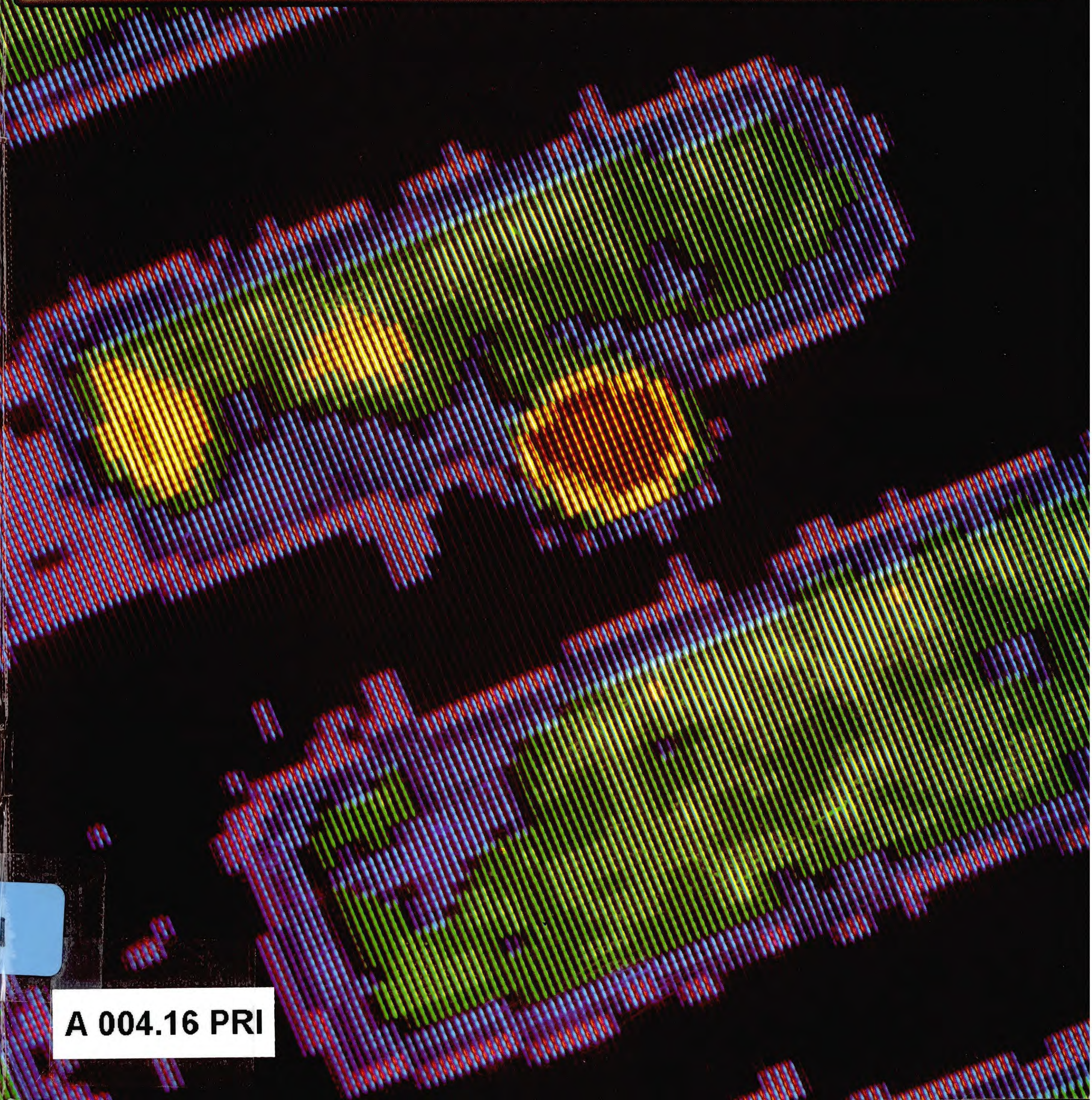
REVISED

NUFFIELD ADVANCED SCIENCE

PHYSICS

MICROCOMPUTER CIRCUITS AND PROCESSES



General editor,
Revised Nuffield
Advanced Physics
John Harris

Consultant editor
E. J. Wenham

Author of this book
Colin Price

2y 2,

PHYSICS MICROCOMPUTER CIRCUITS AND PROCESSES

REVISED NUFFIELD ADVANCED SCIENCE

Published for the Nuffield–Chelsea Curriculum Trust
by l

Longman Group Limited
Longman House, Burnt Mill, Harlow, Essex CM20 2JE, England
and Associated Companies throughout the World

Design and art direction by Ivan Dodd
Illustrations by Oxford Illustrators Limited

Filmset in Times Roman and Univers and made and printed in Great Britain at The Bath Press, Avon

ISBN 0 582 35423 4

Cover

The photograph on the back cover shows individual gates of a microchip made visible by EBIC (electron beam induced current) inside a scanning electron microscope. A defective gate (no glow) shows up amongst operating cells (fluorescent glow).

The photograph on the front cover shows the defective cell of the microchip analysed by an elemental analyser built into the electron microscope. The material distribution at the defective emitter of the cell is mapped by computer on a colour monitor screen in colours representing relative material thickness.

The width between adjoining tracks in each case is 10 μm.

Courtesy ERA Technology + Micro Consultants/Link Systems

Photographs: Paul Brierley

CONTENTS

PROLOGUE

This Reader explains how microcomputers work. It does not refer to any one computer, but all ideas discussed are relevant to any computer. It begins with ...

CHAPTER 1 where the history of the microprocessor is traced out from ancient to modern times. It continues with ...

CHAPTER 2 which looks at the electronics of a microcomputer, building up a simple machine from scratch.

CHAPTER 3 describes how a small program stored in memory actually gets all the electronics working, to carry out the programmer's desires.

CHAPTER 4 is about how microcomputers can be used to make measurements and display results of computations. There are also some notes on computer memory.

Although no reference is made to any commercial microcomputer, all the circuits drawn in this Reader were tested by building a small computer. Any differences between the text and reality are small, and intended to make everything a little clearer for you. If you know nothing about these things, I hope you learn something. If you are already a microprocessor boffin, there will still be some pieces of interest.

I must thank the Intel Corporation of Santa Clara, California, for supplying photographs and numerous other data for use in this Reader. And they make good microprocessors too. Thanks to David Chantrey for help in correcting the original text, any remaining mistake is my own.

Colin Price

CHAPTER 1

A HISTORY OF THE MICROPROCESSOR

SOWING THE SEEDS

The revolution now shaking every aspect of life from banking and medicine to education and home life began in 1947 in the Bell Laboratories with the invention of the transistor by Brattain, Bardeen, and Shockley. This small amplifier (which now in modern memory circuits can be made very small – 6 µm × 6 µm) quickly replaced the large power-consuming vacuum tube. The evolution of the microprocessor from the humble transistor illustrates the relationship between technology and economics, and also how the technology of electronic device production is related to the desired applications of electronics at any time.

For example, it is difficult to build a practical computer which needs a large number of switching circuits using vacuum tubes. But when the transistor was invented, the situation at once changed. In the 1950s the transistor did replace vacuum tubes in many applications, but as far as building computers was concerned, there still remained the problem of the myriad of connections which had to be made between individual transistors. That meant time and materials. Also, studies made in the early 1950s suggested that only 20 computers would be needed to satisfy the World's needs. The market did not exist, and technology was then unable to stimulate it. The solution to the interconnection problem was the *integrated circuit*, invented by Jack Kilby of Texas Instruments in 1958, and Bob Noyce, at Fairchild in 1959.

THE INTEGRATED CIRCUIT INDUSTRY

Manufacturing an integrated circuit (IC) involves the 'printing' of many transistors, with a sprinkling of resistors and capacitors, on to a wafer of silicon. Those in the trade call this printing process *photolithography* and *solid-state diffusion.* Using this technology, hundreds of very small, identical circuits can be made simultaneously on one wafer of silicon. These circuits are bound to be cheap. But most important is the ability to print the interconnections between the individual transitors on to the silicon chip, removing the need for manually wiring the circuit together. So costs were again reduced. As reliability of the circuits improved dramatically, so maintenance costs fell too.

Such economic trends inspired manufacturers to research into further miniaturization, and in 1965, Gordon Moore, later chairman of Intel Corporation, predicted that during the next decade the number of transistors per integrated circuit would double every year. This exponential 'law' held good, and remained good into the 1980s. The graph in figure 1.1 shows this law, from the first IC of 1959 containing just one transistor, through the logic circuits of 1965 and the 4096-transistor memory of 1971, to the 4 million-bit 'bubble' memory introduced in 1982. Of course the rate of doubling has slowed recently, but there is still anticipation of further miniaturization.

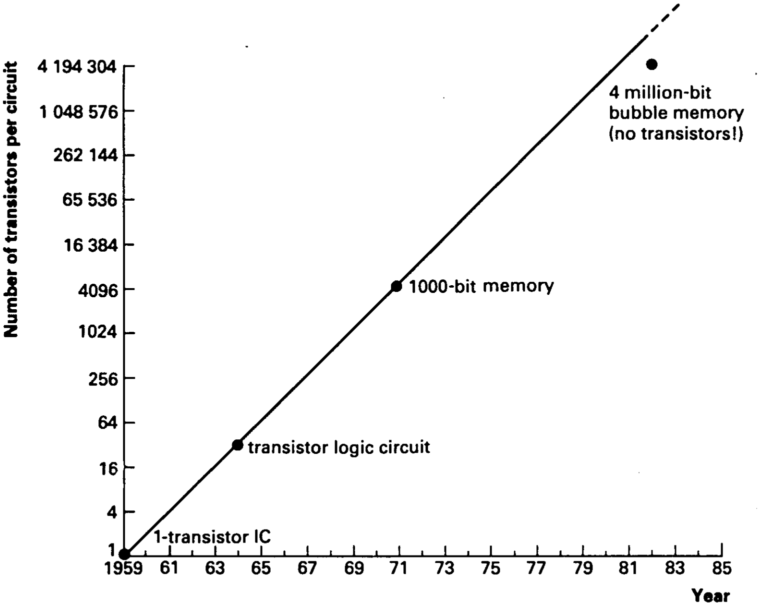


Figure 1.1
Graph showing 'Moore's Law' – the doubling of number of transistors per IC per year. The 4-megabit bubble memory introduced in 1982 does not use transistor technology.

As any industry grows and gains experience its production costs fall, but the IC industry has been unique in achieving a constant doubling in component density coupled with falling costs. How has this been achieved? It is due to the IC concept, replacing transistor-based circuits in traditional equipment, producing smaller, more reliable, easily assembled devices. Much of the success of the IC industry has come from stimulating new markets. A good example is computer *memory*. Up to the end of the 1960s, computer memory was made from small rings of magnetic material sewn together by electrical wires. It worked, but compared with the component densities being achieved in the IC, the package was bulky. People in the semiconductor industry realized this, and understood that the requirements of computer memory – a large number of storage cells connected with a small number of leads – could be met by specially designed IC chips. Companies were founded with the sole purpose of memory manufacture, and as miniaturization continued and prices fell, semiconductor memory became established as the standard. Thanks to these devices, the large, room-sized computers of the 1950s and 1960s, with their hundreds of kilometres of wiring, could be made smaller and more powerful. The mainframe and minicomputers were born.

BIRTH OF THE MICROPROCESSOR – A SOLUTION TO A PROBLEM

The electronic systems manufacturers of the late 1960s were able to produce quite interesting products by engineering custom-designed ICs, each doing a specific job. Several tens or even hundreds of chips and other components were connected together by soldering them onto printed circuit boards. As the complexity of designs increased, this method of production became very expensive, and could only be employed when large production runs were involved, or else for government or military applications. A radically different approach to construction was needed.

In 1969, Intel, one of today's leading producers of microprocessors, was commissioned to design some dedicated IC chips for a Japanese high-performance calculator. Their engineers decided that, using the traditional approach, they would need to design about 12 chips of around 4000 transistors per chip. That was pushing the current technology a bit (see figure 1.1) and would still leave a nasty interconnection problem. Intel's engineers were familiar with minicomputers and realized that a computer could be programmed to do all the clever functions needed for this calculator. Of course, a minicomputer was expensive and certainly not hand-held. But they also realized that they had, by then, sufficient technology to put some of the computer's functions onto a single chip and connect this to memory chips already on the market. The first microprocessor, the Intel 4004, was born.

But how exactly did this solve the calculator design problem? It was all a question of how to approach a problem. Instead of the 12 calculator chips, each performing a specific job, one microprocessor chip would be made which could carry out a few very simple tasks. The different complex functions needed for the calculator could be produced by the microprocessor being made to carry out its simple tasks in a repeating but ordered fashion. The instructions needed by the microprocessor to do this would be stored in memory chips, sent to the microprocessor, and implemented. Seemingly a long-winded procedure it could be made to work since ICs can be coaxed along at very high speeds, needing only microseconds to do a single task.

The implications of this alternative approach were very profound. The same microprocessor as designed for the calculator could be used in endless other devices – watches, thermostats, cash registers. No new custom-designed ICs would be needed, the 'hardware' would remain intact; only the program put into memory, the 'software', would need to be changed to suit the new application. Manufacturers would have a high-volume product on their hands, and soon would have smiles as wide as a television screen.

Such innovation did not catch on overnight. Intel's 4004, which was heralded as the device that would make instruments and machines 'intelligent', did not achieve that status. A great improvement came in 1974 with the 8080 microprocessor, designed by Masatoshi Shima (who later designed the Z80 for Zilog, the now famous hobby microprocessor). The 8080 had the ability to carry out an enormous range of tasks (large computing 'power'), was fast, and had the ability to control a wide range of other devices. The modern version of the 8080, the 8085 (which you can buy for a few pounds) is firmly established as one of the standard 8-bit processors of the decade. No bigger than a baby's fingernail, it comes packaged in a few grams of plastic with 40 terminals tapping its enormous potential (figure 1.2).

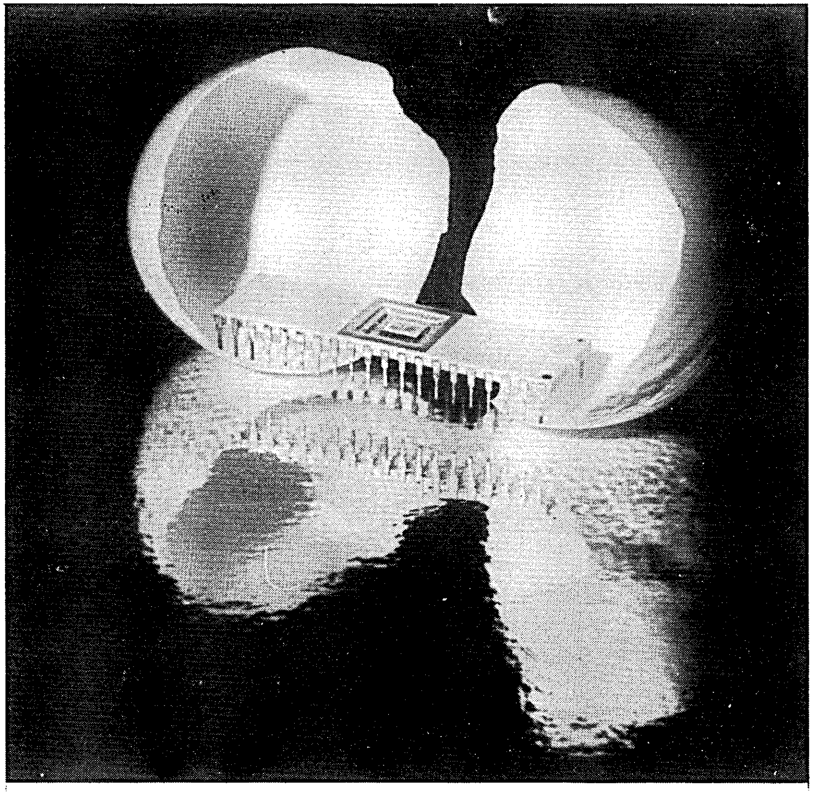


Figure 1.2
The birth of the 8085 microprocessor chip.
Intel Corporation (UK) Ltd

RESPONSE TO THE FIRST MICROPROCESSOR

In mechanics, momentum must be conserved; but in the electronics industry it seems to increase! It had taken only three years since the introduction of the 4004 for the total number of microprocessors in use to exceed the combined numbers of all minicomputers and mainframes. In 1974 there were 19 processors on the market, and one year later there were 40. Different manufacturers aimed at different markets: RCA developed a CMOS (Complementary Metal Oxide Semiconductor) processor, using practically no current; Texas Instruments developed a 4-bit processor for games applications. Advances in design included putting the memory, input/output circuits, and even analogue-to-digital converters on the processor chip, resulting in single-chip microcontrollers. Figure 1.3 outlines the development of the industry reminding us of the reducing size of the elements thereby increasing chip density.

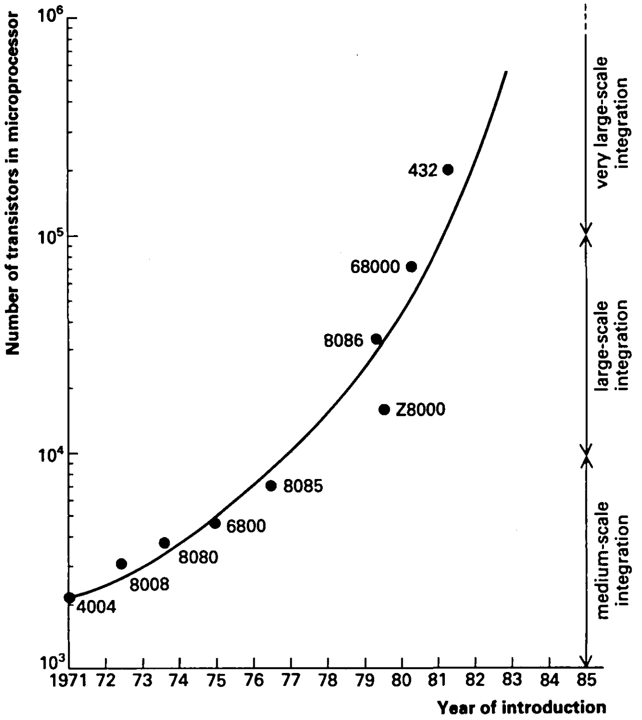


Figure 1.3
Evolution of the technology of making ICs is shown by the increasingly large transistor densities being obtained.

By the 1970s between 1000 and 10 000 transistors could be put on a single chip. This *medium-scale integration* (MSI) was used to make the 4004, 8080, and 8085 processors. *Large-scale integration* (LSI), 10 000 to 100 000 transistors on a chip, dominated the market in the late 1970s and early 1980s. It was used in the production of processors such as the 16-bit 8086 and 8088 used by IBM in their personal computer launched in 1984. Large-scale integration will be followed by *very large-scale integration* (VLSI) as more than 100 000 transistors are put on to a chip. The iAPX 432 contains 225 000 transistors and is the product of 20 million dollars and 100 worker-years of development. This processor, on a single chip, has all the power of a minicomputer which would fill a wardrobe-sized cabinet. It can execute 2 million instructions per second.

APPLICATIONS – THE CREATION OF NEW MARKETS

When the initial 4004 development was under way, marketing departments envisaged the microprocessor as being sold only as minicomputer replacements, and made sales estimates of only a few thousand per year. In 1981 sales of the latest 16-bit processors rose above the

800 000 mark. Such high sales resulted from the creation of new markets as microprocessors appeared. The great boom in digital watches, pocket calculators, and electronic games of the late 1970s is a good example of such a market. In electronic instrumentation (oscilloscopes, chromatographs, and surveying instruments, for example) which had been a stable, mature market, the microprocessor brought along a rebirth; instruments not only would make measurements but analyse the data as well.

