Appendix C
Digital Logic

Chapter 3 discussed the differences between analog and digital signals. Without making this an entire digital course, let's briefly describe some of the common digital circuits that are used.

Binary numbers

To review, digital circuits use binary numbers which, in turn, use the binary digits 0 and 1, sometimes also written out as ZERO and ONE. Since a single bit only has two possible values, it does not carry a lot of information, and so most numbers have more than just one bit. A number consisting of n bits can have any one of 2^n possible values. For example, the smallest 8-bit number is 00000000, which equals our decimal number 0. The largest 8-bit number, on the other hand, is 11111111, which equals our decimal number 255. There are thus 256 different numbers ranging from 0 through 255. This makes sense, since we know that an 8-bit number can express 2^8 or 256 different values.

A group of eight bits is usually called a *byte*, while a group of four bits is called a *nibble*. Larger groups of bits are sometimes called a *word*, although there is no agreement on exactly how long the word should be. Computer users often talk of short words, long words, and so on.

Voltage levels

Within a digital circuit, bits are usually carried as a voltage on some wire. Although the specific voltage used could be anything, in practice there are some standard values that are fairly common.

The most common voltage levels use approximately 0 volts for a ZERO, and approximately +5 volts for a ONE. This dates back to the 1960's and 70's, when digital integrated circuits (ICs) first became cheap enough for common use. The most common digital ICs were called Transistor-Transistor Logic, or TTL, and so we often refer to these voltages as TTL levels.

As usual, the real world is a bit more complicated than our simple explanation. For one thing, to allow slight voltage variations without introducing errors, the actual voltages do not have to be exact. For example, the voltage for a 0 need not be exactly 0 volts — it can be anything from 0 up to 0.8 volts in most TTL circuits, and even more in other types of ICs. Likewise, the voltage for a 1 need not be exactly +5 volts — it can be anything from +2 volts up to +5 volts. As long as the bits stay within these limits, they will still be properly recognized. Only if they vary a lot do we get errors. For example, the range from +0.8 volts up to +2 volts is a "no man's land" with TTL circuits that is guaranteed to create havoc.

But there is also another complication. For various reasons, many designers reverse their voltages and use +5 for a 0, and 0 volts for a 1. They may even flip back and forth in the same device. There are some sound technical reasons for doing this, but it does complicate the issue.

To avoid problems, let's therefore change our terminology. Instead of using the words ONE and ZERO, let's call the 0 to 0.8-volt level a *low*, and call the +2 to +5-volt level a *high*. If you insist on thinking of the low as a 0, and the high as a 1, go ahead, but realize that this may sometimes get you into trouble!

TTL voltage levels are still common today; even though many ICs are no longer TTL, they still often use the same voltage levels.

Incidentally, the words high and low can be used with other voltages as well. For instance, a high might be +15 volts, while a low might be –10. The only thing required is that the high voltage really be higher than the low.

Logic gates

Digital devices (and computers as well) are complex devices built from a large number of very simple parts. These simple parts are often called gates. A typical computer, for example, might have millions of these gates. Let's look at some of these.

The Inverter

The *inverter* is a simple circuit which is very similar to a gate, but most of us just call it an inverter, not a gate (other gates have more than one input). An inverter can be built in many different ways; the simplest one needs just one transistor and two resistors. But we don't generally need to show the actual schematic diagram, so we use the block diagram symbol in Fig. C-1 instead, without caring what is actually inside.

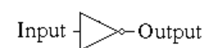

Fig. C-1. Inverter

The job of the inverter is to invert — that is, to change a high input voltage into a low output voltage, or a low input into a high output. Think of the triangle symbol as an arrowhead — it shows that the signal goes from the input side toward the output side. Not also the small circle on the output lead — it is called a bubble, and it reminds us that the output is inverted. It means "invert".

Copyright © 2006 by Peter A. Stark - All Rights Reserved

The inverter is a fairly common circuit used in digital equipment, and its job is to change a signal into its opposite. One way to show this is via a *truth table* like this:

Input	Output
Low	High
High	Low

The purpose of the truth table is to show what the output is for any particular input.

Most beginning books on digital logic automatically assume that a low is a 0, while a high is a 1 — OK for beginners but dangerous in real life, so we won't do it any more after this — so they might show the truth table like this:

Input	Output
0	1
1	0

The basic idea is the same — the output is the opposite or inverse of the input.

