “decide” whether to Jump or not.

Every instruction cycle uses the ALU in step 1 to add 1


Stepper
4
5
6
Bus 1
Enables
IAR
ACC
RAM
Sets
MAR
ACC
IAR
IAR
000
001
010
011
3X8
100
101
110
111
from
Flag
Reg:
,carry out
,a larger
,equal
,zero
C
A
E
Z
0 1 2 3 4 5 6 7
IR

JAE	Addr	Jump if A is larger or Equal to B
JAZ	Addr	Jump if A is larger or answer is
JEZ	Addr	Jump if A Equals B or answer is Z
JCAE	Addr	Jump if Carry or A larger or Equa
JCAZ	Addr	Jump if Carry or A larger or Zero
JCEZ	Addr	Jump if Carry or A Equals B or Ze
JAEZ	Addr	Jump if A larger or Equal to B or
JCAEZ	Addr	Jump if Carry, A larger, Equal or

The Clear Flags Instruction

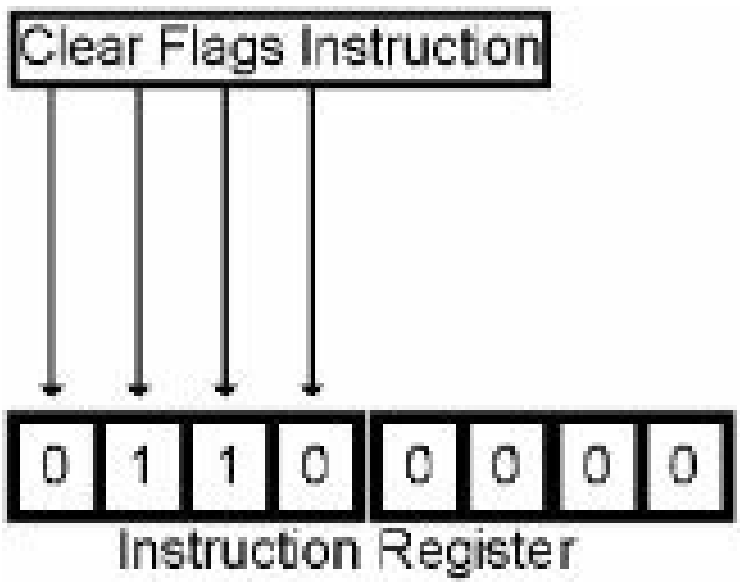
There is one annoying detail that we need to have here. When you do addition or shifting, you have the possibility of getting the carry flag turned on by the operation. This is necessary, we use it for the Jump If instruction as in the previous chapter.

The Carry Flag is also used as an input to the addition and shift operations. The purpose of this is so you can add numbers larger than 255 and shift bits from one register to another.

The problem that arises is that if you are just adding two single-byte numbers, you don't care about any previous Carry, but the Carry Flag may still be set from a previous operation. In that case, you might add 2+2 and get 5!

Bigger computers have several ways to do this, but for us, we will just have a Clear Flags Instruction that you need to use before any adds or shifts where an unexpected carry bit would be a problem.

Here is the Instruction Code for this instruction. Bits 4, 5, 6 and 7 are not used.



The wiring for this is very simple and a bit tricky. We will not enable anything onto the bus, thus it and the 'A' ALU input will be all zeros. We will turn on 'Bus 1'

Ta Daa!

We have now wired up the Control Section of our CPU. As a result, we can place a series of instructions in RAM, and the Clock, Stepper, Instruction Register and wiring will fetch and execute those instructions. Here is the entire control section:

Yes, this looks pretty complicated, but we have looked at every part of it already. The only thing we had to add were some OR gates because most of the 'enables' and 'sets' need multiple connections. This actually has a lot fewer parts than the RAM, but that was much more repetitive. Most of the mess here is just getting the wires from one place to another.

The byte that is placed in the Instruction Register causes a certain activity to occur. Each possible pattern causes a different activity. Therefore, we have a code where each of the 256 possible codes represents a different specific activity.

As mentioned, this is called the Instruction Code. Another name for it is "machine language," because this is the only language (code) that the machine (computer) "understands." You "tell" the machine what to do by giving it a list of orders you want it to carry out. But you have to speak the only language that it "understands." If you feed it the right byte-sized patterns of ons and offs, you can make it do something that will be useful.