It is now impossible to escape from the processor. This is perhaps most evident in the consumer area: games, inside television sets, hi-fi sets, and video recorders, and of course personal computers all depend on them. In medicine the microprocessor directs life-saving and life-support equipment, and has enabled the modern science of genetic engineering to develop. In commerce, word processing, telephone exchanges, and banking systems rely on the microprocessor; indeed, future developments lie in this area. Complete office units will be microprocessor based, where dictation, passing interdepartmental memos, and so on, will all be managed by microcomputers. The cashless society, where the familiar 'cash-card' will contain a memory chip containing details of your accounts, is almost upon us. Supermarkets will no longer have tills but will automatically debit you via such a card.

Towards the more academic pursuits, Artifical Intelligence is being seriously researched. Microprocessor-controlled voice-synthesis chips are on the hobby market and voice recognition and visual shape recognition are being investigated.

TODAY'S PROBLEMS ARE TOMORROW'S DEVELOPMENTS

Little has been said so far of the software side of the microprocessor industry. Remember that when the microprocessor was born, so also was the need to program it, to make it work. The history of the chip, as described above, is one of increasing complexity and power while reducing cost. But as the processor became more powerful, so the complexity of the programs increased. Software became very expensive, as you will know if you have ever bought games for your own computer. Different manufacturers have different strategies, but there are two very obvious ones at the time of writing. Firstly, when a manufacturer has just marketed a microprocessor, he knows that the programmers will be busily writing operating systems, compilers, business packages, and the like; and the research and development departments will already be busy on the next generation of hardware. Now all of this activity must be co-ordinated so that programs written on today's machine will be more-or-less compatible with tomorrow's machine, so they can run with the minimum of change. Such a philosophy will be welcomed by the buyer of a particular microprocessor system who does not want to throw out his computer every five years. Of all the manufacturers of microprocessors around, those who are most successful are the ones that pursue a policy of open-planned development.

The second strategy chosen by some manufacturers is a subtle move away from software by replacing some programs by special chips designed to carry out certain functions. In the light of what has been said, you may think this sounds like going into reverse gear, but I did say it was a subtle move. A good example is the 8087 Numeric Data Processor, marketed by Intel. This chip, a microprocessor in its own right, sits inside a system quite close to the central microprocessor, which could be an 8086. The 8087 hangs around until a mathematical job is to be done, anything from adding to computing a trigonometric function. Then it springs into action, doing the mathematics about 100 times faster than a software program stored in memory.

The microprocessor revolution has touched all of us, from the company executive who has everything programmable to the farmer in India who depends on satellite weather pictures. In about a decade, it has become possible for you to buy, for several pounds, a microprocessor chip as powerful as a room-sized machine; and all of this by return of post. I hope you will try it out one day.

CHAPTER 2

THE ELECTRONICS OF A MICROCOMPUTER

If you spend a few thousand pounds on buying a minicomputer – a wardrobe-sized cabinet where the processing is done by several boards full of medium-scale integrated circuits – or if you spend a few hundred pounds or less on a microcomputer, where the processing is done by an LSI microprocessor like the 8085, then your computer will always have three types of circuit:

a central processing unit (CPU);
some memory;
input and output devices – keyboard, television screen, printer, etc.

This chapter is about how these three types of circuit are connected together. We shall think about a small microcomputer, where the CPU is a microprocessor, just like the one in the laboratory or even in your home.

GETTING IT TOGETHER – THE BUS CONCEPT

The three circuit types must be connected together; figure 2.1 suggests two ways of doing this. The first suggestion involves wires between each device. This may work, but would be clumsy. In the second suggestion, each device is connected in parallel to a set of wires called a *bus*.

The bus is drawn on circuit diagrams as a wide channel and, in reality, consists of many parallel wires carrying signals. Each device – CPU, memory, keyboard, television screen – is connected to or 'hangs on' the bus.

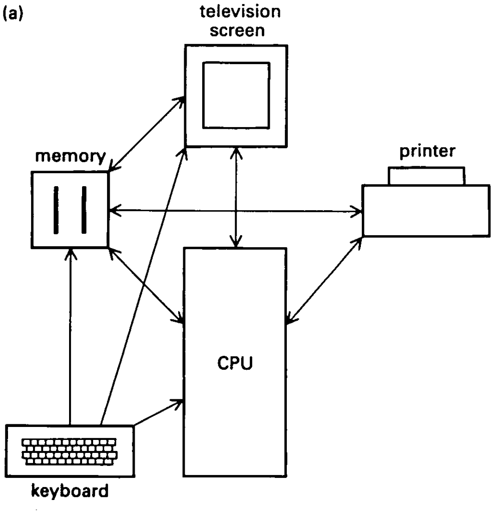


Figure 2.1(a)
Interconnecting computer devices: wires connect each device.

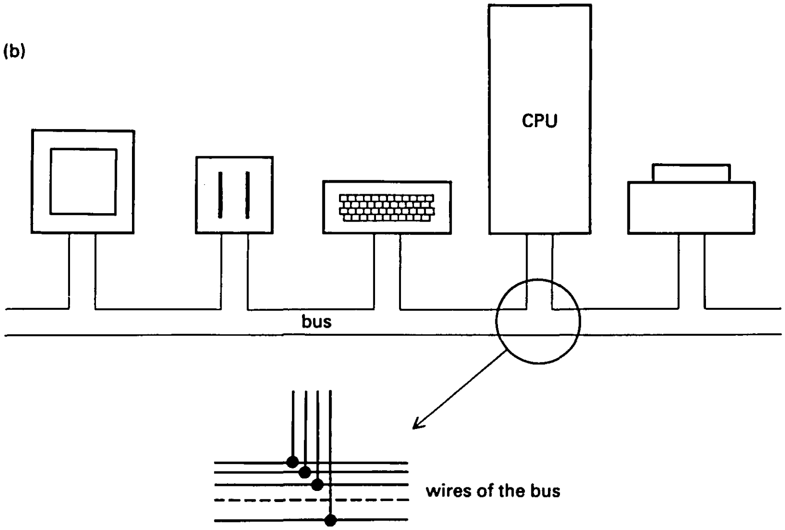


Figure 2.1(b)
Interconnecting computer devices: the efficient bus connection.

It saves a lot of wires, but there is just one problem. What happens if two devices put their signals onto the bus simultaneously? This is shown in figure 2.2, where just one wire of the bus is shown, and two logic inverters are shown connected. The output of one is high and the output of the second is low. What happens?

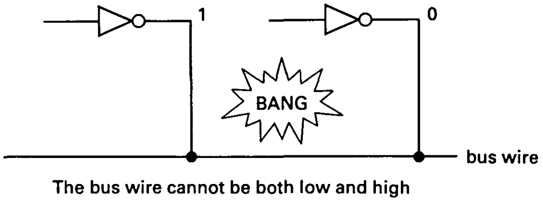
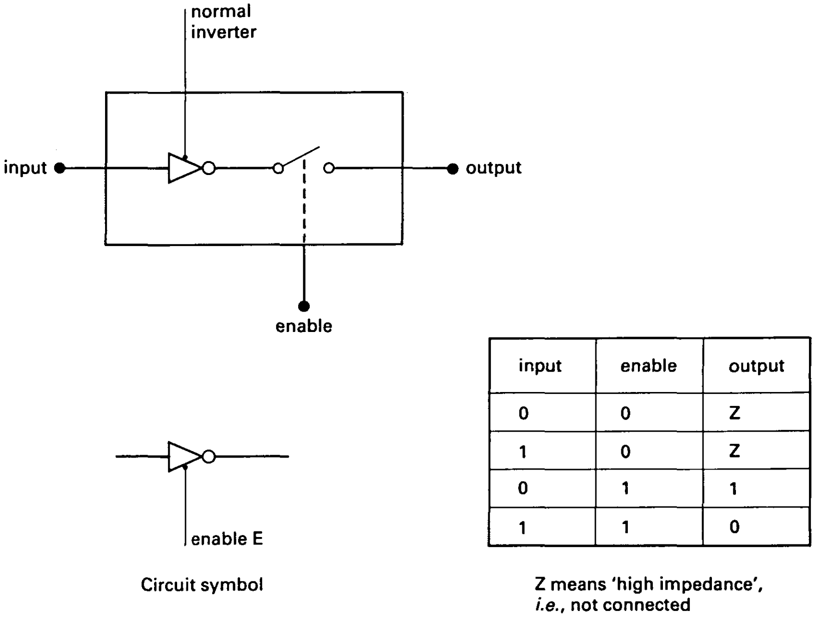


Figure 2.2
There is a problem when two devices put different signal levels on to the bus. This situation must be avoided.

Almost certainly, something will get very hot and be damaged; at the very least the system will not work. This is called *bus contention*, and steps must be taken to ensure that no two devices put their signals on the bus together. In technical jargon, no two devices may 'talk' to the bus simultaneously. This is achieved using a type of logic gate called the '*Tri-State*' gate (which is a trademark of National Semiconductor Corporation). A tristate gate does not have three different logic states, but is like a normal logic gate in series with a switch. Look at figure 2.3 which shows a tristate inverter gate. The idea is that a control signal is able to connect the gate to its output. This is called 'enabling' the gate. If the *enable* is low, then there is no output from the gate, neither high nor low. The gate is in a 'high-impedance' state, meaning that anything connected to its output does not know it is there. When the gate is enabled, as the truth table in figure 2.3 shows, it behaves just like a normal inverter.



input	enable	output
0	0	Z
1	0	Z
0	1	1
1	1	0

Z means 'high impedance', *i.e.*, not connected

Figure 2.3
The Tri-State buffer. (Tri-State is a trademark of the National Semiconductor Corporation.)

To solve our problem of bus contention, *tristate buffers* are used. You can think of these simply as switches, many of them controlled by a common enable. Figure 2.4 shows the beginnings of the computer system, with the CPU, a printer, and a television screen connected to the bus. The printer and television screen hang on via tristate buffers. The CPU contains its own buffers and can hang on the bus when it wants to.

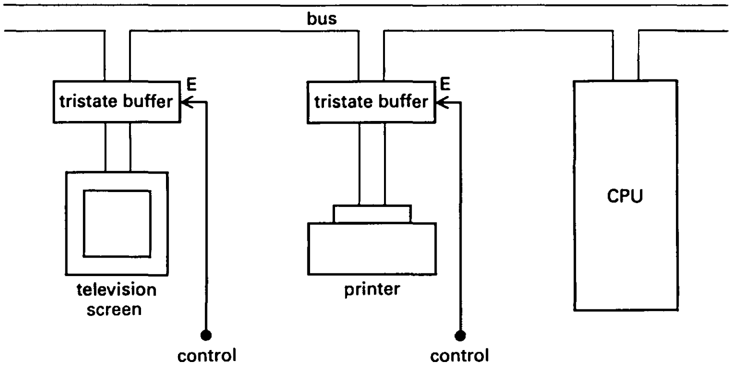


Figure 2.4
Connecting to the bus via tristate buffers.

Look again at figure 2.4. You will see that the buffers' enable terminals are connected to wires labelled 'control', which must ultimately be connected to the CPU. The CPU will decide when either device will be enabled on to the bus. As you continue reading this chapter, you will realize that the CPU has a lot of controlling to do, and its control connections to the various devices on the bus are therefore sent down the control bus. You will meet the control signals on this bus one by one.

The situation so far is shown in figure 2.5. We have not given a name to the top bus; think of it for the moment as an 'information' bus used to shunt numbers to and fro. So how many wires should be used in this bus, or put another way, how 'wide' should it be? Well, you know from binary arithmetic that to write the numbers 0 to 10 in binary, you need four binary digits or *bits*. For example $2_{10} = 0010_2$ and 9_{10} is 1001_2.* So if we only wanted to send numbers from 0 to 10 down the bus, 4 wires would do. So it was in the days of the 4-bit 4004. But if you want to send English characters and words down the bus then you need a wider bus. Taking 8 bits gives at once $2^8 = 256$ possible symbols. That sounds a bit more like it, but there is a bit more to it than that, as we will now see when we look at computer memory.

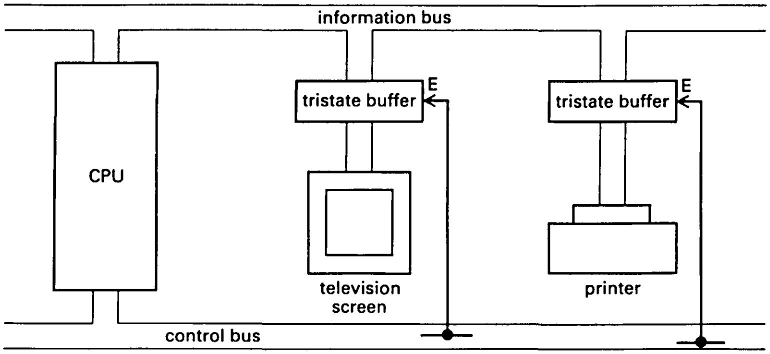


Figure 2.5
Addition of control bus provides signals to enable the buffers.

A FIRST LOOK AT MEMORY

The basic requirement of memory is to be able to write some information into it, leave it there, and come back later to read it. This can be done with floppy disks, tape, or pencil and paper, but here we are concerned with the type of memory all computers have, semiconductor memory. The basic unit of memory is a single storage cell which may hold binary 0 or binary 1. We will not worry about what electronics could be in the cell just yet – you probably know one arrangement of gates or flip-flops that could do the job. It must be possible to read and

* 2_{10} (read as 'two base ten') means 2 in our common base-10 counting system. 0010_2 (read as 'one zero base two') $= (1 \times 2^1) + (0 \times 2^0) = 2_{10}$. And 1001_2 is $(1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 9_{10}$.

write from and to these cells, and to choose which cells you are interested in. These cells, one of which is shown in figure 2.6, form *random access memory* (RAM).

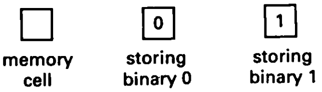


Figure 2.6
Cells of RAM (random access memory).

To make up a useful memory chip, these cells are packed into a square array. We shall think about a 16 by 16 cell array (256 cells in total) for simplicity; a modern 2164 memory chip contains 4 groups of 128 by 128 cells, giving 65 536 cells in all! A rather smaller memory array is shown in figure 2.7.

Controlling the memory cells are two blocks of logic circuits: a set of row selectors on the side and the column select at the bottom. These circuits pick out one particular cell from the 256 by specifying on which one of the 16 rows the cell lives and on which one of the 16 columns it lives. So, both row and column select have 16 outputs. Now 16 is 2^4, so only 4 bits of information are needed to address any of the 16 rows or columns. That is why each select block is driven by a 4-bit binary number, as shown in figure 2.8.

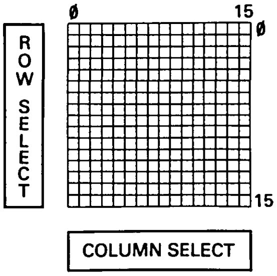


Figure 2.7
Structure of a 256-cell memory chip.

Here, the row select is being driven by binary 0100 (4_{10}) and the column select is being driven by binary 0011 (3_{10}), so the cell being addressed is row 4, column 3. (When looking at the diagram, do not overlook the existence of row 0 and column 0.) In this way, each memory cell may be specified by two 4-bit binary numbers, and this is called the *address* of the memory cell, in this case 0100 0011.

Now that we are able to address memory, we must look at how data is got in and out of the cells. One way of doing this on the memory chip is to put some in–out circuits next to the column selects, as in figure 2.9. This block has a data-in wire and a data-out wire. If data in = 1 and a write into memory is performed, then the 1 is stored in the cell defined by the two 4-bit address numbers. Similarly, if the contents of cell 0111 1101 are needed, this address is sent to the select logic, and the contents of this cell are read out via data out.

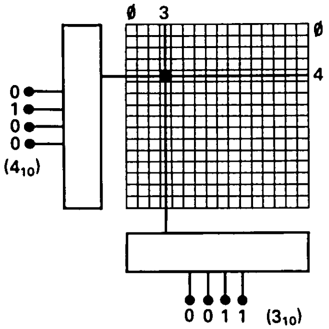


Figure 2.8
Addressing one cell of a 256-cell memory chip.

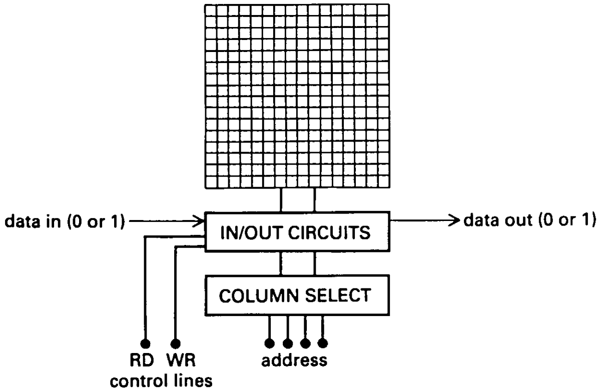
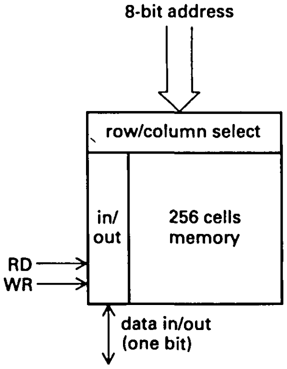


Figure 2.9
Additional circuitry to get data in and out of 256-bit memory.



'igure 2.10
ɩepresentation of 256-bit memory as it ıay appear in a manufacturer's)atabook. Note address lines, data line, nd read (RD) and write (WR) lines.

To perform reads and writes, the in–out circuits must receive some control signals, defining READ or WRITE. These come from the CPU and are put on to the wires labelled RD (read) and WR (write). The READ and WRITE signals form part of the control bus signals.

The whole memory chip is complete, and is shown again in figure 2.10. Note how the two 4-bit address numbers have been written as a single 8-bit address. This memory chip will be able to store 256 different numbers, but each number may be only 0 or 1. As before, to be able to store numbers and letters we would like to store 8 bits at a time. To do this we need 8 of these memory chips, as shown in figure 2.11. Each chip is responsible for storing one bit of an 8-bit word. For example, if we wanted to store the word 01011011, then the first chip would hold 0, the second 1, the third 0, and so on. If all the address lines of the chips are bussed in parallel, then each chip will store its own bit of the 8-bit word at the same address. That makes life easy. To store an 8-bit number in memory, you first give an 8-bit address (which each memory chip splits into two 4-bit row–column selects) and then send the 8-bit data word whose bits each go to a memory chip.

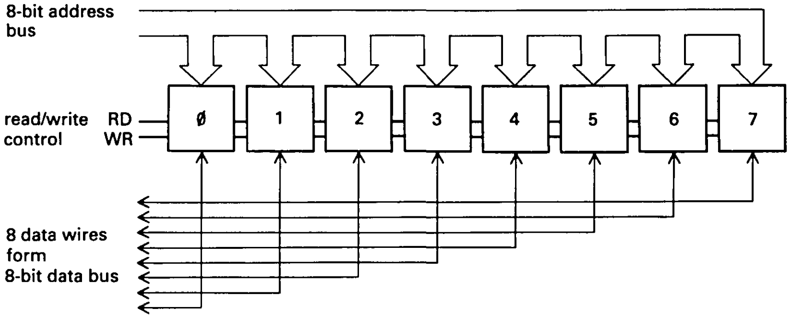


Figure 2.11
Joining 8 of the 256-bit memory chips to make a 256-byte memory board. Note how the data bus is born.

We have rediscovered the bus. The address wires can be taken to the CPU as an *address bus* and the data wires as a *data bus.* If you look back at the bus mentioned in figure 2.5, which we called an 'information' bus, you will now understand that it is really an address bus plus a data bus. Before we increase the size of the memory one step further, look again at figure 2.11. Note that the read (RD) and write (WR) wires are all connected to common, respective, RD and WR lines. That is because we want all the chips to respond simultaneously to a read or a write operation. The whole 256-byte memory package can be redrawn. In figure 2.12 it is shown connected to the address bus and, via a tristate buffer, to the data bus of the computer system. Actually, manufacturers put the tristate buffer in the data lines on to the chip to make life easy for constructors. At this point, make sure you understand why there are no buffers needed on the address lines for the memory joining the address bus.