The OR Gate

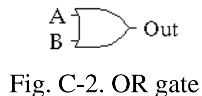

Fig. C-2. OR gate

Fig. C-2 shows the logic symbol for a two-input OR gate. Although OR gates can have more than two inputs, the two-input kind is probably the most common.

The truth table for the OR gate is this:

Input A	Input B	Output
Low	Low	Low
Low	High	High
High	Low	High
High	High	High

With two inputs, there are four possible combinations of highs and lows, so this truth table has four rows of data.

We can see why it is called an OR gate if we explain how it works like this: if input A is high **OR** input B is high, then the output is high. The word OR describes what it does.

The AND Gate

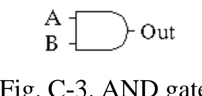

Fig. C-3. AND gate

Fig. C-3 shows the logic symbol for a two-input AND gate. The AND gate could also have more than just two inputs, though two-input gates are probably the most common.

The truth table for the AND gate is this:

Input A	Input B	Output
Low	Low	Low
Low	High	Low
High	Low	Low
High	High	High

Here we have the same four possible combinations of input, but the outputs are different. We can see that this is an AND gate by explaining it like this: if input A is high **AND** input B is high, then the output is high. The word AND describes what it does.

Fig. C-4. Gate symbols

Gates come in many types and sizes, and the easiest way to remember what they do is like this: First, memorize the two symbols in Fig. C-4: the one with straight side is the AND, while the one with both sides curved and the pointed right end is the OR.

Fig. C-5. A bubble

Second, pay attention to bubbles. Fig. C-5 shows a wire without a bubble, and another wire with a bubble. When you see a wire with a bubble, think of the word "low"; when you see a wire without a bubble, think of "high".

Keep this idea in mind as we cover the next gate, the NAND gate.

The NAND gate

Fig. C-5. NAND gate

Fig. C-6 shows the symbol for the NAND gate (which means "Not AND".) One way to figure out what it does is to note that it looks just like the AND, except that it has an a bubble at the output. Since a bubble means "invert", we would expect the output to be the exact opposite of an AND. The truth table shows us that this is exactly what happens:

Input A	Input B	Output
Low	Low	High
Low	High	High
High	Low	High
High	High	Low

Copyright © 2006 by Peter A. Stark - All Rights Reserved

Whereas the AND gate's output column had three lows and then a high, the NAND has three highs and then a low.

There is another way to figure out what the gate does, and that is by remembering our last detour — that the straight-sided picture is an AND, and that bubbles mean low, while no bubble means high. Let's try to express what the gate does in a sentence — read the **bold** words below as a complete sentence, and look at the *italics* for an explanation of why:

Input A doesn't have a bubble, so we say --->	**If input A is high**
the picture is an AND, so --->	**AND**
Input B doesn't have a bubble either, so say --->	**input B is high,**
Now look at the output --->	**then**
Notice the bubble on the output, so think low --->	**the output is low**,
Because the condition above is the only time when the output is low, we add --->	**and only then!**

The sentence "If input A is high AND input B is high, then the output is low, *and only then*" describes how the gate works — it tells us that the bottom line of the truth table gives an output that is low, whereas al other lines above it have a high output.

The NOR gate

Let's try the same process for the gate in Fig. C-6, which is a NOR gate (or a Not OR). We wouldt therefore expect the output to be the exact opposite of the OR gate, as in the following truth table:

Fig. C-6. NOR gate

Input A	Input B	Output
Low	Low	High
Low	High	Low
High	Low	Low
High	High	Low

Instead of the output column being low-high-high-high, it is high-low-low-low, the exact opposite.

If we try to express the operation in a sentence, we get this:

Input A doesn't have a bubble, so we say --->	**If input A is high**
the picture is an OR, so --->	**OR**
Input B doesn't have a bubble either, so say --->	**input B is high,**
Now look at the output --->	**then**
Notice the bubble on the output, so think low --->	**the output is low**,
Because the condition above is the only time when the output is low, we add --->	**and only then!**

Since this sentence describes the bottom three lines of the truth table (where at least one of the two inputs is high), the bottom three outputs must be low, and everything else is high.