Here are all of the Instruction Codes and our shorthand language brought together in one place.

Instruction Code		Language		Meaning
1000	rarb	ADD	RA,RB	Add
1001	rarb	SHR	RA,RB	Shift R
1010	rarb	SHL	RA,RB	Shift L
1011	rarb	NOT	RA,RB	Not
1100	rarb	AND	RA,RB	And
1101	rarb	OR	RA,RB	Or
1110	rarb	XOR	RA,RB	Exclusi
1111	rarb	CMP	RA,RB	Compare
0000	rarb	LD	RA,RB	Load RB
0001	rarb	ST	RA,RB	Store R
0010	00rb xxxxxxxx	DATA	RB,Data	Load th
0011	00rb	JMPR	RB	Jump to
0100	0000 xxxxxxxx	JMP	Addr	Jump to

on. When this Memory bit is on, it means that the keyboard adapter is active.

AND gate #3 comes on during ‘clk e’ time of an IN Data instruction. If the Memory bit is on, AND gate #4 will come on and the Keycode Register will be enabled onto the bus, which will be set into Reg B in the CPU.

Every adapter that is connected to the I/O bus needs to have the type of circuitry we see in gates #1 and #2 and the memory bit above. Each adapter will have a different combination that turns gate #1 on; this is what allows the CPU to select each adapter individually.

Here is a little program that moves the current keypress into Reg 3 in the CPU.

Instruction		Comments
Data	R2,0000 1111	* Put Addr of Keyboard in
OUT	Addr,R2	* Select Keyboard
IN	Data,R3	* Get ASCII of key presse
XOR	R2,R2	* Clear Address in Reg 2
OUT	Addr,R2	* Un-Select Keyboard

That little ‘Control’ box clears the Keycode Register after it has been sent to the CPU.

The program running in the CPU will check the keyboard adapter on a regular basis, and if the byte that it receives is all zeros, then no key has been pressed. If the byte has one or more bits on, then the program will do whatever the program has been designed to do with a keystroke at that time.

Again, we are not going to go through every gate in the Keyboard adapter. All device adapters have the same sorts of circuitry in order to be able to respond when they are addressed, and send or receive bytes of information as needed. But it is no more complicated than that. That is all that I/O devices and adapters do.

works.

In this chapter we will look at the simplest kind of screen, the kind that is black and white, and whose pixels can only either be fully on or fully off. This type of screen can display characters and the type of pictures that are made of line drawings. Later in the book we will see the few simple changes that enable a screen to display things like color photographs.

The major parts are three. First there is the computer, we have seen how that works. It has an I/O Bus that can move bytes to and from things outside of the computer. Second is the screen. The screen is just a large grid of pixels, each of which can be selected, one at a time, and while selected, can either be turned on, or not. The third item is the 'display adapter.' The display adapter is connected to the I/O Bus on one side, and to the screen on the other side.

The heart of a display adapter is some RAM. The display adapter needs its own RAM so it can "remember" which pixels should be on, and which pixels should be off. In the type of screen we are going to describe here, there needs to be one bit in RAM for each pixel on the screen.

In order to make the screen scan every pixel 30 times every second, the Display Adapter needs its own clock that ticks at a speed that is 30 times the number of pixels on the screen. At each tick of the clock, one pixel is selected and it is turned on or not by the corresponding bit from the RAM.

As an example, lets use an old type of screen. It is a black and white screen that displays 320 pixels across the screen and 200 pixels down. That comes out to 64,000 individual pixels on the screen. Each pixel on the screen has a unique address consisting of two numbers, the first being the left-right or horizontal position, and the other being the up-down or vertical position. The address of the top left pixel is 0,0 and the bottom right pixel is 319,199

64,000 pixels times 30 pictures per second means that this Display Adapter's clock needs to tick 1,920,000 times per second. And since there are eight bits in a byte, we will need 8,000 bytes of display RAM to tell each of the 64,000 screen pixels whether to be on or

operation. But the IAR has been replaced! So the next instruction will be fetched from whatever address was in RAM byte 2.