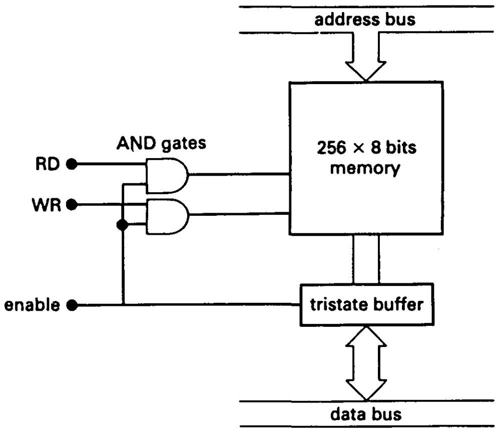


Figure 2.12
256-byte memory from figure 2.11 shown as a single board with buffer, and control lines, and logic.

Note also how the *enable* connection, which controls the tristate buffer hanging on the data bus as usual, is also ANDed with the read (RD) and write (WR) connections. If the enable is high, then RD and WR may perform their functions, but if the enable is low, RD and WR have no effect and the memory cells remain unchanged. This is useful when larger memory systems are designed, as you will find out.

MORE MEMORY – DEVICE SELECTION

We have put together 256 bytes of memory quite quickly, but that is not an awful lot, especially when you consider that a Sinclair ZX home computer comes with 16 or 48 kilobytes. Before we see how to wire up larger amounts of memory, there is the vocabulary to be sorted out. First, an 8-bit number like 01110100 is called a *byte*. Our 256-byte memory size is often called *a page* of memory, and four pages make up *1 kilobyte* (or 1K, pronounced 'kay' as in 'OK'). Although 4 × 256 = 1024, this is what computer people mean when they talk about 1K. So 64K of memory means 64 × 1024 = 65 536 bytes. It may sound like feet and inches, but at least everyone, including you, knows what it means.

To make up 1K of memory all we need is four 256-byte boards. But we are only able to address one of these with our 8-bit address bus. To address the other boards, the address bus has to be expanded. If we doubled the size of the address bus to 16-bits wide, then we would have more potential. You may think we would use each of the extra eight wires on the bus to select a 256-byte memory board. This is shown in figure 2.13, and would give us a maximum of 8 × 256 = 2048 (2K) bytes of memory. Each of the extra address lines is connected to the enable input on a memory board. This would work, but is not normally done, for two reasons. Firstly, the memory addresses will not be continuous as you go from one page to another. For example, the second board would have addresses from 0010 00000000 (512_{10}) to 0010 11111111 (767_{10}), and the third from 0100 00000000 (1024_{10}) to 0100 11111111 (1279_{10}). There is a nasty gap between 767 and 1024. Secondly, this arrangement is wasteful of power. There are $2^8 = 256$ possible combinations of 8 bits, and if these were decoded properly, they could control 256 pages of memory, *i.e.* 256 x 256 bytes, the magic 65 536 (64K) bytes.

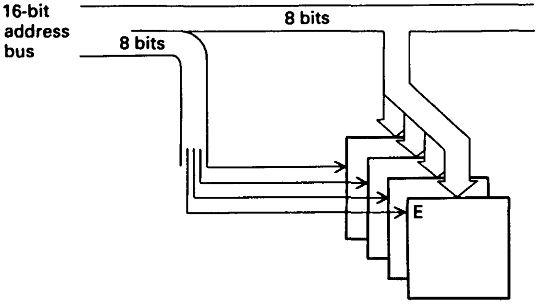
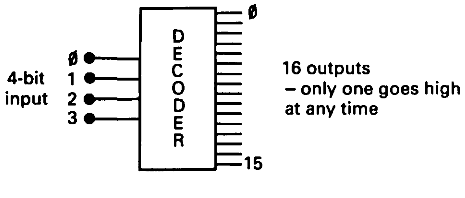


Figure 2.13
Selecting each memory board using extra address lines. 4 boards from a maximum of 8 are shown. Note that each board shown here, and in all following diagrams, has its own gates and buffers, as in figure 2.12. E is the enable input connection.

Most computers will employ special decoding chips which will allow this full use of the 16-bit address bus. The resulting memory addresses run continuously from 0 to 65 536. To apply this technique to the memory, let us just *decode* four of the extra eight address bits, giving us $2^4 = 16$ signals which we can use to select the memory boards. The chip which does the job is called a *1-of-16 decoder*, and contains some sixteen, 4-input NAND gates and a sprinkling of inverters. It has 4 input wires and 16 outputs. Each output will go high for one combination of inputs on the 4 input wires. Each 4-bit binary code will cause one output to go high. Figure 2.14 shows the decoder and its truth table.



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

Figure 2.14
Circuit diagram for a decoder, and its rather large truth table.

If you get the chance, look up 74154 IC in a databook and check the logic inside it. The decoder's 4 inputs are connected to 4 bits of the extra address bus, and the outputs to the memory board enables, only the first 4 being used here to get our 1K of memory.

Figure 2.15 shows these connections. Remember we also had to decide how to connect the read (RD) and write (WR) control signals. That is simple, since each memory board only responds to RD and WR when it is selected by its enable via the decoder. The decoder only selects one board at a time, so the RD and WR signals only reach one board at any time. Stated simply, the enable signal overrides both RD and WR signals.

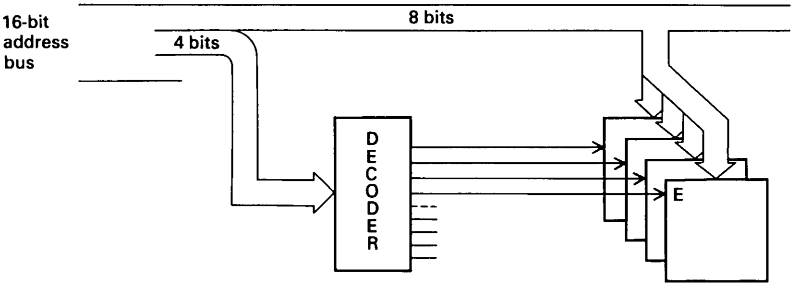


Figure 2.15
Use of a decoder to address memory boards. Here, 4 extra memory lines are decoded, giving a total of 16 addressable memory boards.

The use of a decoder to select the correct memory board is just one example of decoding addresses. The same technique can be used in selecting input and output chips, as you will see. Some commercial equipment uses this technique for selecting whole subsystems, such as data acquisition systems, measurement modules, robot control boards, advanced graphics boards, and the like.

TIMING DIAGRAMS

We have seen how a microprocessor interacts with some memory via the address and the data buses and via the RD and WR signals of the control bus. For the microprocessor to work properly things must happen at the right time: for example, it is no good sending data along the data bus to memory, then some time later sending the address where the data was to be stored. Obviously the correct sequence is first to send out the address on the address bus, and then send out the data. Finally, a pulse has to be sent out on WR to actually write the data into the memory. This is an example of *timing*.

It is the job of the CPU to do the timing. That is why there is a clock on the CPU chip producing a square wave of frequency around 2 MHz. These clock pulses drive complex logic circuits inside the CPU which decide when to output address, data, RD, WR, and other signals onto the correct bus. Connecting an oscilloscope with several traces to the CPU signals is the next best thing to stripping down a CPU and dissecting its logic. Figure 2.16 shows the format of the display.

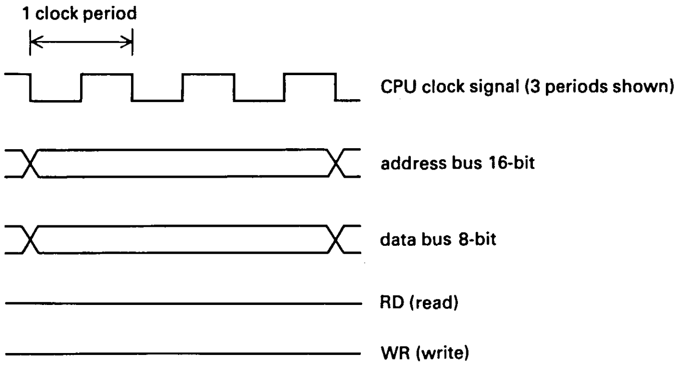


Figure 2.16
The format of a *timing diagram.* The standard way of drawing various signals is shown; the lines are the 'axes' of working timing diagrams. RD, WR, and clock signals are drawn as single lines, since these signals travel along single wires. The buses are drawn as full bars since these involve up to 16 wires.

The top trace shows the CPU clock ticking away. The next two traces show what is on the address and data buses respectively. Since these buses are 16 and 8 bits wide, they are shown as full bars not lines on the diagram. The crossings at either end show that the bits on the bus change at that time. At the bottom are the read and write control lines. In figure 2.16 the CPU is not doing anything, so most traces are not changing. Here is an example of the CPU being asked to send some data to memory and write it there. Figure 2.17 shows what the oscilloscope would reveal as the CPU performed its task. It can be followed period by period for the three clock periods.

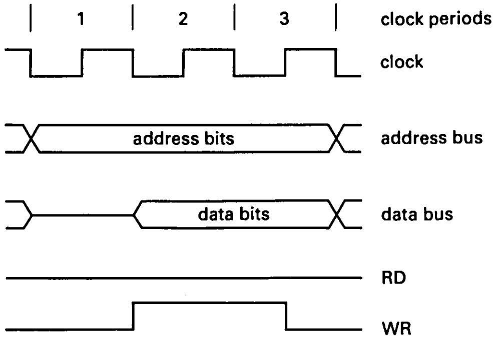


Figure 2.17
Timing diagram for a WRITE cycle. This is easily seen as the WR signal goes high.

Period 1 The address is put on the bus straight away. The data bus is tristated and disconnected (shown by the,line). RD and WR are both low. Eight of the sixteen address bits reach the memory and another four have been decoded, selecting the correct memory board. The correct memory cells are now addressed.

Period 2 The data bits are put on to the data bus and the write (WR) line goes high. The data bits reach the memory chip, where they are written into memory as WR is high. They are written in the memory location set up during Period 1.

Period 3 By this time, the data has been written into memory and so the WR pulse can go low. That is the end of the operation.

Note how, throughout the entire operation, the address bits were always there on the bus. That made sure the data went into the correct memory cell.

Perhaps just one more example may help you appreciate the beauty of the system. Figure 2.18 shows a READ operation, taking data out of a certain memory address and loading it into the CPU.

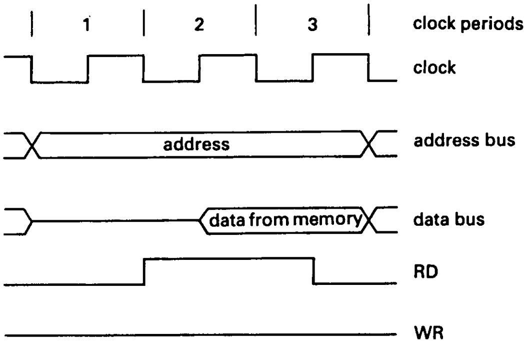


Figure 2.18
Quite similar to the WRITE cycle shown in figure 2.17, this is a READ cycle, getting data from memory to CPU. Note how the RD signal goes high.

Period 1 Again, the address is first to be put on to the address bus. This goes down the bus, selects the appropriate memory board, enables it, and selects the cells which contain the desired data. The data bus is tristated.

Period 2 The CPU now issues a READ signal, raising RD to high. Soon after, the memory responds by putting the byte from the addressed cells on to the data bus. These pass along the bus into the CPU.

Period 3 The data is now stable inside the CPU and so the RD line is brought low, completing the operation.

A WORKING COMPUTER

It is worth pausing just a while and assembling all of the work so far onto one circuit diagram (see figure 2.19). Check that you understand the functions of the RD and WR signals, and the memory enable signals. Check that you know why the data bus is 8 bits wide, and the address bus 16, and check you know how the decoder works.

The final task is to add circuits that will control the input and output devices: keyboards, television screens, etc. This is described in the final section of this chapter. The next section is about some more advanced memory and bus techniques. You could skip this on your first reading.

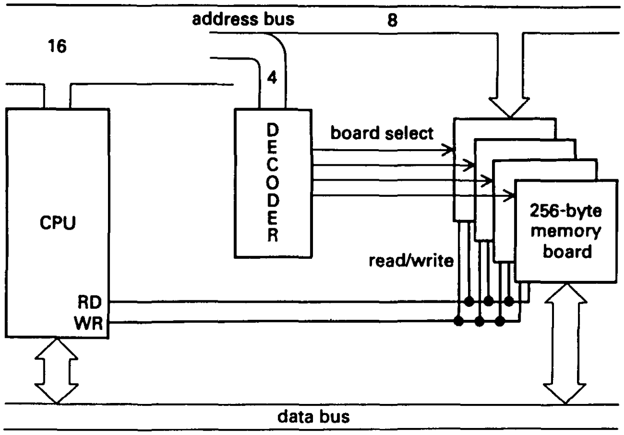


Figure 2.19
Pausing for breath, here is the state of our computer so far. Check that you understand how all the control signals work.

BUS MULTIPLEXING – AN INCREASE IN EFFICIENCY

In building up the computer the need for more and more connections to the CPU is discovered. There are 16 address lines, 8 data lines, and 2 control lines so far. The CPU needs power, giving 2 more lines; a clock crystal, that is another 2. We are already up to 30 lines, and the standard IC package has only 40 pins. If the memory capacity of the computer is to be increased, that means more address lines. We are running out of pins. So we have to think about sharing one of the buses. If we use the data bus to carry both 8 bits of data and 8 bits of the address, then we need only have an 8-bit address bus. We save 8 pins. Figure 2.20 shows this arrangement.

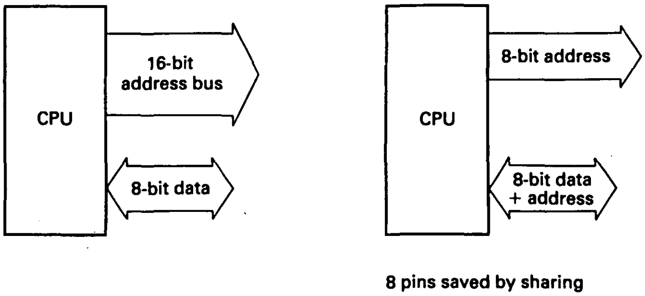


Figure 2.20
Sharing the bus. On the left is our bus system as described previously. On the right is shown how the lower bus can be shared between data and address information.

Of course the data bits and address bits cannot be haphazardly mixed; the sharing has to be organized. This careful sharing is called *multiplexing*.

To see how it is done, look again at the timing diagram for a WRITE operation, reproduced in figure 2.21. The 16-bit address bus transfers the full 16 bits of address throughout the operation, but the 8-bit data is only needed later on, after the address bits are stable.

So what about putting 8 of the address bits onto the data bus during the first clock cycle, before the data is put onto the data bus? Then the 16-bit address bus may be shrunk to 8 bits. This multiplexing will work, and is used in commercial processors.

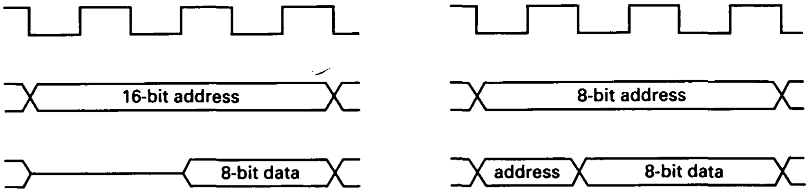


Figure 2.21
Timing diagrams for WRITE operations; on the left the standard WRITE cycle, showing a gap on the data bus. On the right is shown how address information is put into this gap, thus multiplexing the bus.

Of course, another control signal will be needed to say when the multiplexed address–data bus is carrying address bits, and when it is carrying data bits. This signal is called *ALE*. The timing diagram for the WRITE operation now looks like figure 2.22.

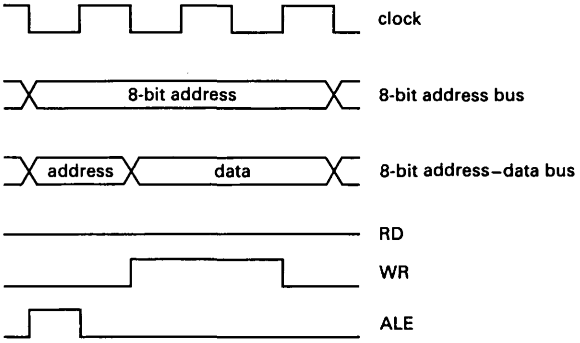


Figure 2.22
Timing diagram for a WRITE operation for a multiplexed bus system. Note how the ALE pulse identifies when the address–data bus contains address information.

Note how the ALE control line goes high when the address–data bus contains address bits, and goes low before data bits appear on the bus. Now, how does memory respond to a multiplexed bus? Not very well, until it is de-multiplexed at the memory end. Remember that the memory needs 16 bits of address information throughout the whole operation. On the multiplexed address–data bus 8 of the address appear for a moment, then vanish. They must be held on to. This is a job for a *latch* circuit.

A typical latch has 8 inputs and outputs and an enable connection. When the enable is high, the outputs are equal to the inputs; they 'follow' the inputs. But the instant the enable control goes low, the outputs do not change any more, even though the inputs do. This is illustrated in figure 2.23.

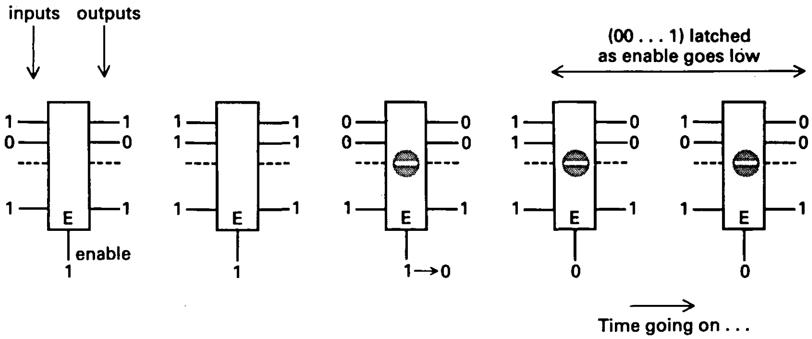


Figure 2.23
The latch. Going from left to right, the diagrams indicate a few nanoseconds in the life of a latch: the data input is shown continuously changing. The latch output follows the input, or is held stable, depending on the enable signal.

If an 8-bit latch is fed with the multiplexed address–data bus, enabled to let through the address information which appears first on the bus, then de-enabled, it will hold the address information thereafter. What is needed is a signal to do this which is high when the address is on the shared bus. The ALE pulse is the right signal. ALE stands for *address latch enable.* See figure 2.24.

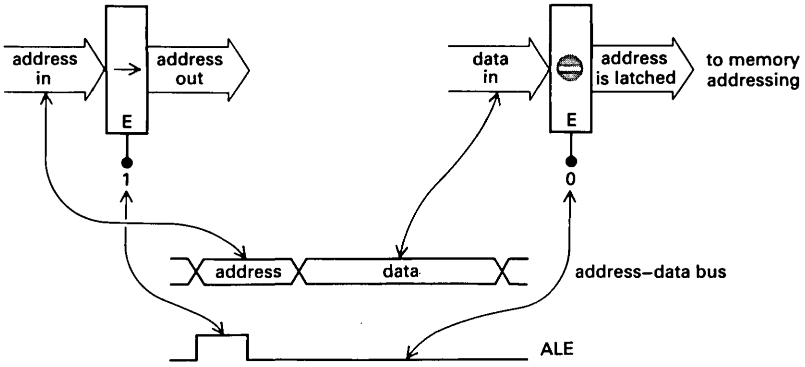


Figure 2.24
How the ALE signal latches the address information, so that it is available to the memory throughout the complete cycle.