An example

Just to make sure you understand this idea, consider the gate in Fig. C-7. This is not a normally available item, but it could be easily built as a custom device. How would you describe its operation in a sentence? How about this: "If input A is high, AND input B is low, then the output is high, and only then." This tells us that the truth table has to have one high, like this:

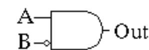

Fig. C-7. What's this?

Input A	Input B	Output
Low	Low	
Low	High	
High	Low	High
High	High	

The "and only then" part then tells us that all of the other lines in the output column must be Low.

The Exclusive OR gate

There is one more kind of simple gate we need to cover —the exclusive OR gate shown in Fig. C-8. It's similar to the regular OR gate, but has an additional curved line at

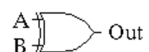

Fig. C-8. Exclusive OR gate

Copyright © 2006 by Peter A. Stark - All Rights Reserved

the input. It also has a slightly different truth table:

Input A	Input B	Output
Low	Low	Low
Low	High	High
High	Low	High
High	High	**Low**

The difference is the **Low** in the last line. The sentence describing the gate also differs in one important phrase: "If input A is high, OR input B is high, **but not both,** then the output is high, and only then."

The difference means this: the output from a normal OR gate is high whenever either input is high, which includes both being high at the same time. In the exclusive OR, however, the output is high if either input is high, but not if both are high at the same time. So the bottom line of the truth table, which covers the case where both inputs are high, is Low for the exclusive OR gate, instead of High.

Boolean algebra

The operation of logic gates can be described with a type of math called Boolean algebra (named after George Boole, a 19th century English mathematician.) When we use Boolean algebra for digital logic, we use some symbols that are very similar to normal arithmetic:

- The OR function is written as a plus sign +. For example, "A OR B" would be written as A+B.
- The AND is similar to a multiplication, so "A AND B" could be written as $A \times B$, or A • B, or (usually) as just AB.
- An inversion is indicated either with a not bar, or with an apostrophe. For example, the opposite of A could be written as either $\overline{A}$ (or A' if the printer or typewriter cannot print a bar above a letter), and it would be pronounced "A not" or "not A". The bar over the A is called a *not bar*.

Logic gates can be described with Boolean algebra as follows. If we assume gate inputs are called A and B, then

- Inverter output = $\overline{A}$
- AND gate output = AB
- NAND gate output = $\overline{AB}$
- OR gate output = A + B
- NOR gate output = $\overline{A + B}$
- Exclusive OR gate output = $A\overline{B} + B\overline{A}$

Some of these look complex, but become quite clear when you think about them. For example, the exclusive OR gate showed that the output was high when the two inputs were *different* (go back and check the truth table!) So let's try to read that last equation, $A\overline{B} + B\overline{A}$, this way: "The output of an exclusive OR gate is high when {A is high AND B isn't} OR when {B is high AND A isn't)".

Whereas we prefer to use the words high and low with logic gates, Boolean algebra is easier to use with 1 and 0 because it is then closer to normal math. For example, look at the truth table we had earlier for the AND gate:

Input A	Input B	Output
Low	Low	Low
Low	High	Low
High	Low	Low
High	High	High

Substitute 1 for High, and 0 for Low, and you get

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

which is just like multiplication.

Do the same for the OR gate:

Input A	Input B	Output
Low	Low	Low
Low	High	High
High	Low	High
High	High	High

Substitute 1 for High, and 0 for Low, and you get

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Except for the last line, this is the same as our normal addition.

Using Boolean equations, you can develop all sorts of identities. For example,

$$A + \overline{A} = 1$$

regardless of what A is. For example, if A=0, then the equation is 0 + 1 = 1; if A is 1, then the equation says that 1 + 0 = 1, so either way we get 1.

Or look at

$$A \times \overline{A} = 0$$

Copyright © 2006 by Peter A. Stark - All Rights Reserved

Regardless of what A is, you will get either $0 \times 1 = 0$, or else $1 \times 0 = 0$.

How about this one:

$$\overline{\overline{A}} = A$$

Inverting A twice changes it first into $\overline{A}$, and then back into A.

Mathematicians can go wild developing more of these identities, such as A + A = A, and so on, but let's talk about one particularly useful one.