In other words, what the CPU had been doing is saved, and the CPU is sent off to do something else. If at the end of this new activity, the program puts RAM bytes 0 and 1 back into the IAR and Flags, the CPU will pick up from exactly where it left off, before it was interrupted.

This system is very useful for dealing with I/O operations. Without interrupts, the program running in the CPU would have to make sure to check all of the devices on the I/O Bus on a regular basis. With interrupts, the program can just do whatever it is designed to do, and the program that deals with things like keyboard input will be called automatically as needed by the interrupt system.

We have not included this in our CPU because it would just make our Control Section wiring diagram too big. It would need to add the following: two more steps to the stepper, wiring to do the above 8 steps in place of the normal instruction cycle, paths for the Flags register to get to and from the bus, a method of sending a binary 0, 1, 2 or 3 to MAR, and an instruction that restores RAM bytes 0 and 1 to the IAR and Flags register.

And that is an Interrupt system. As far as the language is concerned, the computer designers took an existing verb, 'interrupt,' and used it in three ways: It is a verb in "the keyboard interrupted the program," it is an adjective in "This is the Interrupt system," and it is a noun in "the CPU executed an interrupt."

Software can be copied easily by machine. All you need is something that can read the disk or whatever it is recorded on, and something else to write it onto a new disk. The new one will be just like the original, it will do all the same things. If the original is your favorite movie, the copy will also be your favorite movie. If the original is a program that will prepare your tax papers, so will the copy.

Software is not a physical thing, it is just how the physical things are set.

By far the most commonly used definition of 'software' is to refer to a package of computer instruction code. I think that the way it got this name is that once you have built a device as versatile as a computer, there are many different things that it can be made to do. But when there are no instructions in it, it can't do anything. So the software is an absolutely necessary part of a computer that is doing some task. It is a vital part of the total machine, yet it isn't like any other part in the machine. You can't weigh it or measure it or pick it up with a pair of pliers. So it is part of the 'ware,' but it isn't hardware. The only thing left to call it is 'software.'

Programs

As mentioned earlier, a series of instructions in RAM are called a program.

Programs come in many sizes. Generally, a program is a piece of software that has everything needed to do a specific task. A system would be something larger, made up of several programs. A program might be made up of several smaller parts known as 'routines.' Routines in turn may be made up of sub-routines.

There are no hard and fast definitions that differentiate between system, program, routine and sub-routine. Program is the general term for all of them, the only difference is their size and the way they are used.

There is another distinction between two types of programs that is not related to their size. Most home and business computers have a number of programs installed on them. Most of these programs are used to do something that the owner wants to do. These are called application programs because they are written to apply the computer to a problem that needs to be solved. There is one program on most computers that is not an application. Its job is to deal with the computer itself and to assist the application programs. This one program that is not an application is called the Operating System.

programs had to be written directly in ones and zeros. Then somebody got tired of the tedium of programming that way, and decided to write the first compiler. Then ever after, programs were written in this easier language, and then translated into Instruction Code by the compiler. With the original compiler, you could even write a better compiler.

So in order for a computer language to exist, you need two things, a set of words that make up the language (another code,) and a compiler that compiles the written language into computer instruction code.

The language that we have seen in this book has only about 20 words in it. Each word correlates directly to one of the instructions of which this computer is capable. Each line you write results in one computer instruction. When you write an 87 line program in this language, the instruction code file that the compiler generates will have 87 instructions in it.

Then someone invented a "higher level" language where one line of the language could result in multiple computer instructions. For example, our computer does not have an instruction that does subtraction. But the compiler could be designed so that it would recognize a new word in the language like 'SUB RA,RB' and then generate however many machine instructions were necessary to make the subtraction happen. If you can figure out how to do something fancy with 47 instructions, you can have a word in your language that means that fancy thing.