The latched address information is then fed to the memory in the normal way. That is fine for the address, but the address bits are still on the address–data bus during the first CPU clock cycle, and so will get into memory and be taken as data. Fortunately there is no problem since data is only written into memory when WR goes high. A glance at the timing diagram in figure 2.22 will convince you that WR goes high only after the address bits have disappeared from the address–data bus.

A lot of trouble you may say, but a total of 7 pins and a lot of wiring have been saved, and the speed of transfer has not been reduced.

INPUT AND OUTPUT

Devices that input bits of information into the computer are vital: to get the program loaded into memory in the first place, to input users' commands via the keyboard, and to make measurements on the external (real, non-computer bussed) world. Similarly, output devices are needed to drive television screens, printers, and circuits which control robots making cars.

Many input and output devices transfer their information in whole bytes, like the keyboard and television shown in figure 2.25. These need to be hung on the data bus, via the usual tristate buffer. In the case of an ouput the CPU must select the correct device by pulsing the enable high while sending out data, on the bus, which is then picked up by the device.

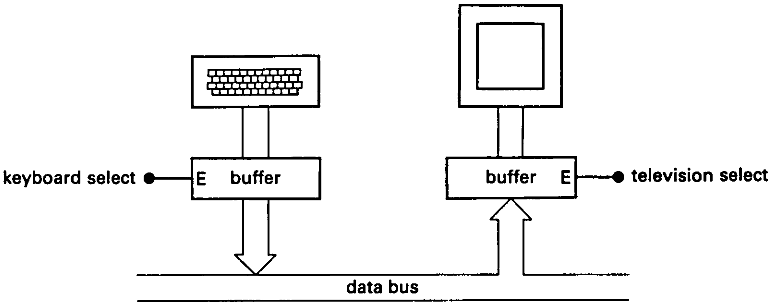


Figure 2.25
How input and output devices are hung onto the bus via tristate buffers. This implies the need for more control signals to select the devices.

The only connection to be made is a signal to enable the output buffer. One way of doing this is to think of the output device as a location in memory which may be written to by a normal WRITE operation. In that case, the output buffer can be enabled by a decoded memory address. Of course, no real memory must live at this address – that would cause bus contention. The method is shown in figure 2.26. It has the disadvantage of using up memory space, and needs a large

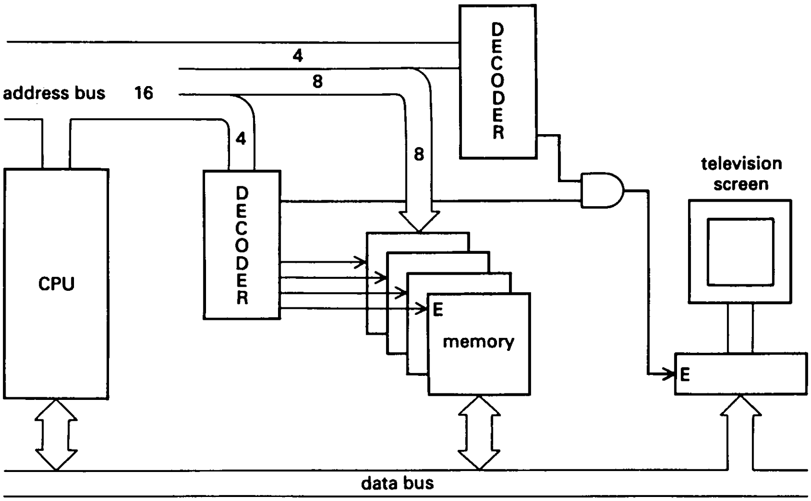


Figure 2.26
Selecting an output device by decoding one memory address. The output device looks like a memory cell to the CPU.

amount of decoding in big systems. It has the advantage of looking like memory to the CPU. Now the CPU spends most of its time shoving bytes between itself and memory. So it does that well: it has many memory-oriented operations which the user may use in a program. If input–output looks like memory, then these operations can be applied also to input–output. Input–output devices which are enabled in this way are called *memory-mapped* devices.

Most processors are designed with special input and output facilities so that the input–output circuits are quite separate from memory. It adds an extra dimension to the computer – often people in computer laboratories talk about 'memory space' and 'input–output space', referring to that particular region in the high-speed bussed dimensions of electrical pulses, where memory or in–out signals rule, respectively. The special input–output facilities involve another control line. We shall call this line $M/\overline{IO}$, meaning *memory or input–output.* Note the bar above $\overline{IO}$; this is consistent with the following definition of $M/\overline{IO}$:

$M/\overline{IO} = 1$ (high)	implies a memory type operation
$M/\overline{IO} = 0$ (low)	implies an input–output type operation.

For example, if the number 0101 1110 is put on to the data bus by the CPU, then it gets sent to memory if $M/\overline{IO} = 1$, or else to an output device if $M/\overline{IO} = 0$.

How can this new line be used to control one single output device, a television screen as shown in figure 2.27? The television screen is buffered as usual on to the data bus. Now, remember that the buffer must be enabled when there is valid data on the bus coming from the CPU. This is true when the write signal WR goes high. So this is the signal that must be used. Also, this is input–output, not memory, so $M/\overline{IO}$ must be low. It is easy to see that the enable pulse must be made from $M/\overline{IO}$ inverted and then ANDed with the WR signal.

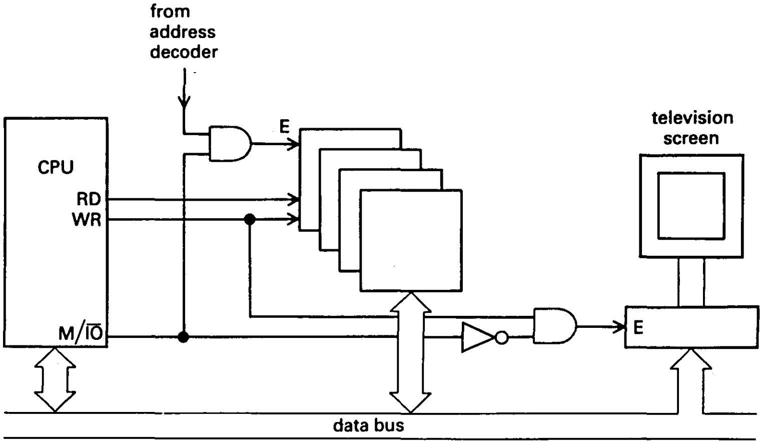


Figure 2.27
Using the memory/input–output control signal ($M/\overline{IO}$) to select output device or memory.

A glance at the timing diagram in figure 2.28 will confirm that this will work. The whole diagram refers to an output cycle, so $M/\overline{IO}$ is low, and no memory is involved. The data bits to be sent to the television screen appear on the address–data bus late on in the cycle, at the same time as WR goes high.

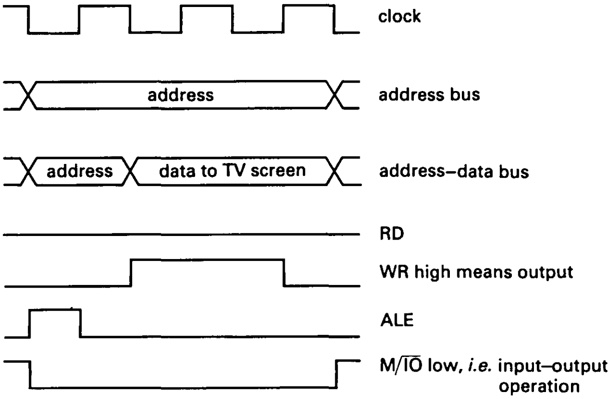


Figure 2.28
Addition of the $M/\overline{IO}$ signal to the timing diagram. Here, the $M/\overline{IO}$ signal is low, implying an input–output instruction. Also, the WR signal is high, implying a write, here an output instruction.

$M/\overline{IO}$	RD	WR	type of operation
0	0	1	output
0	1	0	input
1	0	1	memory write
1	1	0	memory read

Figure 2.29
Operation types for the allowed combinations of RD, WR, and $M/\overline{IO}$.

You can work out how input instructions are carried out using the read (RD) line to open the buffer on the input device. The table in figure 2.29 summarizes the states of RD, WR, and $M/\overline{IO}$ for all types of CPU operation, memory and in–out.

Although this method of using $M/\overline{IO}$ will give one input and one output device, rather more power than that is required. The 8086 processor can control 64 000 input/output devices! The $M/\overline{IO}$ line gives us that potential. Glance again at the timing in figure 2.28. Note that there is an address on the address bus during the entire operation. Why is this address not *decoded* and used to select an input–output device, just as it was used earlier to select a memory card? That is in fact how it works. When the CPU executes an OUT or IN instruction, it will send an address which the programmer can write into the program, thus selecting the input–output device. This select pulse is combined with $M/\overline{IO}$ and RD or WR to enable the device's buffer. The hook-up is shown in figure 2.30.

One input and one output device are shown. As before, the output device is enabled by ANDing WR and $M/\overline{IO}$ inverted, but now it is also ANDed with the device select signal coming from the 4-bit, 1-of-16 decoder. The input device is enabled by ANDing RD with $M/\overline{IO}$ inverted and the device selection pulse. Other input–output devices would use other selection signals, ANDing with RD, WR, and $M/\overline{IO}$ inverted in the same way. With full decoding of an 8-bit address bus, the system could control $2^8 = 256$ input and 256 output devices. Already our simple computer has enormous power.

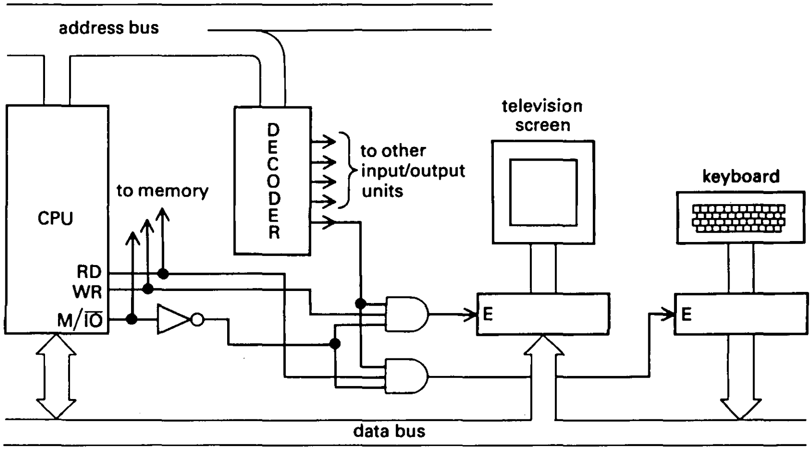


Figure 2.30
How to drive many input–output devices. Control signals are combined with a decoder output to select memory, or any input–output device from a large menu.

There are other ways of achieving input–output, notably 'serial' communication and DMA 'direct memory access'. These will be described briefly in Chapter 4.

To collect all of this chapter's work together, the fully developed microcomputer system is shown in figure 2.31, complete with 1K of memory, in–out, and decoding circuits.

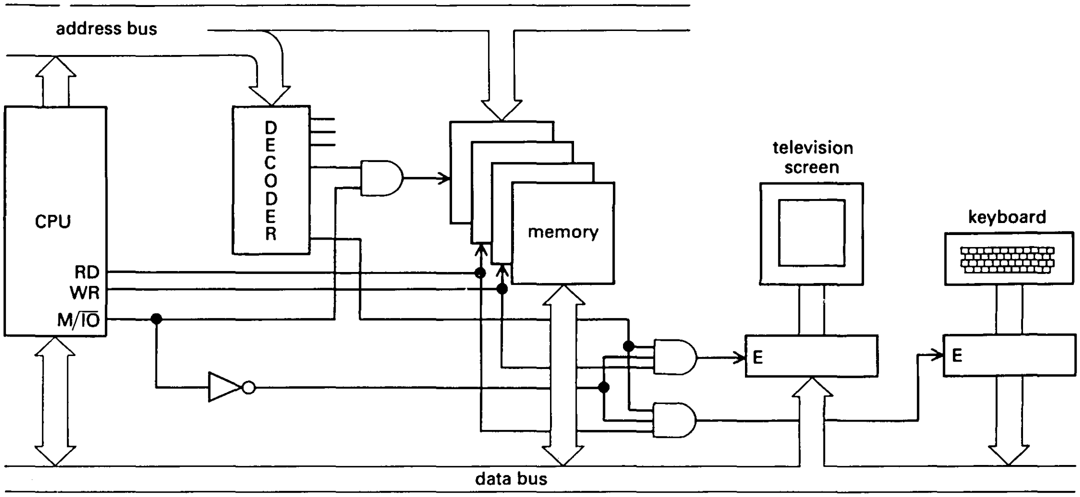


Figure 2.31
All the ideas so far assembled to produce this small computer system. Such a system has been built, and worked first time!

CHAPTER 3
RUNNING A SIMPLE PROGRAM

In Chapter 2 we saw how a CPU, memory, and input–output devices communicated with each other using bussed data and address signals, RD, WR, and ALE signals, all of which did the right thing at the right time to make the system work. Now these data movements and their controlling signals do not happen by magic; it is the job of the computer program, written by you. In this chapter we shall see how a simple program, stored in memory, is able to make a simple computer run. The computer is a hypothetical 4-bit machine called SAM (Simple Although Meaningful). SAM employs most of the ideas discussed in Chapter 2, although it is a bit simpler.

SAM'S HARDWARE

SAM has a 4-bit multiplexed (shared) address–data bus, and the usual RD, WR, and ALE signals. It is connected to memory only, there is no input–output device in use at the moment, so we can forget about the M/$\overline{\text{IO}}$ signal. The memory is also simple, just one board, so we do not have to worry about decoding. SAM is shown in figure 3.1.

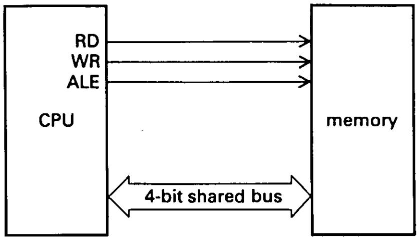


Figure 3.1
SAM.

What is inside the CPU? This is shown in figure 3.2. You can see a number of *registers*, which are simple 4-bit memories used to hold 4-bit binary numbers. These communicate with the 4-bit bus via the usual tristate buffers. The registers are: *reg* A, *reg* B, the *accumulator*, the *program counter*, and the *instruction register*. They all have very specific jobs to do; for example, the accumulator holds all results of maths-type operations. The purposes of the other registers will be revealed as you read on. There are two other important units on the CPU chip; the *ALU* or *arithmetic logic unit*, which does what it says: arithmetic and logic jobs, like adding, subtracting, ANDing, and ORing. Then there are the clock and timing, and the instruction register circuits. These circuits provide the RD, WR, ALE signals and also signals to enable the register buffers.

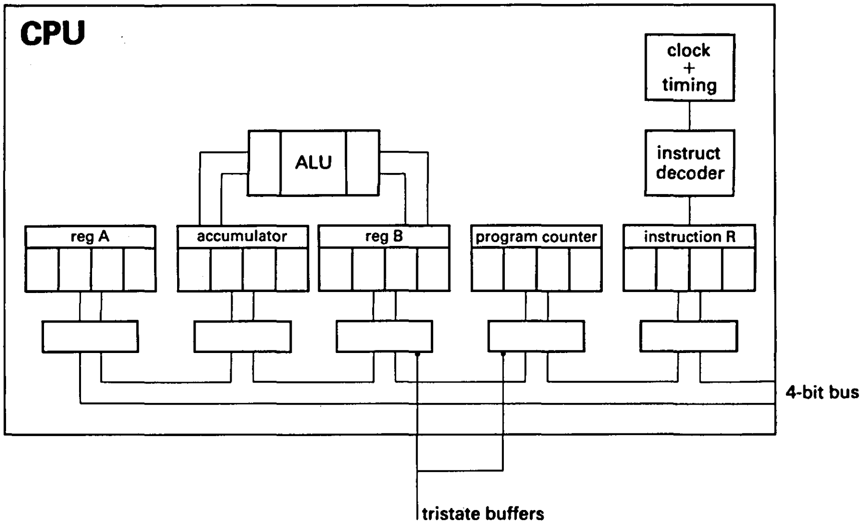


Figure 3.2
SAM's CPU laid bare.

SAM's memory board is shown in figure 3.3. Since SAM is a 4-bit machine, it has $2^4 = 16$ possible addresses. Eight of these are shown. Since the address–data bus is multiplexed, there must be an address latch, driven by the ALE signal, and also a tristate buffer between the memory data lines and the bus. These are shown in figure 3.3.

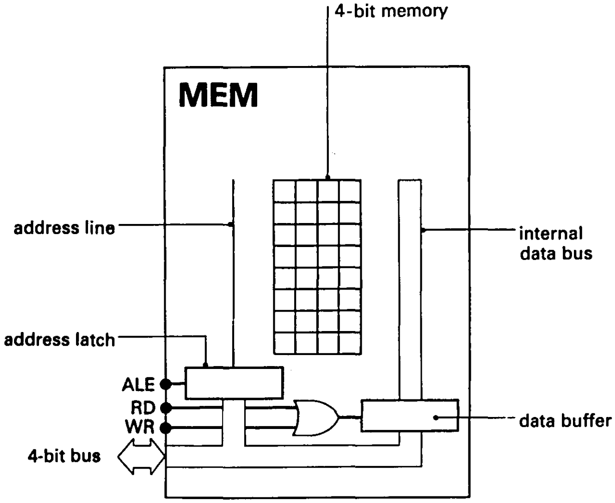


Figure 3.3
SAM's memory laid bare.

THE PROGRAM LIVES IN MEMORY

On any computer when you type in a line of BASIC like A = A + 1, the program is stored as a series of bits at particular memory locations. SAM stores everything as 4-bit numbers, for example the operation of adding is represented by 1111. Figure 3.4 shows this instruction stored somewhere in SAM's memory. A whole program, made up of more instructions plus some data, is stored in memory as a long list of 4-bit numbers.

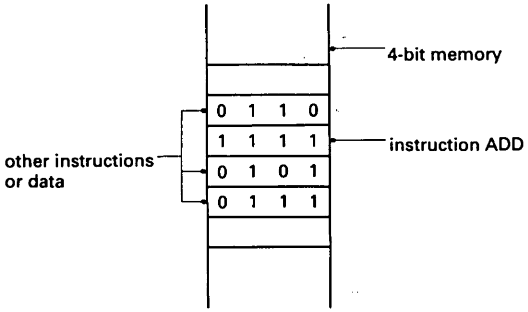


Figure 3.4
A section of memory, showing the instruction for addition in one memory location.

How does the CPU actually execute a program? First, it must fetch the instruction from the memory, by READing memory. But how does it know which memory location to read next? This is the job of the program counter register in the CPU, which holds a 4-bit number – the address of the instruction to be dealt with next. When SAM is switched on, the program counter (PC) is reset to 0000. As the program runs, so the PC is incremented, usually fairly steadily, as shown in figure 3.5.

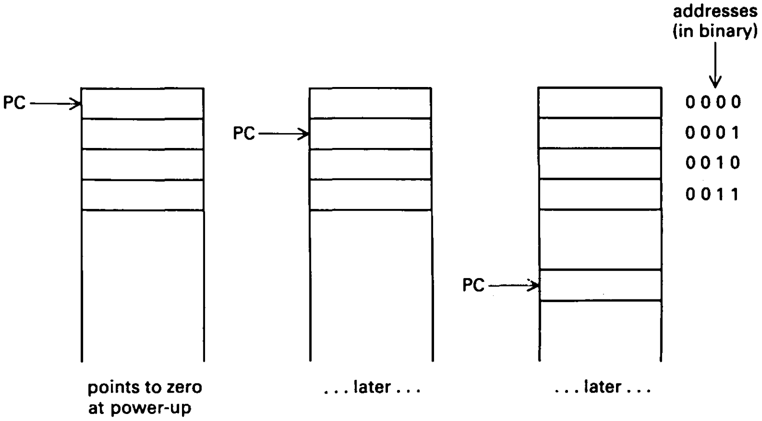


Figure 3.5
Showing how the program counter points to memory locations. Low memory addresses are near the top.