DeMorgan's theorem

DeMorgan's theorem is very useful because it often lets a designer replace a complicated circuit (or a complicated Boolean equation) with an equivalent but different one, which might or might not be simpler. It should actually be called DeMorgan's Rule, rather than a theorem, since it is a *rule* for changing an equation or circuit into another one. The rule goes like this:

Suppose you have an expression of the form A + B or A ×B. To change it into its equivalent form, do this:

1. Invert each of the two terms in the equation.
2. Change the AND or OR sign into the other one.
3. Invert the entire expression.

For example, to change A × B into its DeMorgan's equivalent, proceed like this:

1. Change A into $\overline{A}$, and change B into $\overline{B}$.
2. Change the × into +. So far, we have changed A × B into $\overline{A} + \overline{B}$.
3. Now invert the entire expression, which gives us

$$\overline{\overline{A} + \overline{B}}$$

Aside from giving us a different equation which still means exactly the same thing, it also means that the two circuits in Fig. C-9 do the same thing. Is this useful to know? It might be if you had the right-hand circuit and wanted to change it into something simpler.

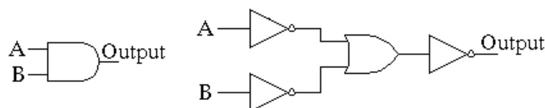

Fig. C-9. Two DeMorgan equivalent circuits

Combining gates

Although the circuit and function of a typical gate can be quite simple, the power comes from combining lots of gates — perhaps even millions — to make a larger system. For example, Fig. C-10 shows two inverters connected together to make a simple circuit called a *flip-flop*.

This diagram shows the typical kind of additional information that would be shown in a real digital circuit. The notation U1a and U1b tells that these are two sections of the same integrated circuit U1; the number 7404 tells that this is a specific kind of IC, and the numbers 1, 2, 3, and 4 give the IC pin numbers.

Flip-Flops

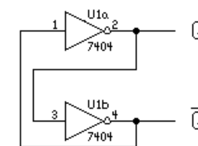

Fig. C-10. Two inverters make a flip-flop

Let's look at Fig. C-10. This circuit has two outputs, called Q and $\overline{Q}$. Since each of the outputs is obtained by inverting the other one, the two outputs must always be opposites of each other. If Q is high, then $\overline{Q}$ must be low; alternatively, if Q is low, then $\overline{Q}$ must be high.

But which one is really high, and which one is low??? The answer is that either output could be high, with the other one low. We say that a flip-flop has two stable states.

Think of a sheet of paper with some writing on one side. You can place the sheet on a table with the writing facing up, or with the writing facing down. Once you put it down, it will stay that way — either one is a stable state. But you can't put it down with both sides up or both sides down at the same time. In the same way, either output of the flip-flop can be high, but not both at the same time.

A flip-flop has two stable states, and will stay in one of those states as long as power is applied, or until you purposely change it. Making the flip-flop change from one state to the other is called *toggling* or *flipping* it.

It is fairly easy to flip the circuit of Fig. C-9 — just force one of the outputs to the opposite state. For example, if Q is high, then grounding it with a wire (i.e., forcing it to go low) would flip it to the opposite state. In order to be able to describe these states easily, we say that a flip-flop is *set* when Q is high (and therefore $\overline{Q}$ is low), and is *reset* or *cleared* when Q is low (and therefore $\overline{Q}$ is high).

Although you can wire a simple flip-flop from two inverters or two gates, most flip-flops come prewired

Copyright © 2006 by Peter A. Stark - All Rights Reserved

with some additional circuitry added to make them easier to operate.

Type D Flip-flop

One common kind of flip-flop is called a Type-D flip-flop, whose block diagram symbol is in Fig. C-11.

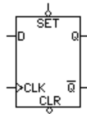

Fig. C-11. Type D flip-flop

The flip-flop is shown as a rectangle, with a number of inputs and outputs along the edges. Let's look at these in turn:

- Q and $\overline{Q}$ are the two outputs. Just as before, they are always opposites of each other.
- SET and CLR are two inputs. Grounding the SET input will immediately set the flip-flop, whereas grounding the CLR pin will immediately reset or clear it (but trying to set or reset the flip-flop when it is *already* set or reset does nothing). Note the bubbles — they remind us that grounding (i.e., making them low) is what is needed to make the flip-flop act.
- The D and CLK inputs work together, and can also set or reset the flip-flop, but in a slightly different way. The D input decides *which way* to flip it, whereas the CLK input decides *when*.