Then someone invented an even higher level language where the words that make up the language don't even resemble the CPU's actual instructions. The compiler has a lot more work to do, but still generates instruction code that does the things that the words in that language mean. A few lines from a higher level language might look like this:

```
Balance = 2,000
Interest Rate = .034
Print "Hello Joe, your interest this year is: $"
Print Balance X Interest Rate
```

The compiler for this language would read this four-line program, and generate a file that could easily contain

To use the file system, there will be some sort of rules that the application program needs to follow. If you want to write some bytes to the disk, you would need to tell the OS the name of the file, the RAM address of the bytes that you want to write, and how many bytes to write. Typically, you would put all of this information in a series of bytes somewhere in RAM, and then put the RAM address of the first byte of this information in one of the registers, and then execute a Jump instruction that jumps to a routine within the Operating System that writes files to the disk. All of the details are taken care of by this routine, which is part of the OS.

If you ask the OS to look at your disk, it will show you a list of all the file names, and usually their sizes and the date and time when they were last written to.

You can store all sorts of things in files. Files usually have names that are made up of two parts separated by a period like "xxxx.yyy." The part before the period is some sort of a name like "letter to Jane," and the part after the dot is some sort of a type like "doc" which is short for "document." The part before the period tells you something about what is in the file. The part after the dot tells you what type of data is contained in this file, in other words, what code it uses.

The type of the file tells both you and the OS what code the data in the file uses. In one popular operating system ".txt" means text, which means that the file contains ASCII. A ".bmp" means BitMaP, which is a picture. A ".exe" means executable, which means it is a program and therefore contains Instruction Code.

If you ask the OS what programs are available to execute, it will show you a list of the files that end with ".exe". If you ask for a list of pictures that you can look at, it will show you a list of files that end with ".bmp".

There are many possible file types, any program can invent its own type, and use any code or combination of codes.

Computer Diseases?

Another place where human characteristics get assigned to computers is something called a computer virus. This implies that computers can come down with a disease and get sick. Are they going to start coughing and sneezing? Will they catch a cold or the chicken pox? What exactly is a computer virus?

A computer virus is a program written by someone who wants to do something bad to you and your computer. It is a program that will do some sort of mischief to your computer when it runs. The motivation of people who write virus programs ranges from the simple technical challenge of seeing whether one is capable of doing it, to a desire to bring down the economy of the whole world. In any case, the people who do such things do not have your best interests in mind.

How does a computer 'catch' a virus? A virus program has to be placed in your RAM, and your computer has to jump to the virus program and run it. When it runs, it locates a file that is already on your hard disk, that contains a program that gets run on a regular basis by your computer, like some part of the operating system. After the virus program locates this file, it copies the virus program to the end of this file, and inserts a jump instruction at the beginning of the file that causes a jump to where the virus program is. Now your computer has a virus.

When a computer with a virus is running, it does all of the things it is supposed to do, but whenever it runs the program that contains the virus, the inserted jump instruction causes the virus program to be run instead. Now the virus usually will do something simple, like check for a predetermined date, and if it is not a match, then the virus program will jump back to the beginning of the file where the operating system program still exists.

Thus, your computer will appear totally normal, there are just a few extra instructions being executed during its regular operations. The virus is considered dormant at this point. But when that date arrives, and the virus

But that isn't the end of the story. The ROM described above had to be built that way at the factory. Over the years, this idea was improved and made easier to use.

The next advance was when someone had the bright idea of making ROM where every bit was set on at the factory, but there was a way of writing to it with a lot of power that could burn out individual connections, changing individual bits to an off. Thus this ROM could be programmed after leaving the factory. This was called 'Programmable ROM' or 'PROM' for short.

Then someone figured out how to make a PROM that would repair all of those broken connections if it were exposed to ultraviolet light for a half an hour. This was called an 'Erasable PROM', or 'EPROM' for short.

Then someone figured out how to build an EPROM that could be erased by using extra power on a special wire built into the EPROM. This was called 'Electrically Erasable PROM', or 'EEPROM' for short. One particular type of EEPROM has the name 'Flash memory.'

So there is RAM, ROM, PROM, EPROM, EEPROM and Flash. These are all types of computer memory. The thing they have in common is that they all allow random access. They all work the same way when it comes to addressing bytes and reading out the data that is in them. The big difference is that RAM loses its settings when the power goes off. When the power comes back on, RAM is full of all zeros. The rest of them all still have their data after power off and back on.