Some instructions like JUMP or CALL SUBROUTINE may send the PC bouncing around anywhere within memory. Wherever the instruction comes from, the CPU now has to interpret it, and execute it.

INTERPRETING THE INSTRUCTIONS

The instruction 1111 (addition) is fetched from memory and is loaded into the *instruction register*. Instructions always go into this register. From here, they pass into the instruction decode logic. The number 1111 is interpreted as an addition, so an enable signal is sent, for example, to the ALU to tell it to do an addition. The actual guts, the logic circuits inside the instruction decoder and inside the ALU, are too complex to describe here, but if you are keen, get hold of data sheets for the CD40181 or CD4057 circuits, made by RCA, and have a look at these.

A SELECTION FROM SAM'S INSTRUCTION SET

Here is how to write a small program for SAM to add two numbers. This program will be made up of a few simple instructions. We must use the 1111 addition instruction, but also we will need other instructions to move data in and out of memory. Begin by looking at a MOVE instruction, represented by 0010.

move immediate data to reg A	MVI A	0010
↑ what the instruction does	↑ mnemonic	↑ op-code

The 4-bit number 0010, which the instruction decoder interprets as a 'move', is called the *operation-code* or *op-code.* The abbreviation for the move, MVI A, is called the *mnemonic.* But what does this particular MOVE instruction do?

When the CPU fetches 0010 from memory, the instruction decoder tells it that it must look at the immediately following memory location. There, it will find a 4-bit number, say 1100, which it must move into CPU register A. This is illustrated in figure 3.6.

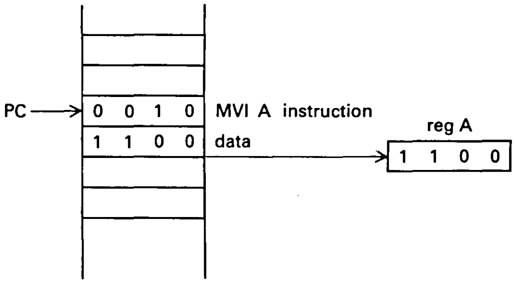


Figure 3.6
MVI A instruction. The instruction 0010 causes the next number, 1100, to be moved into register A.

The number moved into register A immediately follows the MOVE instruction in memory. That is why 0010 is a *MOVE IMMEDIATE* instruction, represented by the mnemonic MVI A. Care has to be taken in preparing for the next instruction in memory. The PC started out pointing to the MVI A instruction. If the PC were incremented by one, it would then point to the data, the number 1100. If that happened, the CPU would think 1100 were another instruction and things would go wrong. So the PC must be incremented twice, to skip over the data number. See figure 3.7.

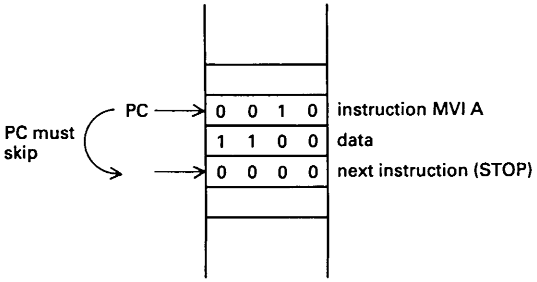


Figure 3.7
The program counter must be incremented twice in all 'immediate' instructions, to skip over the data number.

There is a second 'immediate'-type instruction, which is SAM's addition, 1111.

add immediate data to the accumulator	ADI data	1111

When the CPU fetches 1111 from memory, the instruction decoder knows it must perform an addition. It also knows the number to be added to the accumulator is contained in memory immediately after the 1111 instruction. Figure 3.8 shows the situation. Again, the PC must be incremented twice to skip over the data number, here 0001. This double-increment is true of any 'immediate'-type instruction.

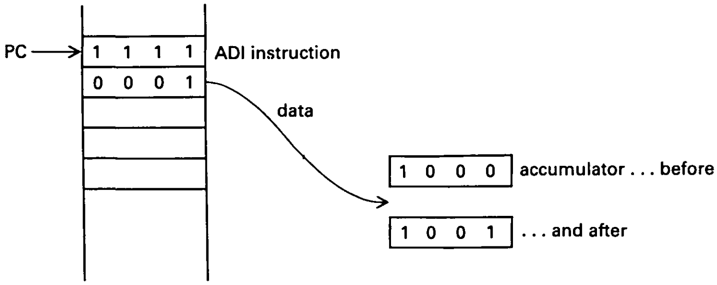


Figure 3.8
ADI instruction. The number 0001, following the ADI instruction, is added to the accumulator, and the result is stored in the accumulator.

The *accumulator* is a very useful register, since it has access to the ALU, and any data put into the accumulator can then be subjected to arithmetic and logic operations. So instructions to move data in and out of SAM's accumulator are vital. First, how to get data into the accumulator from memory.

move memory data into accumulator, via reg A	MOV M→Acc	0110

In the same way as the ADI instruction, the accumulator is expected to receive data. But, unlike the immediate instruction, the data is not in the following memory location. So where is it? The address of the data to be moved into the accumulator is contained in register A. In figure 3.9, register A has previously been loaded with the number 0111. When the CPU executes an MOV M→Acc instruction, it first looks at register A to find out which memory location contains the data. The CPU then gets the data, here 0001, from that location, and puts it into the accumulator. This way of moving data is quite useful, since data may be got from *any* memory location using just the one MOV M→Acc instruction, by changing the address stored in register A.

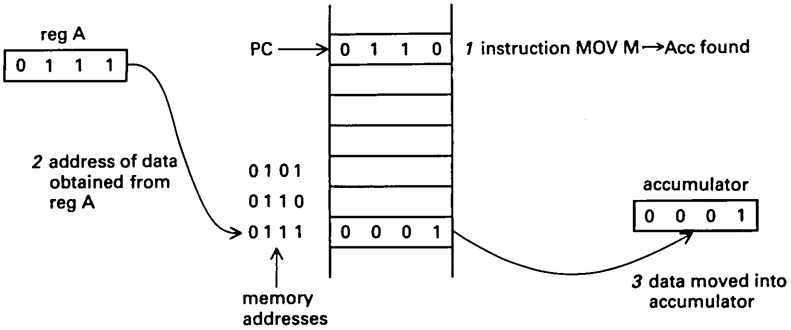

reg A
0 1 1 1
PC
0 1 1 0
1 instruction MOV M→Acc found
2 address of data obtained from reg A
0 1 0 1
0 1 1 0
0 1 1 1
0 0 0 1
memory addresses
accumulator
0 0 0 1
3 data moved into accumulator

Figure 3.9
The instruction MOV M→Acc. Note how register A must be interrogated, to get the address from where the data is to be obtained.

Perhaps a second example of this way of doing things will help you see what is going on. This is the reverse instruction to the above, moving data out of the accumulator into memory.

move accumulator data into memory, via reg A	MOV Acc→M	0111

Here the number stored in the accumulator, which could be the result of an arithmetic operation, is to be put into memory somewhere. The destination of the data is an address stored in register A. Look at figure 3.10. Here the number 0011 in the accumulator has to be moved into memory. The address where it is to go, 0101, has been previously loaded into register A. So the CPU looks first at reg A, gets the address 0101, and stores the data 0011 there, by a WRITE operation.

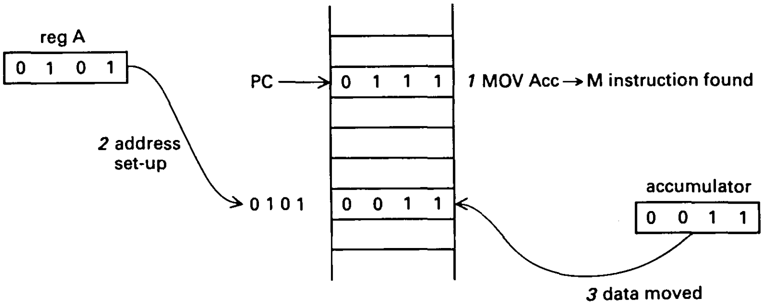


Figure 3.10
MOV Acc→M instruction. Again, note how register A must be interrogated for an address; this time the address is where the data is to be written.

That completes the instructions we shall need for our program in the next section, with the exception of one final instruction, which in fulfilling itself, needs no explanation.

stop	STOP	0000

A SIMPLE PROGRAM TO ADD TWO NUMBERS

In rough outline, a program to add a number 6_{10} in memory to another number 5_{10} somewhere else in memory, and put the result 11_{10} back into memory would look like this:

1 Get the number 6_{10} and put it into the accumulator.
2 Get the number 5_{10} and then add it, putting the result back into the accumulator.
3 Put the accumulator contents back into memory.

This program is shown nicely assembled in memory in Table 3.1. The two columns of 4-bit numbers show the instructions/data and the addresses where they are stored.

Table 3.1

EXECUTE cycle number (These numbers correspond to those in the titles of figures 3.15 to 3.25)	**Mnemonic**	**Op-code or data** (*indicates data)	**Address**	**Effect of instruction**
1	MVI A	0010	0000	Move the number 0111 into register A, ready to be used as
		0111*	0001	an address.
2	MOV M→Acc	0110	0010	Move memory contents into accumulator.
3&4	ADI	1111	0011	Add the number 0101 (5_{10}) to the accumulator, and put
		0101*	0100	result in accumulator.
5	MOV Acc→M	0111	0101	Move accumulator contents into memory.
6	STOP	0000	0110	Stop.
		0110*	0111	Data 0110 (6_{10}) to be added.

The program in detail looks like this:

Execute cycle number

1 The number 0111 is put into reg A, where it will be used as an address in the MOV M→Acc instruction in EXECUTE cycle 2, and the MOV Acc→M instruction in EXECUTE cycle 5.

2 The contents of memory whose address 0111 is in reg A is moved into the accumulator. So 0110 (6_{10}) ends up in the accumulator.

3 and 4 The number immediately following the ADI instruction, the number 0101 (5_{10}), is added to the accumulator, and the results stored in the accumulator.

5 The accumulator contents 1011 (11_{10}) are moved to memory whose address (0111) is in reg A.

6 Stop. The computer comes to a halt.

The memory location referred to by reg A which initially held the number 6_{10}, ends up receiving the result (11_{10}).

RUNNING THE PROGRAM

Now you are familiar with SAM's instructions, and have seen the assembled program, we come to the real task of this chapter: looking in detail at data flows and signal changes as the program is run. Remember from Chapter 2 that the CPU contains a clock, and three clock periods are needed to carry out either a READ or a WRITE operation. These two operations are all that is needed to run the program. Remember that the instructions live in memory, so each instruction has to be FETCHed before it can be EXECUTEd. This gives us two cycle types, FETCH and EXECUTE, which run in sequence as shown in figure 3.11.

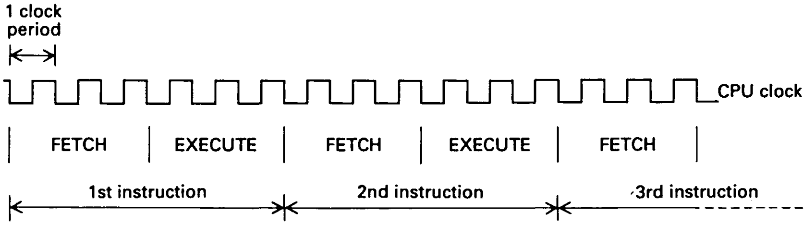


Figure 3.11
Showing how each instruction is made up of FETCH and EXECUTE cycles, which in turn are made up of three CPU clock periods.

What actually goes on during the EXECUTE cycle depends on what instruction is being executed. But the FETCH cycles are always the same for any instruction; so begin by looking at a fetch cycle.

The FETCH cycles

FETCH cycles, which get instructions from memory, are each made up of three clock periods. Preceding EXECUTE cycles, FETCH cycles always involve the same movements of data and changes in control signals, so we need to look at only one FETCH cycle. Figures 3.12 to 3.14 show what happens during each of the three clock periods making up one FETCH cycle. In each of figures 3.12 to 3.25 shading shows where data is being transferred, and where changes in registers are happening.

FETCH cycle, clock period 1
When the computer is switched on, the program counter points to memory location 0000. This is the place where the first instruction of the program lives, in this case instruction 0010, which is a MOVE IMMEDIATE instruction. So during clock period 1, the contents of the program counter are put on to the bus by a signal from the clock and timing circuits. These circuits also send an address latch enable (ALE) signal to the address latch, which latches the address 0000 from the bus. The address latch then points to address 0000.

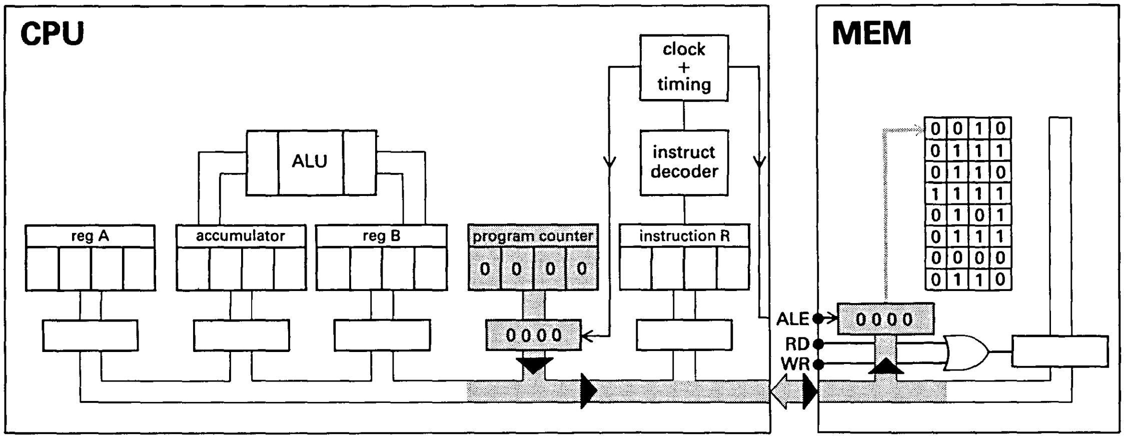
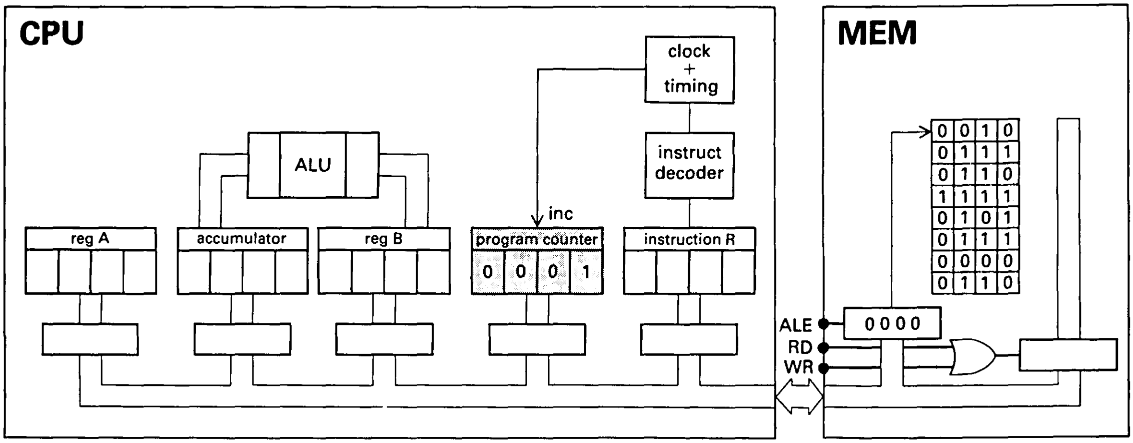


Figure 3.12

FETCH cycle, clock period 2

During the second clock period of each FETCH cycle, the program counter is incremented. This is a sort of forward-planning; it gets ready for a future data transfer, or for the fetching of the next instruction. Note how ALE remains low and how the program counter buffer is not enabled, so the new value in the program counter is not on the bus, and the address latch still points to the memory location 0000.

Figure 3.13

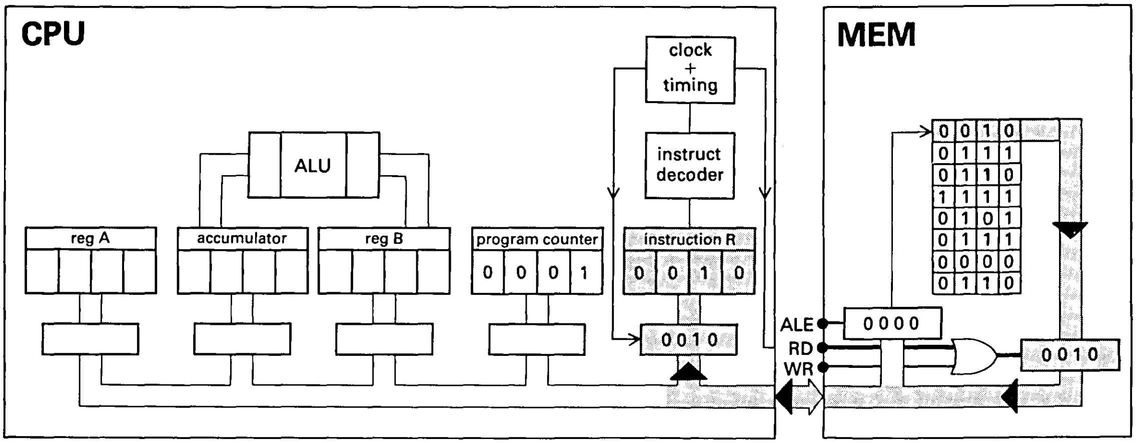


FETCH cycle, clock period 3

Here the instruction is fetched from memory. The clock and timing circuits send a READ signal to memory. Since the address latch points to memory location 0000, this READ signal puts the contents of this location, the instruction 0010, on to the bus. The clock and timing circuits also enable the instruction register buffer, allowing the instruction 0010 into the instruction register.

That is the end of the first FETCH cycle. The instruction 0010 (MVI A) is now in the instruction register waiting to be executed.

Figure 3.14



The EXECUTE cycles

During EXECUTE cycles, the instruction which has been fetched into the instruction register causes various buffers to open and close, RD and WR signals to be generated, and data to be shunted around. These signals, and their timing, are controlled by the instruction decoder working with the clock and timing circuits. We will not be concerned with how these signals are produced, but will simply observe the effect they have on buffers and registers as our program runs.

As you study the following diagrams, refer constantly to the program outline, table 3.1. (There is a copy of table 3.1 on the last page of this book.) On a first reading, take them fairly quickly, leaving a detailed study for a second pass. Also, remember that between each EXECUTE cycle there is a FETCH cycle, but this has not been drawn.

First EXECUTE cycle MVI A, clock period 1
The number immediately following is to be loaded into register A. To do this, the address of this number must be loaded into the address latch. The instruction decoder enables the program counter buffer, putting the contents of the program counter (0001) on to the bus. The address latch is enabled, via the ALE signal, and the address 0001 is loaded into the address latch. The address latch now points to the immediate memory location, containing 0111.