Most of the time, the flip-flop ignores the D input. But when there is a pulse on the CLK input (or more precisely, at the instant that there is a rapid voltage change from a low to a high) then the flip-flop takes a quick look at the D input and makes the Q output the same as D. That is, if the D pin at that instant is high, then the flip-flop sets and makes Q high as well; if the D input is low at that instant, then the flip-flop resets and thus makes Q low (and, of course, $\overline{Q}$ is always the opposite.)

In the D flip-flop block diagram, the small triangle on the CLK input reminds us that that pin is sensitive to the rising edge of a signal — the edge when the voltage goes from a low voltage to a high. This *edge-sensitive* nature of that input is what makes the D flip-flop particularly useful. It means that we can rather precisely control just when the flip-flop operates.

Counters

If we connect a D flip-flop as shown in Fig. C-12, we can make it toggle each time it receives the rising edge of a clock pulse on its CLK input. The explanation is quite simple: If the flip-flop is originally reset, then its $\overline{Q}$ output is high, and so is the D input. Thus at the clock pulse, the flip-flop will set. Once it is set, now the $\overline{Q}$ output is low, so at the next clock pulse the flip-flop will again reset. This will repeat at every clock pulse, so it will always flip to the opposite state at every clock pulse's rising edge.

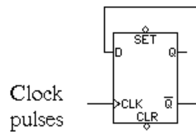

Fig. C-12. Toggling a flip-flop

Another way to look at it is this: If the flip-flop starts reset, then after one clock pulse it will be set; after two pulses it will be reset again; after three pulses it will be set, and so on. For instance, if the flip-flop ends up set, then you know there was an odd number of clock pulses.

Now suppose we connect two of these D flip-flops as shown in Fig. C-13. Every time that flip-flop A resets, its Q output goes from low to high, and this toggles flip-flop B. So flip-flop A toggles at every clock pulse, but flip-flop B toggles only every second clock pulse.

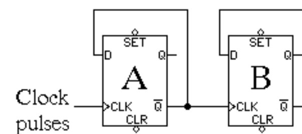

Fig. C-13. Two-stage counter

If we assume that a reset flip-flop holds a 0, while a set flip-flop holds a 1, here is what is happening:

	Flip-flop B	Flip-flop A
Initially, both flip-flops start reset or 0 --->	0	0
At the first clock pulse, flip-flop A sets, which is a 1 --->	0	1
At the second clock, A resets, and this makes B flip --->	1	0
At the third clock pulse, A sets, but B stays the same --->	1	1
At the fourth clock pulse, A again resets, and this makes B flip back to reset, and we're back at the beginning --->	0	0

Now look at those numbers at the right: we start with 00, then have 01, then 10, then 11, and then back to 00 again. If you look at these numbers, they are simply the binary numbers for 0, 1, 2, 3, and then back to 0 again. In other words, the circuit is counting the input clock pulses. But after the fourth pulse, it runs out of two-bit

Copyright © 2006 by Peter A. Stark - All Rights Reserved

numbers, and so it starts all over again — we say that it *overflows* at that point.

If we had three of these flip-flops, then it would take eight pulses before it started out all over; it would go like this:

000
001
010
011
100
101
110
111
000

Thus flip-flops are handy for counting pulses. This time, with three flip-flops we were able to count eight pulses (counting from 0 to 7), so it overflows after 8 pulses. In general, a counter with n flip-flops would overflow at 2^n pulses, and the maximum number it counts to is $2^n - 1$. For example, a counter with 24 flip-flops could count up to 16,777,215, which is $2^{24} - 1$.

Registers

Once a flip-flop is set or reset, it will stay that way until you either turn the power off, or until you intentionally flip it to some other state. It can therefore act as a memory, but it will only hold one bit. To store a number consisting of more than one bit, you need more than one flip-flop. A group of flip-flops which holds a number is then called a *register*. For instance, eight flip-flops together can hold an eight-bit byte.

Although you can build a flip-flop out of individual components, they generally come in integrated circuits. A simple IC might hold just two flip-flops; on the other hand, a static RAM (Random Access Memory) IC might hold a million or more flip-flops, along with the support circuits to select a particular group of flip-flops, write data into them, and read data back out.