You may ask then, "Why don't computers use EEPROM for their RAM? Then the program would stay in RAM when the computer was off." The answer is that it takes much longer to write into EEPROM than RAM. It would slow the computer down tremendously. If someone figures out how to make an EEPROM that is as fast and as cheap and uses the same or less power as RAM, I'm sure it will be done.

By the way, the word ROM has also come to be used to mean any type of storage that is permanently set, such as a pre recorded disk, as in 'CD ROM,' but its original definition only applied to something that worked just like RAM.

phonograph, radio, television, tape recorders and videocassettes. Oddly enough though, one of the oldest devices, the telegraph, was digital. Now that digital technology has become highly developed and inexpensive, the analog devices are being replaced one by one with digital versions that accomplish the same things.

Sound is an analog thing. An old fashioned telephone is an analog machine that converts analog sound into an electrical pattern that is an analog of the sound, which then travels through a wire to another phone. A new digital telephone takes the analog sound, and converts it into a digital code. Then the digital code travels to another digital phone where the digital code is converted back into analog sound.

Why would anyone go to the trouble of inventing a digital phone when the analog phone worked just fine? The answer, of course, is that although the analog phone worked, it was not perfect. When an analog electrical pattern travels over long distances, many things can happen to it along the way. It gets smaller and smaller as it travels, so it has to be amplified, which introduces noise, and when it gets close to other electrical equipment, some of the pattern from the other equipment can get mixed in to the conversation. The farther the sound goes, the more noise and distortion are introduced. Every change to the analog of your voice becomes a part of the sound that comes out at the other end.

Enter digital technology to the rescue. When you send a digital code over long distances, the individual bits are subjected to the same types of distortion and noise, and they do change slightly. However, it doesn't matter if a bit is only 97% on instead of 100%. A gate's input only needs to 'know' whether the bit is on or off, it has to 'decide' between those two choices only. As long as a bit is still more than half way on, the gate that it goes into will act in exactly the same way as if the bit had been fully on. Therefore, the digital pattern at the end is just as good as it was at the beginning, and when it is converted back to analog, there is no noise or distortion at all, it sounds like the person is right next-door.

I Lied – Sort of

There is one piece of hardware in a computer that is not made completely out of NAND gates. This thing is not really necessary to make a computer a computer, but most computers have a few of them. They are used to change from something that is analog to something that is digital, or digital to analog.

Human eyes and ears respond to analog things. Things that we hear can be loud or soft, things that we see can be bright or dark and be any of a multitude of colors.

The computer display screen that we described above had 320 x 200 or 64,000 pixels. But each pixel only had one bit to tell it what to do, to be on or off. This is fine for displaying written language on the screen, or it could be used to make line drawings, anything that only has two levels of brightness. But we have all seen photographs on computer screens.

First of all, there needs to be a way to put different colors on the screen. If you get out a magnifying glass and look at a color computer or television screen, you will see that the screen is actually made up of little dots of three different colors, blue, red, and green. Each pixel has three parts to it, one for each color. When the display adapter scans the screen, it selects all three colors of each pixel at the same time.

For a computer to have a color screen, it needs to have three bits for each pixel, so it would have to have three times the RAM in order to be able to control the three colors in each pixel individually. With three bits, each color could be fully on or off, and each pixel would therefore have eight possible states: black, green, red, blue, green and red (yellow,) green and blue (cyan,) blue and red (magenta) and green, blue and red (white.)

But this is still not enough to display a photograph. To do that, we need to be able to control the brightness of each color throughout the range between fully on and fully off. To do this, we need a new type of part that we will describe shortly, and we need more bits in the display RAM. Instead of one bit for each color in each

possibilities.

If the brain and the mind are the same thing, you might not be able to build a synthetic person today, but as time went on, eventually you could understand every structure and function in the brain, and build something of equal complexity that would generate true consciousness, and that really should act just like any other person.

If the brain and the mind are not the same thing, then building a robot buddy will always be about simulating humanity, not building something of equal quality and value.

Restating the question doesn't make it any easier to answer, but this idea of separating what we know about minds from what we know about brains may be useful. Early on, we said that we were going to show how computers work so that we could see what they were capable of doing, and also what they were not capable of doing. We are going to take what we know about brains and what we know about minds and compare each individually to our new knowledge about computers. In doing so we can look for differences and similarities, and we may be able to answer a few less controversial questions.