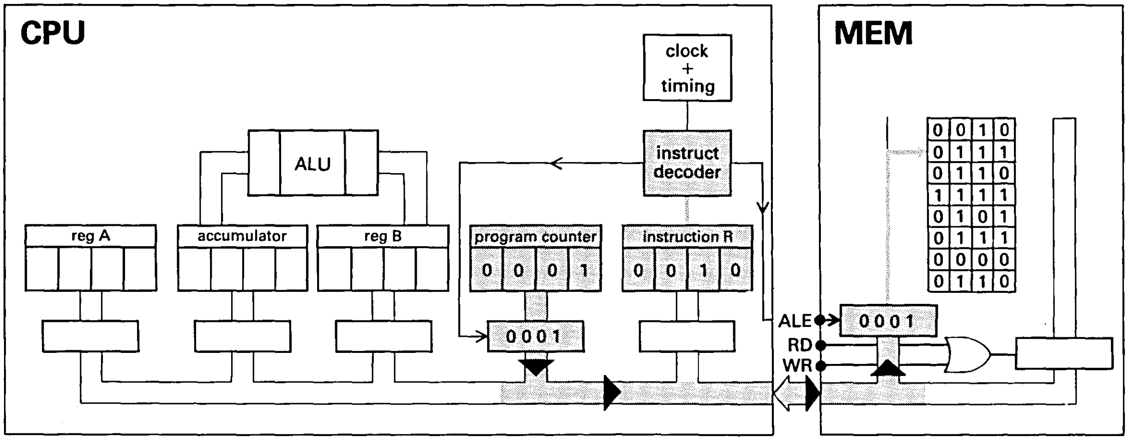
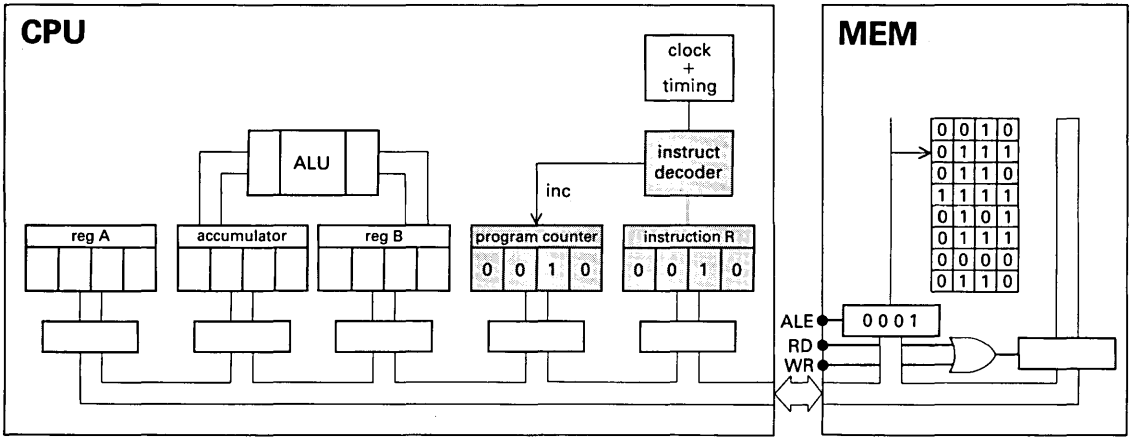


Figure 3.15

First EXECUTE cycle MVI A, clock period 2

During the second clock period of this EXECUTE cycle, the program counter is incremented. Nothing else is done. This is in preparation for the next FETCH cycle, which will fetch the instruction 0110 which lives at address 0010.

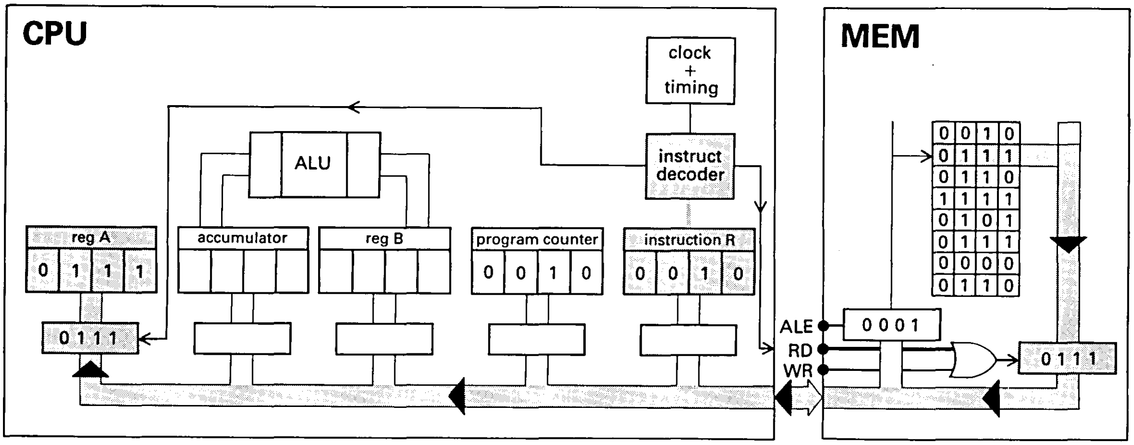
Figure 3.16



First EXECUTE cycle MVI A, clock period 3

Now the execution actually happens, the number 0111 is moved into register A. To do this, the instruction decoder sends a read (RD) signal to memory which puts the data 0111 on to the bus. Remember, the address of the memory location which is read was set up during clock period 1 of this cycle. The instruction decoder also enables register A's buffer, loading the data 0111 into register A. This completes the first EXECUTE cycle.

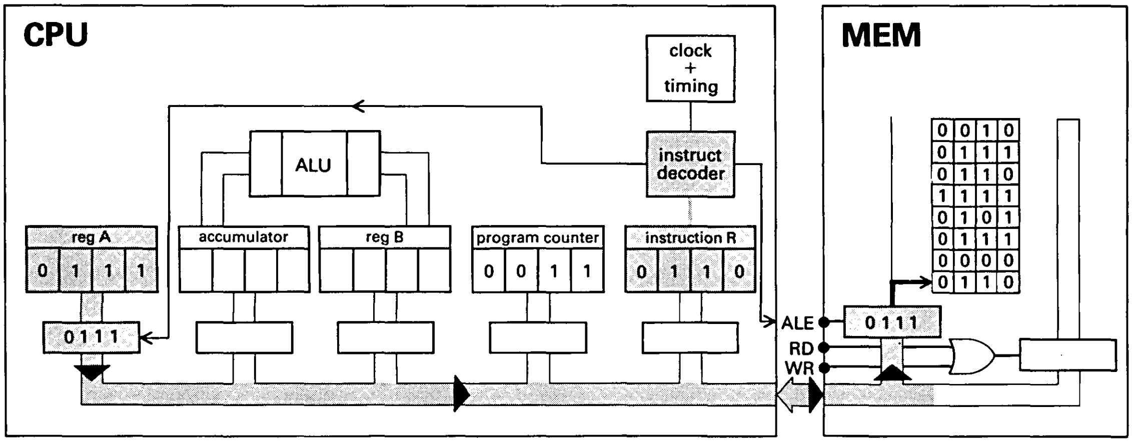
Figure 3.17



Second EXECUTE cycle MOV M→Acc, clock period 1

The instruction 'Move the contents of memory into the accumulator' has been fetched into the instruction register. This has code 0110. Now this instruction must be executed. Remember that the address from where the data must be read is held by register A. So this address must first be loaded into the address latch. To do this, the instruction decoder enables register A's buffer, thus putting the address 0111 on to the bus. The instruction decoder also sends out an ALE signal which loads this new address into the address latch. Memory location 0111 is thus addressed.

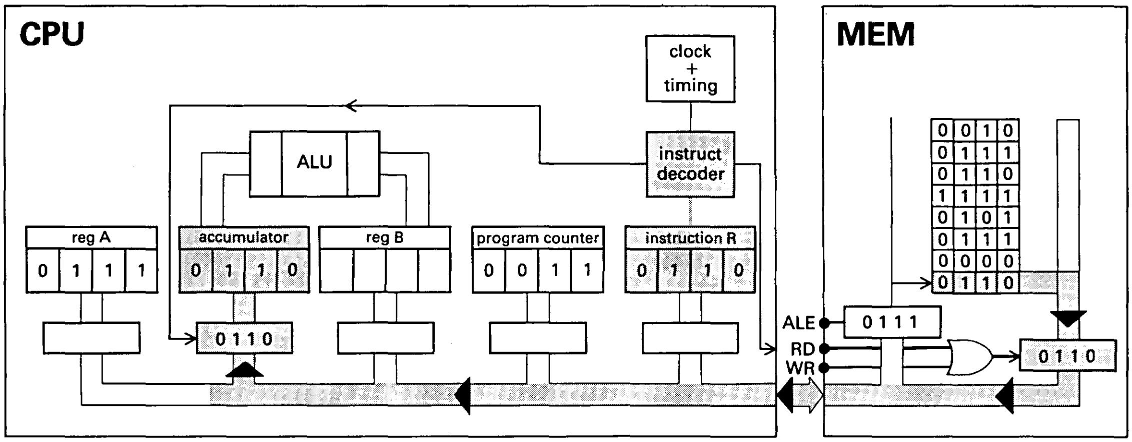
Figure 3.18



Second EXECUTE cycle MOV M→Acc, clock periods 2 and 3

Here, in two clock periods, the contents of memory location 0111, which is the number 0110 (6_{10}), is moved into the accumulator. The instruction decoder sends a read (RD) signal to memory putting the number 0110 on to the bus, and then it enables the accumulator's buffer, allowing the number 0110 to pass into the accumulator. This is the end of the second EXECUTE cycle, which has loaded 0110 into the accumulator, as directed by register A.

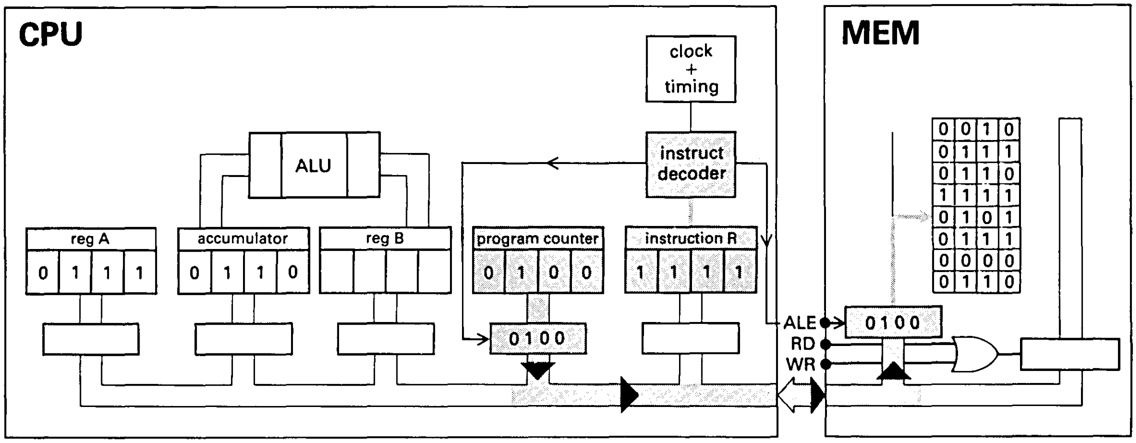
Figure 3.19



Third EXECUTE cycle ADI, clock period 1

The instruction which has now been fetched into the instruction decoder has code 1111. This is an ADD IMMEDIATE (ADI) instruction. Note how the program counter has been incremented, as usual, during a FETCH, and now holds number 0100. Now the program counter is enabled on to the bus, by the instruction decoder, which also sends out an ALE signal to enter the contents of the program counter, 0100, into the address latch. The immediate number (after the ADI instruction), 0101, which lives at address 0100, is now addressed, ready for the addition.

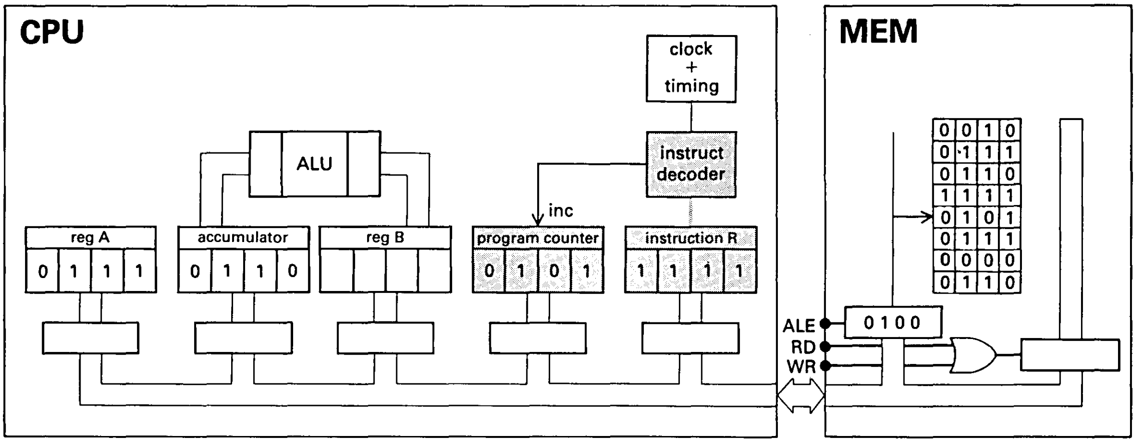
Figure 3.20



Third EXECUTE cycle ADI, clock period 2

Since this instruction is an 'immediate' type, the program counter must be incremented to skip over the data number 0101, ready for the next instruction.

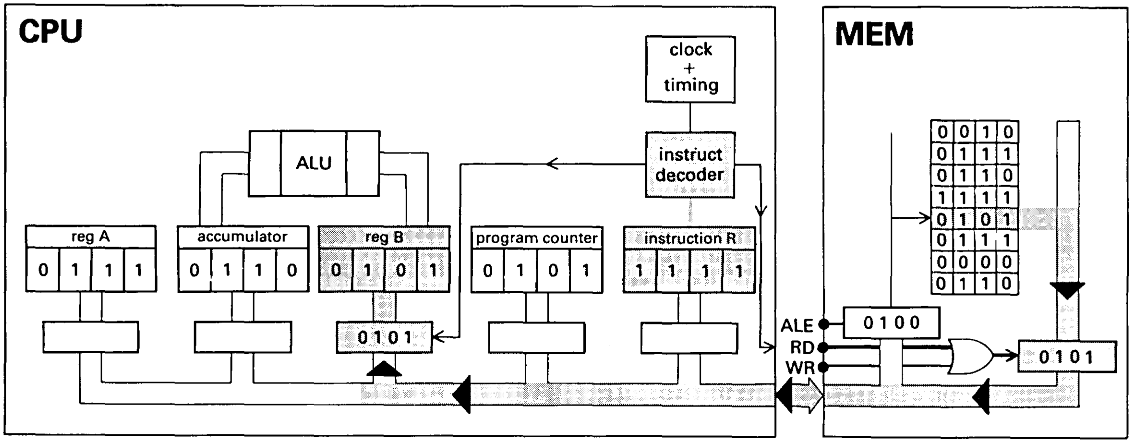
Figure 3.21



Third EXECUTE cycle ADI, clock period 3

If you look at the structure of SAM's CPU, you will see that the arithmetic logic unit (ALU), which is about to perform the addition, works on two numbers, one in the accumulator and the other in register B. That number must get into register B, and that is what happens during this clock period. The instruction decoder sends a read (RD) signal to memory, reading number 0101 on to the bus. It also enables register B's buffer, letting the number 0101 off the bus and into register B.

Figure 3.22

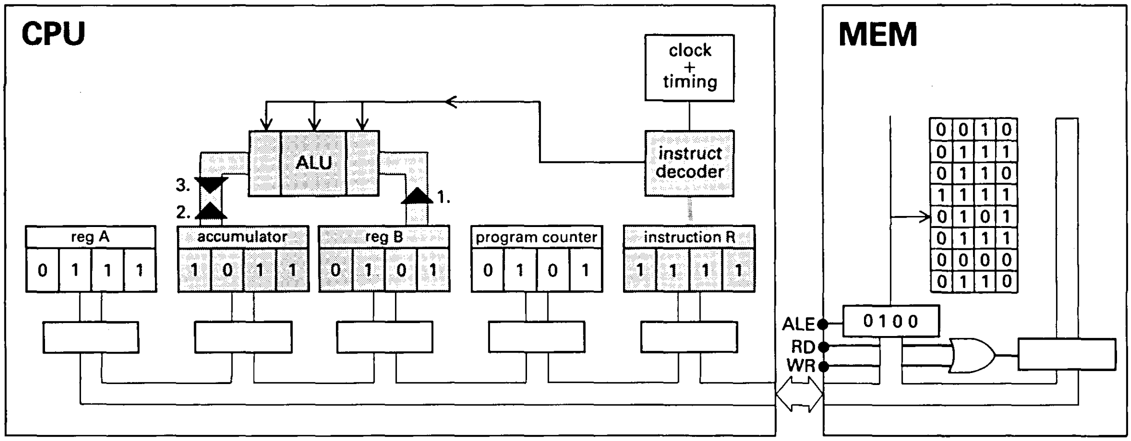


The program is now ready for the addition, but all three clock periods of this EXECUTE cycle are used up. The instruction decoder knows this (it knew it as soon as the instruction was fetched), and so it enters a new EXECUTE cycle.

Fourth EXECUTE cycle ADI, clock periods 1, 2, and 3

In this second EXECUTE cycle for the instruction ADI, the instruction decoder, together with clock and timing circuits, move the numbers from register B and the accumulator, ask the ALU to add them and put their sum into the accumulator. The accumulator now contains 0110 +0101 which is 1011. Note that in SAM there is no provision for overflows such as you would get if you instructed SAM to add 1111 +0001. Real microprocessors do have such a provision.

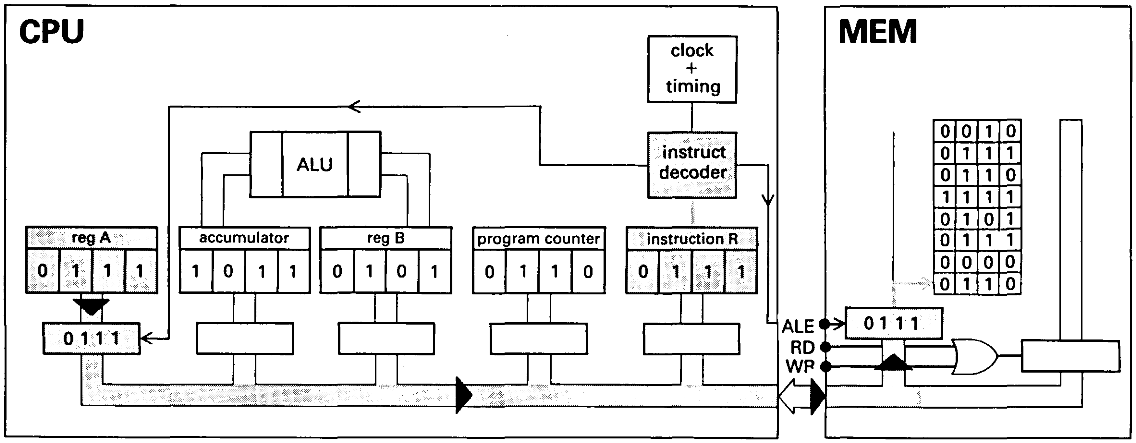
Figure 3.23



Fifth EXECUTE cycle MOV Acc→M, clock period 1

The result of the addition, now in the accumulator, must be written back into memory at address 0111. The 'Move accumulator into memory' instruction achieves this. Remember that the address of the memory location where the number is to be written is contained in register A. So, during clock period 1 of this instruction, the instruction decoder enables register A's buffer which puts the number 0111 on to the bus. The instruction decoder also sends an ALE signal to memory which latches 0111 into the address latch. Memory location 0111 is now being addressed. This is the location that will receive the result of the addition.