Shift registers

Data can be written or read out of a flip-flop register in either parallel or serial. With parallel data transfer, a number of bits is moved at one time along a parallel set of data wires called a *bus*. With serial data transfer, the bits are moved, one after another, along a single wire. Moving data into or out of a register in serial requires wiring the flip-flops as a *shift register*, as in Fig. C-14. This figure shows a four-bit shift register, but a larger register would just need more flip-flops connected the same way. Shift registers are actually a very important part of many digital communications systems, since digital data is usually transferred in serial over long distances.

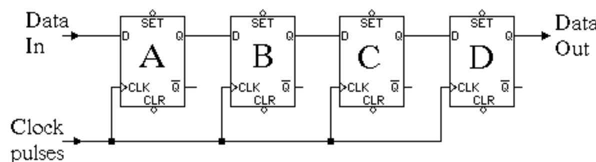

Fig. C-14. Four-bit shift register

In Fig. C-14, all of the CLK inputs are connected together to a source of clock pulses. If, for example, the data is being transferred at 9600 bits per second, then the clock pulses would also come at a rate of 9600 pulses per second.

The D input of each flip-flop (except the first) comes from the Q output of the previous flip-flop. Thus, at each clock pulse, the input data coming in at the left would be transferred into flip-flop A; at that same instant the bit that was in A moves into flip-flop B; at that same instant whatever was in B now moves into C, and so on. At the same time, whatever bit was in the last flip-flop moves out the right end. So this circuit can be used for both input into flip-flop A, as well as output from flip-flop D.

The process of moving a bit from stage to stage is called *shifting*; hence the name shift register. Fig. C-14 shows a shift-right register; with minor changes we could also build a shift-left register. And with a few additional components, the register could be modified to also do parallel input or parallel output, which would make it ideal for converting data to and from serial and parallel.

Computer organization

Computers have gotten more and more complex over the years, but their basic organization has stayed the same for a long time. Since most of modern communications requires a computer of some kind, let's just look at their basic construction.

If you have a computer at home, you are probably familiar with such concepts as hard drives, keyboards, monitors, operating systems, etc. But there are many computers which do not have any of these. These are the so-called *embedded systems*, which are very small computers built into many other pieces of equipment. For example, every modern DVD player, microwave oven, or even cordless telephone has a small computer inside it. Some of these may have a few integrated circuits, others may be all inside one IC. Without trying to