Computers do certain things with great ease, such as adding up columns of numbers. A computer can do millions of additions in a single second. The mind can barely remember two numbers at the same time, never mind adding them up without a pencil and paper.

The mind seems to have the ability to look at and consider relatively large amounts of data at the same time. When I think of my favorite cat, I can re-experience seeing what he looks like, hearing the sounds of his purring and mewing, feeling the softness of his fur and his weight when picked up. These are some of the ways that I know my pet.

What would it mean for our computer to think about a cat? It could have pictures of the cat and sounds of the cat encoded in files on a spinning disk or in RAM. Is

vastly better at math, but the mind is better at dealing with faces and voices, and can contemplate the entirety of some entity that it has previously experienced.

Science fiction books and movies are full of machines that read minds or implant ideas into them, space ships with built-in talking computers and lifelike robots and androids. These machines have varying capabilities and some of the plots deal with the robot wrestling with consciousness, self-realization, emotions, etc. These machines seem to feel less than complete because they are just machines, and want desperately to become fully human. It's sort of a grown-up version of the children's classic "Pinocchio," the story about a marionette who wants to become a real boy.

But would it be possible to build such machines with a vastly expanded version of the technology that we used to build our simple computer?

Optimism is a great thing, and it should not be squashed, but a problem will not be susceptible to solution if you are using a methodology or technology that doesn't measure up to that problem. In the field of medicine, some diseases have been wiped out by antibiotics, others can be prevented by inoculations, but others still plague humanity despite the best of care and decades of research. And let's not even look into subjects like politics. Maybe more time is all that's needed, but you also have to look at the possibility that these problems either are unsolvable, or that the research has been looking in the wrong places for the answer.

As an example, many visions of the future have included people traveling around in flying cars. Actually, several types of flying cars have been built. But they are expensive, inefficient, noisy and very dangerous. They work on the same basic principles as helicopters. If two flying cars have any sort of a minor accident, everyone will die when both cars crash to the Earth. So today's aviation technology just won't result in a satisfactory flying car. Unless and until someone invents a cheap and reliable anti-gravity device, there will not be a mass market for flying cars and traffic on the roads will not be relieved.

So it appears that whichever way we look at it, neither the brain nor the mind work on the same principles as computers as we know them. I say 'as we know them' because some other type of computer may be invented in the future. But all of the computers we have today come under the definition of 'Stored Program Digital Computers,' and all of the principles on which they operate have been presented in this book.

Still, none of this 'proves' that a synthetic human could never be built, it only means that the computer principles as presented in this book are not sufficient for the job. Some completely different type of device that operates on some completely different set of principles might be able to do it. But we can't comment on such a device until someone invents one.

Going back to a simpler question, do you remember Joe and the Thermos bottle? He thought that the Thermos had some kind of a temperature sensor, and a heater and cooler inside. But even if it had had all of that machinery in it, it still wouldn't "know" what to do, it would just be a mechanical device that turned on the heater or cooler depending on the temperature of the beverage placed in it.

A pair of scissors is a device that performs a function when made to do so. You put a finger and thumb in the holes and squeeze. The blades at the other end of the scissors move together and cut some paper or cloth or whatever it is that you have placed in their way. Do the scissors "know" how to cut shapes out of paper or how to make a dress out of cloth? Of course not, they just do what they're told.

Similarly, NAND gates don't "know" what they are doing, they just react to the electricity or lack of it placed on their inputs. If one gate doesn't know anything, then it doesn't matter how many of them you connect together, if one of them knows absolutely zero, a million of them will also know zero.

We use a lot of words that give human characteristics to our computers. We say that it "knows" things. We say it "remembers" things. We say that it "sees," and "understands." Even something as simple as a device adapter "listens" for its address to appear on the I/O

bus, or a jump instruction “decides” what to do. There is nothing wrong with this as long as we know the truth of the matter.

Now that we know what is in a computer, and how it works, I think it is fairly obvious that the answer to the question “But How do it Know?” is simply “It doesn’t know anything!”