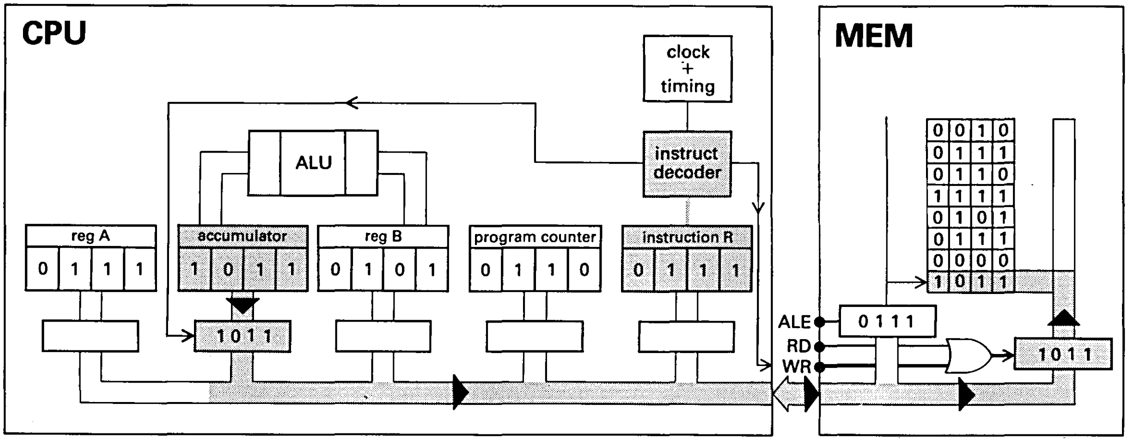
Figure 3.24



Fifth EXECUTE cycle MOV Acc→M, clock periods 2 and 3

During these last two clock periods of this EXECUTE cycle, the data in the accumulator is moved into memory location 0111. The instruction decoder enables the accumulator's buffer, so putting the accumulator's contents, 1011, on to the bus. Also the instruction decoder sends a write (WR) pulse to memory which writes the data 1011 into memory location 0111, this address being stored in the address latch. Note that the number 1011 is written over the previous occupant of this address, the number 0110, without first having to clear out the memory location.

Figure 3.25



That is the end of the program, except for the STOP cycles. This instruction, which has code 0000 lives at memory address 0110. A glance at the program counter in figure 3.25 should convince you that the program counter holds this address.

As described in Chapter 2, computer engineers work extensively with timing diagrams, showing data transfers and control signal changes all on one diagram. To draw a complete timing diagram for the program we have just run through would take a very long piece of paper, but figure 3.26 shows the first three machine cycles for this program. The FETCH and EXECUTE cycles for the first instruction MVI A are drawn, and then the FETCH cycle for the second instruction MOV M→Acc is drawn. This diagram corresponds to figures 3.12 to 3.17, plus a second FETCH cycle.

You see how the bus first transfers an address to memory, the contents of the program counter, which is 0000. Note how the ALE signal goes high to latch this number into the address latch. Then, during the second period the bus is floated – there is nothing on it. This is when the program counter is being incremented. In the third clock period, there is data on the bus, and since the read (RD) line is high, the CPU is reading this data from memory. This data is of course the instruction which is being fetched, 0010, which is an MVI A instruction.

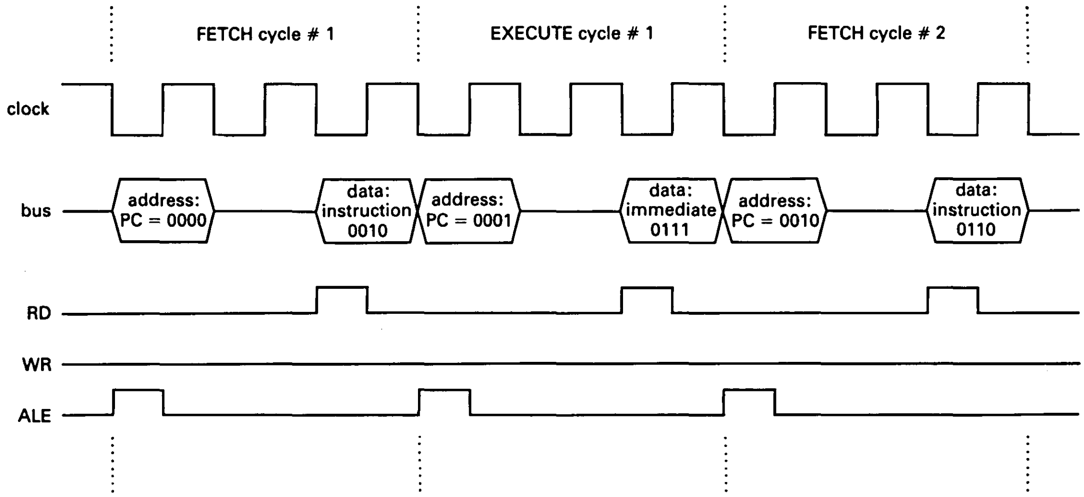


Figure 3.26
The state of the bus and control signals during the first three machine cycles of our program..Note how RD and ALE signals form a regular pattern.

There then follows an EXECUTE cycle. The CPU puts an address on the bus, latched into the address latch by ALE going high (this corresponds to figure 3.15). During the second clock period, the bus is again floated while the program counter is incremented (figure 3.16). In the third clock period of the EXECUTE cycle, data are again present on the bus, and since RD is high, this is being read from memory by the CPU. This corresponds to figure 3.17, where this data, the number 0111, is being read into register A.

The second FETCH cycle runs just like the first. Note the new value of the address on the bus (0010) since the program counter was incremented during the EXECUTE cycle.

You should be able to see already a nice, organized pattern of data or address transfer and control signal variation. If you are feeling brave, why not complete a timing diagram for the entire program? If you do decide to have a go at completing the timing diagram, the following hints will be useful:

a Keep the ALE pulses appearing at regular intervals. Also keep RD (or WR) pulses appearing at regular intervals.

b During EXECUTE cycles 2 and 5, there is no increment in the program counter. Program counter increments are performed during clock period 2 of both FETCH and EXECUTE cycles, so EXECUTE cycles 2 and 5 must look like any other EXECUTE cycle, except that clock period 2 is a 'dummy', and the actual executing takes place, as usual, during the third clock period.

I hope you have got an impression of a lot of organized timings and efficiency behind this simple program. An integrated-circuit microprocessor system is certainly integrated in more ways than one! Just how real is SAM? Quite real: it is possible to build a machine that will work just as described. The CPU 'architecture', how many registers, and how they are connected on the CPU chip, is probably underdeveloped in SAM; a typical 8-bit CPU chip will have seven or eight 16-bit registers, and a 16-bit CPU has to have a very sophisticated architecture. In their instruction sets, these processors may have tens of instructions dedicated to moving data between registers. And of course we have not talked about input–output. Different manufacturers have various philosophies of CPU architecture, but the notions we have been discussing, such as RD, WR, ALE, and the work in this chapter are common to most.

The CPU has executed a STOP

Working through these diagrams of data shunting to and fro along a bus, you may have noticed a similarity to a certain type of puzzle. Have a go at one: the two black cars (figure 3.27) want to go to the right and the three white cars want to go left. There is a passing place big enough for just one car. What should the cars do? Yes, cars do have reverse gears don't they?

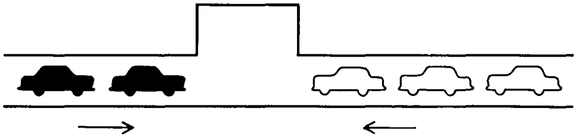


Figure 3.27
Five cars on a data bus. Work out how they must move on the bus and into the 'parking' register, so they may pass in the directions shown.

CHAPTER 4

MEMORIES, INTERFACES, AND APPLICATIONS

This chapter contains some short notes about some specific parts of a microcomputer system, such as memory technologies, and also some applications information suitable for a small physics or engineering laboratory.

MEMORY TYPES

Read only memory (ROM)

Read only memory (ROM) is the simplest type to understand. ROM cells can be read only by the application of RD signals; the user of the computer cannot store data or a program in them. They are manufactured containing the program – each cell is set to 1 or 0 by the manufacturer according to the customers' wishes. It is not a very cheap system.

Figure 4.1 shows how you could make ROM using diodes, wires, and a soldering iron. There are eight memory locations, each of four bits, shown.

The locations are shown selected by a simple switch, but you could design some address decoding logic yourself, using eight 3-input AND gates and three inverters.

You do not find many ROMs being used in small machines these days, because of cost. A much better type of read only memory is the *programmable read only memory* (PROM). Here the user 'burns in' his own program into a PROM chip, which arrives with all cells set to logic 1. In the cells where the user wants a 0, he must burn open the links to those cells using a 'PROM-programmer'. Of course, once the program is loaded, it cannot be changed.

The next development came with the *eraseable programmable read only memory* (EPROM). In order to appreciate the construction of this type of memory, one needs a detailed understanding of solid-state physics. But with EPROM, the user can insert a program not by burning open links, but as stored charges. To change the program, these stored charges, the 1s and 0s, may be removed by exposing the memory to ultra-violet light, making the memory components temporarily conductive, allowing the charges to escape. Depending on the strength of the light, the erasing time may be up to 30 minutes.

The latest development in read only memory has been the *electrically eraseable programmable read only memory* (EEPROM or E^2PROM) where the erasing is done electrically, and takes 20 ms per chip (2Kbytes). This relies on a quantum-mechanical process called 'tunnelling' to make the chip temporarily conducting, to allow the stored charges to run out.

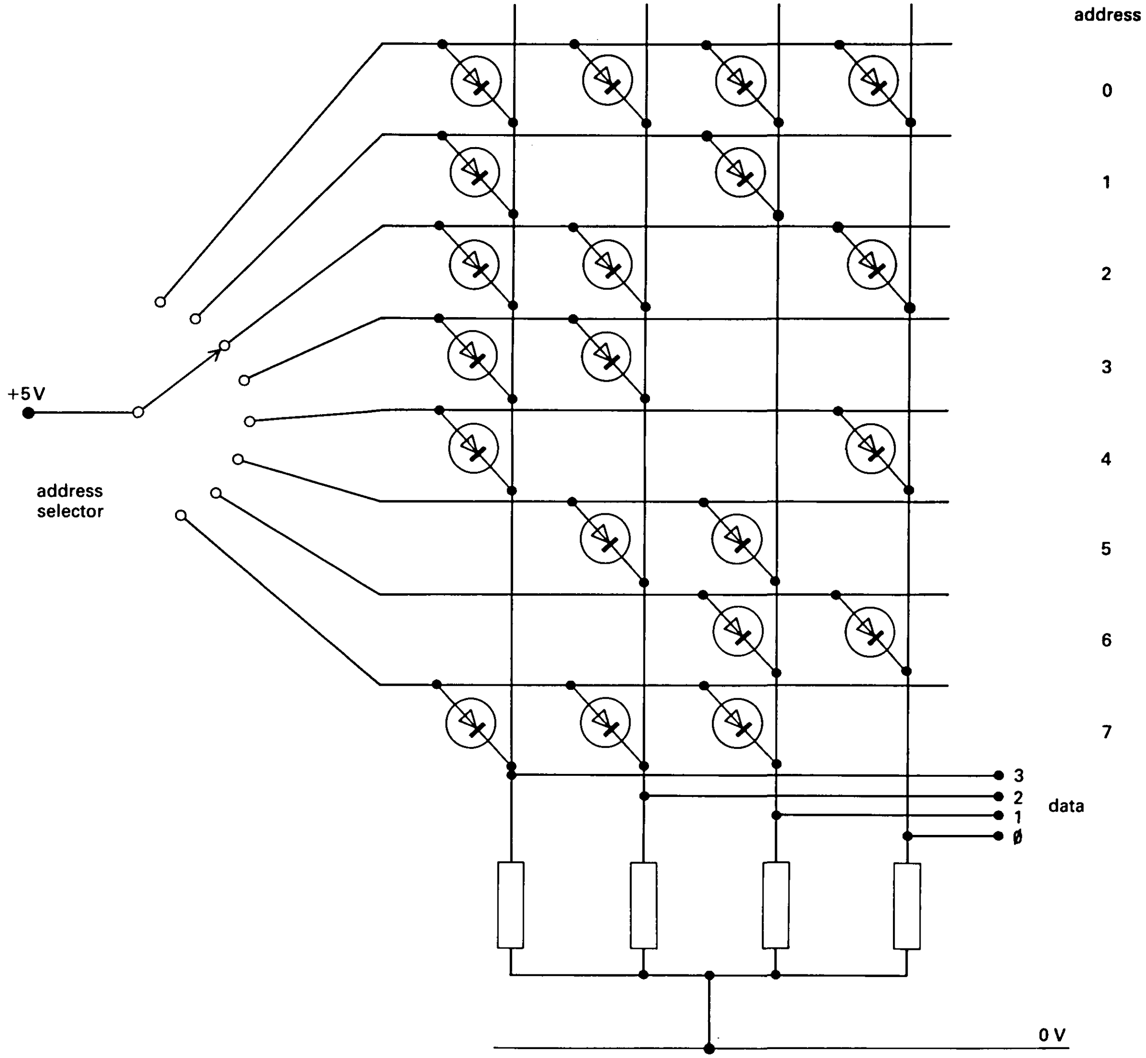


Figure 4.1
Hard-wiring a ROM using diodes. Address 2 is shown selected, the data contained here being 1101.

Whether you use PROM, EPROM, or E^{2}PROM, this memory type is 'non-volatile', meaning that when you turn the power off, the chip will hold the program it contains. This is in contrast to the memory type RAM to be described next. Non-volatile memory is used for storing the system programs. For example, the BASIC language in a home computer is stored in PROM or EPROM. The instructions in a small data-collecting microcomputer in an aircraft are also stored in PROM or EPROM.

RAM – data storage

When your program needs memory to store numbers, the results of computations, the patterns to be displayed on a television screen, or the measurements made from a microprocessor-controlled instrument, then *random access memory* (RAM) is needed. Remember that, as explained in Chapter 2, RAM is made up of a number of cells. Each individual cell is selected by row select and column select signals, generated from the address sent to the memory chip. This is shown again in figure 4.2.

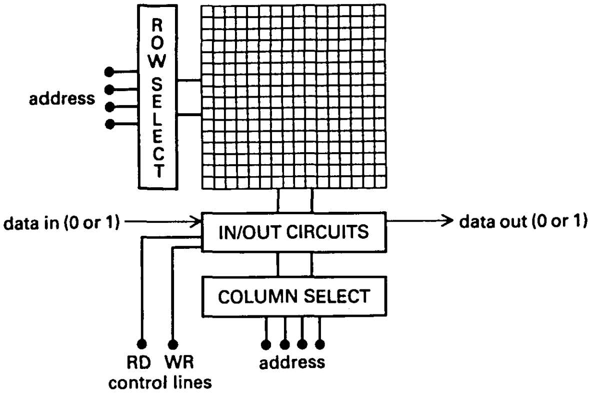


Figure 4.2
RAM cells shown with their support circuitry. Selection of each cell is determined by the address sent to the row and column select logic. Input and output are done in conjunction with the column select logic.

Here, we are interested in what actually makes up an individual memory cell, able to store a binary 0 or 1. The simplest type of RAM to use is called *static RAM*, where each cell is a simple flip-flop. This is shown in figure 4.3.

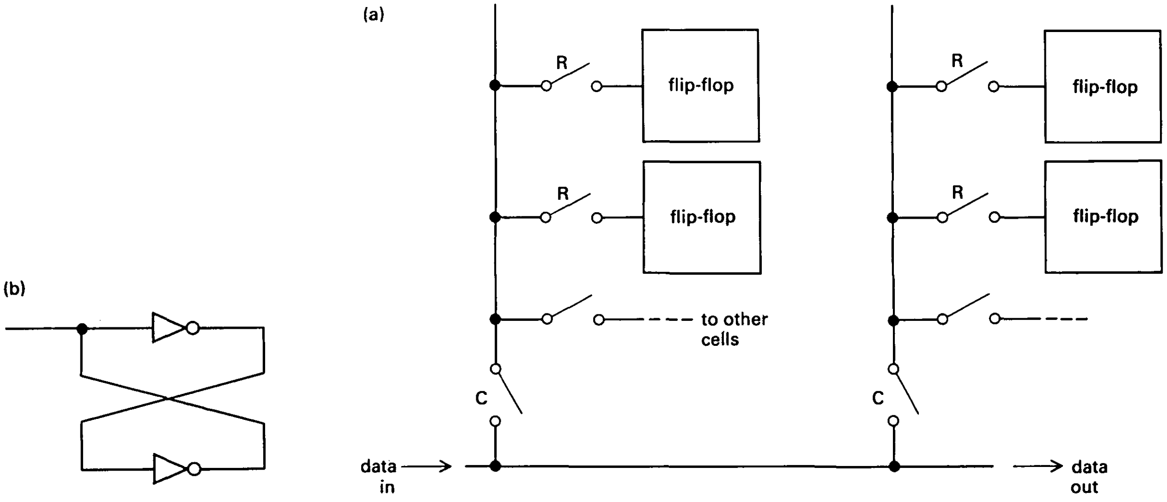


Figure 4.3
Static RAM. (a) The switches C and R used to select the column and row of a cell are in reality transistors. (b) The detail shows a single cell flip-flop shown as two cross-coupled inverters.

The flip-flop is represented here as two cross-coupled inverters – figure 4.3(b). In commercial chips, like the 2147, four transistors would do this job. Note that there is a common input and output connection to the flip-flop; check for yourself that this works by putting a logic level 1 at this connection, and work out the logic states at the inputs and outputs of both inverters. Repeat for input logical 0. Figure 4.3(a) shows how four of these cells are connected with switches. In reality, these switches are also transistors on an integrated chip and are controlled by the row and column select logic. Data is written and read via the line at the bottom of the diagram.

You will realize that in a flip-flop one half is on while the other is off. So two transistors are always at work. This type of memory needs a lot of current, but is fast. A 2147 chip needs about 55 ns (nanoseconds) to read or write a cell. More common RAM needs about 200 ns.

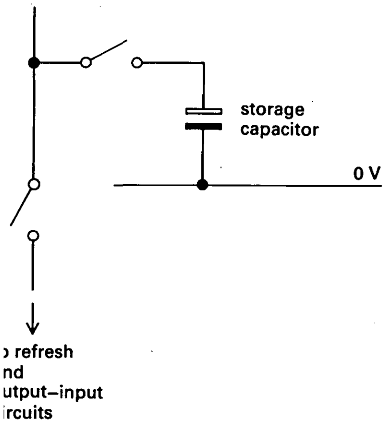


Figure 4.4
A dynamic RAM cell is basically a capacitor whose charging is controlled by a single transistor switch.

The second type of RAM memory cell is the *dynamic RAM* shown in figure 4.4. Here the bit stored, 0 or 1, is stored as no charge, or charge on a capacitor. A single transistor, shown on the diagram as a switch, is needed to control the charging of this capacitor. At once you can see that these dynamic cells are much simpler and smaller than static cells. Since they store information by storing charge, they do not need large operating currents. They score over static RAM, but there is a small problem: the charge on the capacitor leaks away, so special control circuitry has to be installed, to continually top-up the capacitors, or 'refresh' them. This must happen every 20 ms or so. Refresh circuitry is complex to build, and dynamic RAM controllers, chips dedicated to topping up the dynamic RAM capacitors, are not cheap. The expense of this extra circuitry is only recovered when large memory systems of hundreds of K are being built.

Bubble memory – mega storage

Bubble memory units with capacities of up to 4 megabits are the subject of research today. They are not semiconductor devices, but employ magnetic materials in their technology. This is not as hard to understand as semiconductor technology, so here is an outline of how a bubble memory works. It illustrates how physics has been applied in a high-engineering situation.

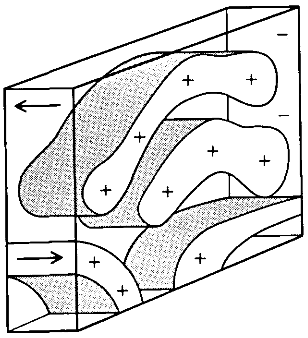


Figure 4.5
Ferrite slice with weak perpendicular magnetic field showing domains.