Copyright © 2006 by Peter A. Stark - All Rights Reserved

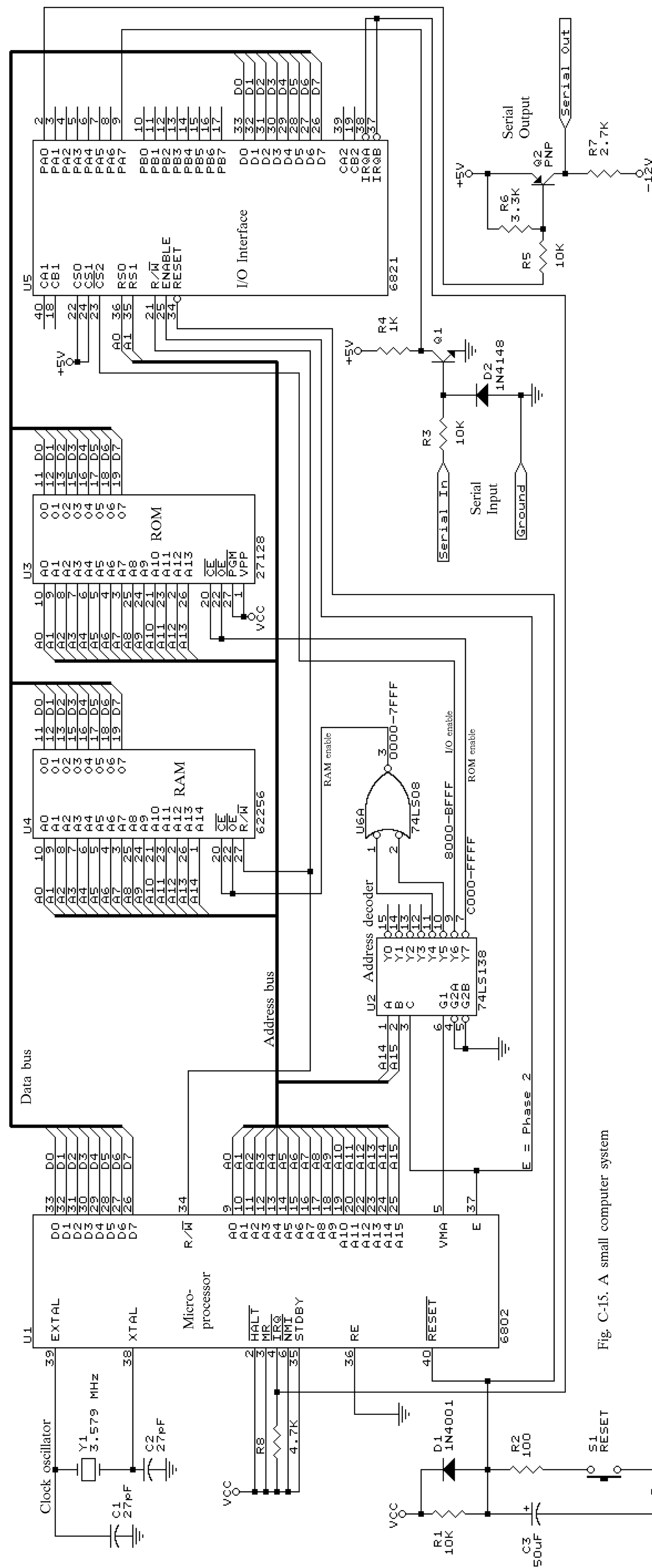

Fig. C-15. A small computer system

make you a computer expert, let's look at Fig. C-15 just to get an idea of what makes such a small computer tick.

Near the top left is the heart of the system; integrated circuit U1 is the microprocessor. It contains a number of registers which temporarily hold data and instructions that are being worked on. Its timing is controlled by a clock oscillator in the top left corner, which uses a 3.579 MHz crystal. We use this particular frequency simply because it is cheap! In the bottom left is a circuit which resets the computer, making it start over either when the power is first applied, or when the RESET switch S1 is pressed.

To the right of the microprocessor is a RAM IC, a ROM IC, and an I/O interface IC. The RAM is a read-and-write memory which contains (in this case) 32,768 bytes of memory. Remember that small embedded computers don't need anywhere the vast amounts of memory that a desktop system might have.

The ROM is a read-only memory which (in this case) contains only 16,384 bytes of memory. It contains program instructions and other data that we now call firmware — it is put into it by the manufacturer, and that does not change over time unless the manufacturer issues an update.

The I/O (or Input/Output) Interface is an integrated circuit that acts as the interface between external input and output devices, and the microprocessor. In our particular case, it connects to a serial input and a serial output down at the bottom right.

The RAM, ROM, and I/O each contain many different locations, each of which has a binary address. The microprocessor keeps track of these addresses to make sure that any information read from these ICs, or any information written into them, comes from or goes to the proper place.

The microprocessor obeys the instructions in a program. Although the program may have been written in various ways and various languages, before the computer can use it, these instructions must be translated into a binary form called

Copyright © 2006 by Peter A. Stark - All Rights Reserved

machine language. These binary instructions sit in memory, and are read out one by one as they are performed.

The microprocessor has an internal register which keeps track of the next instruction to be performed. When it is ready to do it, it sends out that address on the *address bus*. In this particular computer, the address is a 16-bit number which comes out of the microprocessor on sixteen pins labelled A0 through A15. The bus itself is shown as a thick line in the middle of the diagram, and only broken out into each individual wire as it enters a particular IC.

Various parts of the address go to different places. For example, two bits (A15 and A14) go to a circuit called the *address decoder*. This circuit looks at part of the address to decide whom the microprocessor wants to use. If this address is intended for the RAM, then the address decoder sends a ‘wakeup’ signal called RAM Enable to the RAM; if it is for the ROM or I/O, then it sends out either a ROM Enable or an I/O Enable. The address decoder acts as a sort of traffic cop to get the right part of the system to respond.

When the microprocessor is beginning to work on an instruction, the instruction is most likely in the ROM, so the ROM Enable signal will wake up the ROM. The other address bus bits go to the ROM to tell it which specific instruction the microprocessor wants. That instruction is then sent back to the microprocessor on the *data bus*, shown at the top of the diagram. In this case, the data bus contains just eight lines, labelled D0 through D7. In desktop computers, the address bus and data bus may contain many more lines, which allows them to access more addresses and carry more bits at a time. Small embedded computers seldom need that much size or speed.

In any case, when the microprocessor gets the instruction from the data bus, it then looks at the instruction to decide what needs to be done.

Most instructions will tell the microprocessor to either get more data from the memory, or to send some data out. In that case, the address bus will specify the address that it wants to access, and the data bus will carry the actual information to be read or written. Other times, an instruction may tell the microprocessor to do some internal operation, such as adding two numbers or shifting the number in some register to the left or right.

There is a lot more that we could say about this diagram, but let’s just cover a few little details. Notice, for example, the pin labelled $\overline{\text{RESET}}$ at the bottom left of the microprocessor. The not bar over the name tells us that this is an inverted signal — that is, it has to be brought low to ground in order to actually do a reset of the microprocessor — you can see that this is what the reset switch S1 would do. If you follow the reset line over to the I/O interface IC, you will see that over there the same signal is labelled RESET without the not bar, but that it has a bubble on the input. This too tells us that this line must be low to actually do a reset.

Take a look also on the right side of the microprocessor, where there is a line labelled R/$\overline{\text{W}}$. Notice that the not bar only extends over the W, not the R. This wire is called the read/write line, and the terminology means that the microprocessor is reading from memory when the line is high, and is writing when the line is low. R means read when high, whereas $\overline{\text{W}}$ means write when low.

Positive and Negative Logic

Take a look at that strange OR-looking gate, U6A, in the address decoder of Fig. C-15. This gate looks like an OR, but look carefully — it’s not! It has three bubbles on it. It is actually an AND gate! This is worth explaining.

Let’s define some words:

- *Positive logic* is a circuit where a 1 is more positive than a 0. For example, virtually all beginner textbooks call +5 volts a 1, while 0 volts is a 0, so their authors only talk about positive logic. In our computer, the data bus and address bus are both positive logic.
- *Negative logic* is a circuit where a 1 is more negative than a 0. Remember that we said at the beginning of this chapter that sometimes the 1 is 0 volts, while the 0 is +5 volts? That would be negative logic.
- *Active high* is a circuit where a high voltage does something.
- *Active low* is a circuit where a low voltage does something. In our computer, the $\overline{\text{RESET}}$ signal has to be low to do an actual reset, so it is active low.

Gate U6A would be an AND gate if we were talking about positive logic or active high. Unfortunately, here it is in a place that is active low. The RAM Enable signal happens to be active low — the RAM needs a low signal on this wire to turn it on (we didn’t mention this before, but this signal goes into a pin labelled $\overline{\text{CE}}$ which means Chip Enable, and into $\overline{\text{OE}}$, which means Output Enable. Both of these have a not bar, and so they both need a low to work.)

Let’s describe this gate in a sentence using the technique we learned earlier:

Copyright © 2006 by Peter A. Stark - All Rights Reserved

Pin 1 has a bubble, so we say ---> **If pin 1 is low**

the picture is an OR, so ---> **OR**

Pin 2 also has a bubble, so ---> **pin 2 is low,**

Now look at the output ---> **then**

Notice the bubble on the output, so think low ---> **the output is low**,

Because the condition above is the only time when the output is low, we add ---> **and only then!**

But compare this with the truth table for an AND — whenever either input is low, the output is low, so the only place where it can be high is when both inputs are high:

Pin 1	Pin 2	Output
Low	Low	Low
Low	High	Low
High	Low	Low
High	High	High

Perfect match!

We could have come to the same conclusion by using DeMorgan's theorem. When we discussed DeMorgan's theorem earlier, we already showed that

$$A \times B = \overline{\overline{A} + \overline{B}}$$

Just substitute pin 1 instead of A, and pin 2 instead of B, and you have it: Take pin 1 and invert it, take pin 2 and invert it, then OR the two together, and then invert the entire output, and you get the right side of this equation. If you remember that a bubble means invert, then that is exactly what gate U6A in Fig. C-15 does — same as an AND.

One last comment ...

The computer circuit in Fig. C-15 shows how the various parts of a computer interact, but don't try to build it. The actual implementation is hopelessly out of date. For just a few dollars, today we can buy an entire microcomputer on a chip that has all of these components, and more, inside a single integrated circuit. Some of the smaller micros come in eight-pin packages that are smaller than just one gate was thirty years ago. That's progress!

Copyright © 2006 by Peter A. Stark - All Rights Reserved