You are probably familiar with the idea of *magnetic domains* found in all sorts of magnetic materials. When unmagnetized, the domains in the material are randomly arranged in three dimensions, but as the material is slowly magnetized, the domain walls move so that most of the domains align to a common direction. When the material used is a thin film (0.01 mm thick) of ferrite (a magnetic oxide of iron, with other metals), then the domains become two-dimensional, as shown in figure 4.5.

When the slice is immersed in a perpendicular magnetic field, the domains oriented oppositely to the field shrink in size. As the 'bias field' is increased, the few remaining domains become small cylinders, called 'bubbles', as shown in figure 4.6.

These bubbles are a few micrometres in size. If, in addition to the bias field, there is a small field which is weak in one place and strong in

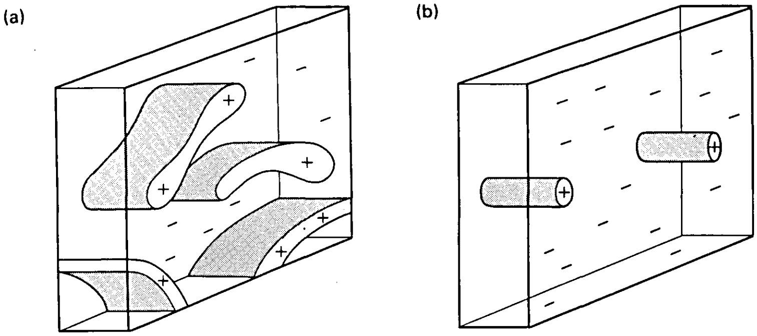


Figure 4.6
(a) Same as in figure 4.5, but with bias field increased. Domains aligned opposite to the bias field direction begin to shrink.
(b) Bias field increased enough to shrink oppositely aligned domains into small magnetic bubbles.

another, the bubbles will move to the place of greater field strength, in the same way as iron filings are attracted to the end of a bar magnet.

In a bubble memory, a bubble or absence of a bubble represent a logical 1 or a 0 respectively, and these bits are made to hop around in a cyclic fashion from one memory cell to another. Since the bits are magnetic, the memory cells must be made of magnetic material. Figure 4.7 shows two cells out of a long line of cells. The progress of one bubble from the first cell to the second is also shown.

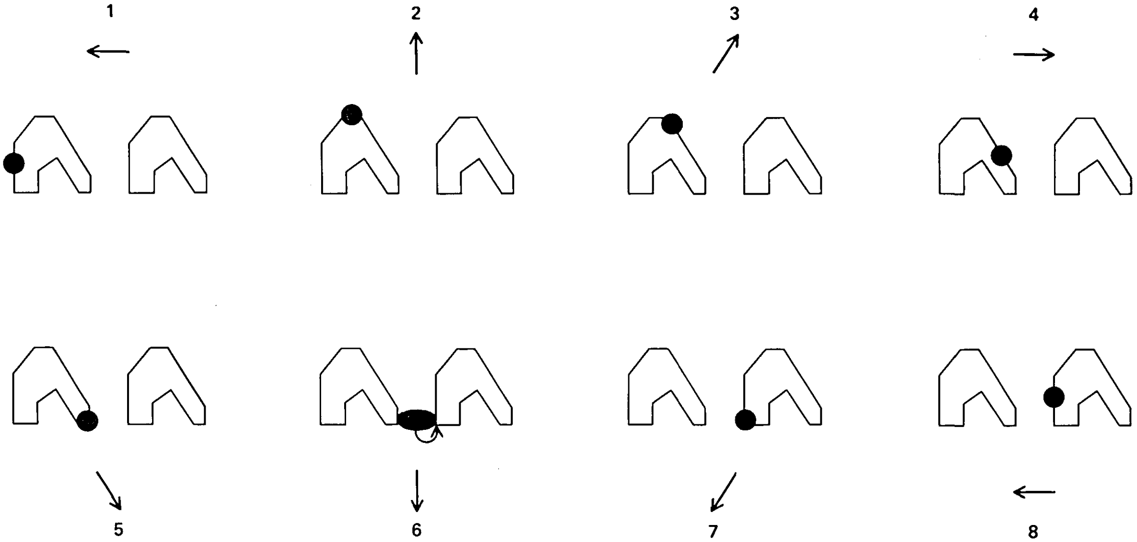


Figure 4.7
The history of a bubble and two memory cells, or 'chevrons'. As the field is slowly rotated, the bubble follows, then jumps across the cell gap. The arrows show directions of the magnetic field as it rotates.

This movement is accomplished by a rotating magnetic field which drags the bubble around the chevrons and makes it jump across at the appropriate time.

This rotating field is obtained by controlled currents passing through two coils, aligned at right angles to each other, surrounding the bubble-slice. The bias field is obtained by two permanent magnets which maintain the bubbles in existence; so this is non-volatile memory. The structure of the device is sketched in figure 4.8.

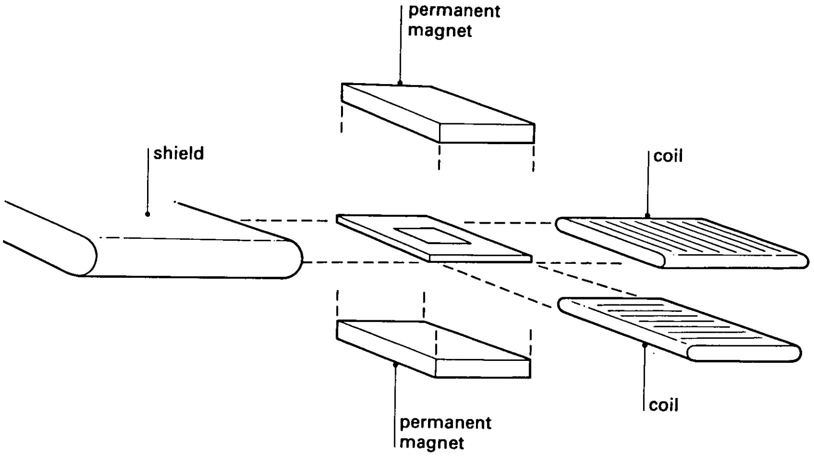


Figure 4.8
Structure of a megabit bubble memory device, showing permanent magnets producing the bias field, orthogonal coils producing the rotating field. ('Exploded view'.)

To understand how this system of moving bubbles can be used as a computer memory, we must look at how the loops of rotating bubbles are arranged. This is shown in figure 4.9.

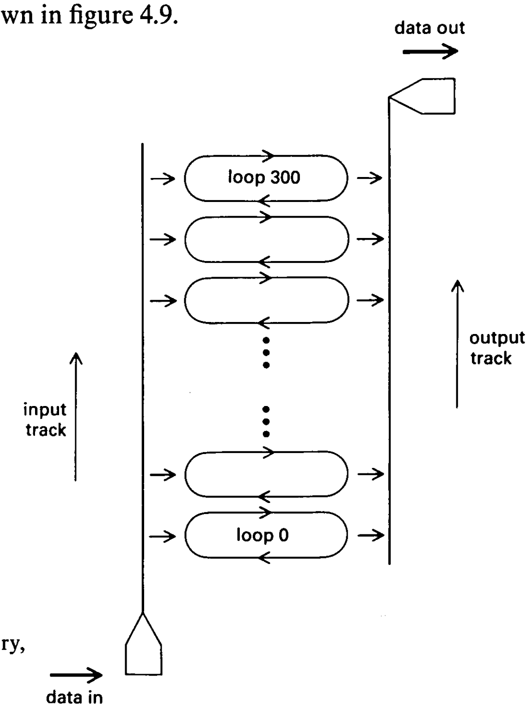


Figure 4.9
Architecture of a bubble memory, showing bubble loops and input and output tracks.

A typical bubble unit contains 300 loops, each with 4096 cells, around which bubbles circulate. Two tracks communicate with the loops: the input track which carries bubbles from a generator – a loop of conductor carrying the input current pulse which locally reverses the bias field and creates a bubble – and the output track. The latter leads to a bubble detector, a magnetoresistive device. When a magnetic bubble passes by this device the change in magnetic field strength causes the resistance of the device to change, which in turn changes a current passing through the device. To complete the memory, a bit of control is needed to synchronize bubble production and loading, or unloading and detecting. Then, with the magnetic field rotating at 50 kHz, data may be got from the memory in about 0.1 ms. So bubble memory is not particularly slow, it can store vast amounts of data, and, unlike disk storage, has no moving mechanical parts to fail. It is non-volatile and so saves its contents when the computer is switched off or if power fails.

APPLICATION NOTE – SEVEN-SEGMENT DISPLAY

How does a calculator-type display work? There may be up to ten digits used to display a large number. How can such a display be hooked up to the computer? Begin with a single digit. A typical display is made of seven light-emitting diodes arranged in a pattern of a figure '8'. This is called a *seven-segment display.* It is normally driven by a special chip that takes in a 4-bit number and produces numbers 0 to 9 and also a decimal point on the display. The arrangement is shown in figure 4.10.

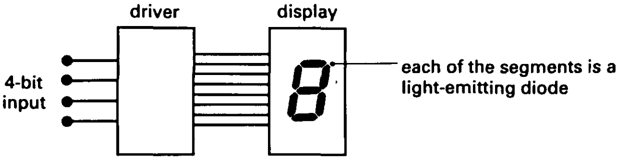


Figure 4.10
The seven-segment LED display shown connected to a driver–decoder chip which is fed by a 4-bit number 0000 (0) to 1001 (9_{10}).

We obviously have to connect this to an output buffer of our system to supply a 4-bit number for the display, and must then decide which CPU signal to use to enable the buffer. The best choice is the combination of M/$\overline{\text{IO}}$, WR, and in–out decode, and is shown in figure 4.11.

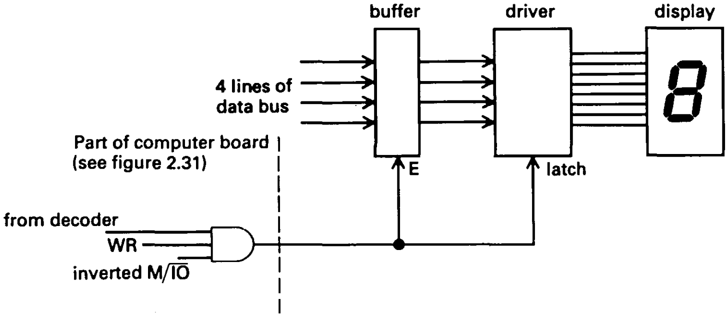

buffer
driver
display
4 lines of data bus
Part of computer board (see figure 2.31)
E
latch
from decoder
WR
inverted M/$\overline{\text{IO}}$

Figure 4.11
Single digit display, arranged as output port of our simple computer.

Now this signal goes high only when the CPU executes an OUTput instruction to this device, and only then will the data be allowed through the buffer. This signal is also connected to the display driver 'latch' terminal. When the signal is high, the data is allowed into the display, but when it goes low, after the OUTput instruction is finished, the data is latched into the display and the number will be displayed. If this were not done, then you would never see any numbers, since the output instruction takes, say, a millisecond to be executed. The latch holds the data firm until the next output instruction is performed, when the displayed number must be updated.

Extending this to 4, 8, or 10 digits is easy. You could use a display-driver chip plus output buffer for each digit, which would then appear to the CPU as separate output ports. Outputting a number to port 1 would set up that number in the first digit, outputting to the second port would set up the second digit, and so on. It would work, but uses a lot of chips. There is a simpler way. To understand how this works, we must take a look inside the display (figure 4.12).

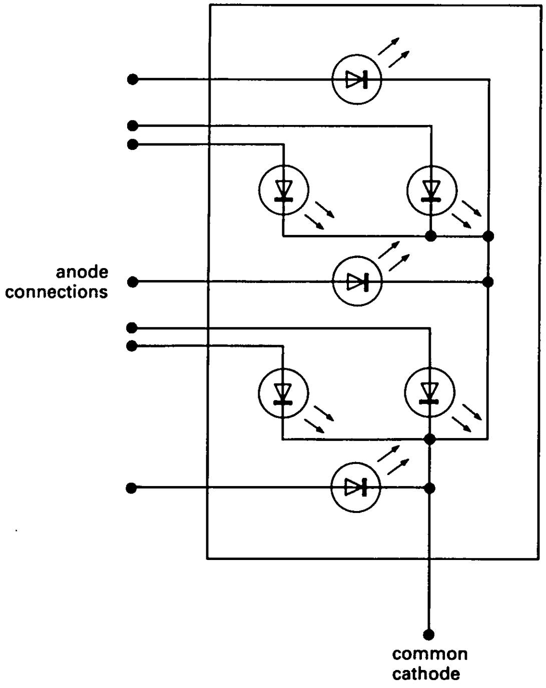


Figure 4.12
Details of a seven-segment display unit, showing how each segment, a light-emitting diode, is selected by the anodes. All cathodes are connected together.

Each segment is a light-emitting diode, and all the cathodes of the diodes are connected together. In the use described above, the driver chip would supply current (10–20 mA) to each anode, and the common cathode would be connected to 0 V to complete the circuit. Now look at the circuit in figure 4.13. Two display chips are shown being driven by one driver circuit, each output of this circuit now being connected to two anodes, one from each display.

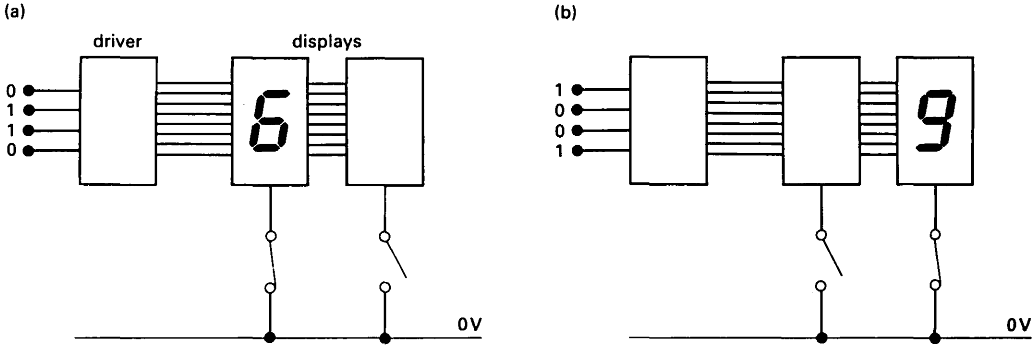


Figure 4.13
Driving two displays with just one driver chip. Switches in the cathode lines of each display turn that display on when its own 4-bit number is being input. (a) The number 0110 (6_{10}) is sent to the lefthand display by closing its cathode switch. (b) The number 1001 (9_{10}) is sent to the righthand display by closing its cathode switch.

Note the switches connected to the cathodes. For the dual display to show the number 69, first the cathode switch on the first digit is closed and 0110 (6_{10}) put into the driver–figure 4.13(a). This causes the number 6 to be displayed on the first digit. Then this switch is opened and the switch on the second digit closed–figure 4.13(b). Sending out 1001 will cause the displayed number 9 to shine on the second digit. This process is repeated a few thousand times per second. The actual switches are contained inside a special IC package. Figure 4.14 shows the final design.

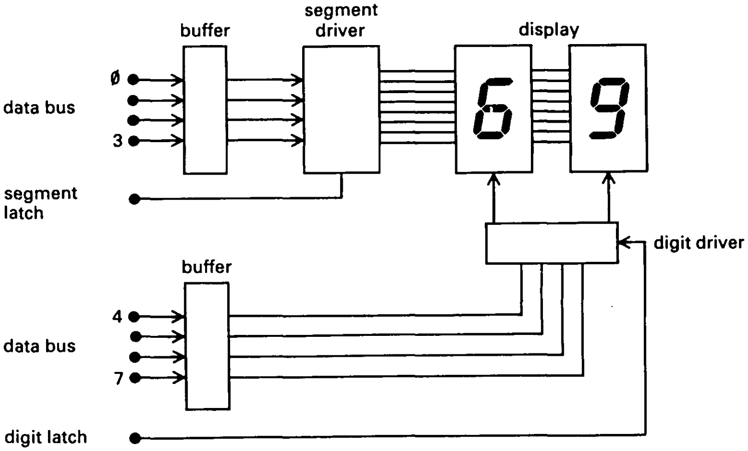


Figure 4.14
How to connect two or more displays to our microcomputer. One output port of 4-bits wide selects the segments which shine, and the second 4-bit port determines which digit these shine on.

Two output ports are needed, one 4-bit number passing to the digit-driver circuits, and the other passing into the decoder which determines which digit is shining. Note how the display driver latch is still used. This holds the digit stable while a different 4-bit number is being output to the digit select. A very short program has to be written to scan across all of the digits in the display, making sure each gets its correct number. But that is cheaper than the previous suggestion.

APPLICATION NOTE – KEYBOARD SCANNING AND INTERFACE

A keyboard may be a typewriter-type board just like the one on a personal computer, it could be a music synthesizer's 4- or 5-octave keyboard, or it could be a small 16-key pad. To take this last example, it will have 16 switches and need the computer to determine the keys pressed. Each key is given a number, here 0 to 15, and when a key is pressed, say 3, the computer has to save the number 3 in one of its registers or in memory.

The arrangement is shown in figure 4.15. The computer sends a 4-bit number (0 to 15) via an output buffer to a decoder, just like the ones discussed in Chapter 2.

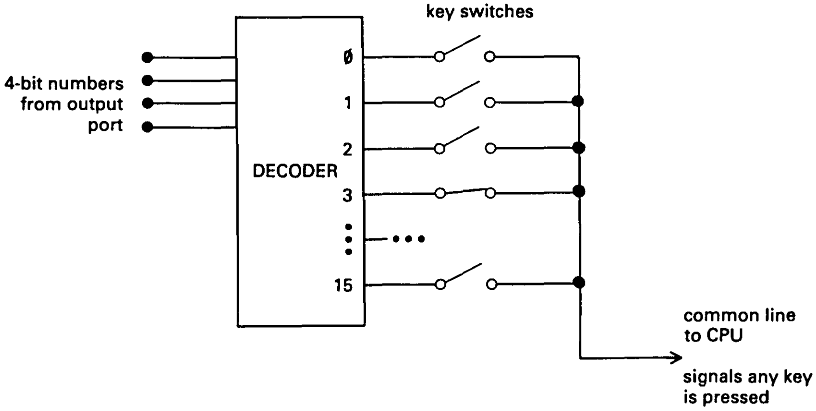


Figure 4.15
A decoder chip is used to scan a keyboard made of 16 key switches.

Remember, a decoder receives a 4-bit number, and then makes one of its 16 output lines go high. If the processor has output 0011 (3_{10}), then the decoder makes its output 3 high, which is connected to key switch 3. Notice that all the key switch outputs are connected together. If key 3 is pressed then the high gets through to this common line. If 4 is pressed the high does not get through. A high on the common line means number 3. The common line is connected directly to the microprocessor, so that the microprocessor knows that a key has been pressed, and stops. Now the microprocessor has been programmed to output not just 3, but all numbers 0 to 15 in order, over and over. So if key 3 is pressed continually then the common line will remain low until the processor has got round to outputting 0011. Then it will go high, and the microprocessor will be stopped. Its last output, 0011, will be in some register which can be got at and so used in another part of the program to implement whatever pressing key 3 is meant to do.

Where exactly is this common line connected to the processor? An obvious place would be one of the input lines. The program would instruct the processor to look at this line from time to time. If it is low, then no key is pressed. If it is high, then some key has been pressed. The key number is the last number the processor has outputted.

Most processors have a special sort of input called an *INTERRUPT*, which could have been used to hook up to the common line. When the interrupt goes high, the program jumps to another subprogram which 'services' the interrupt. Here the servicing would involve getting the last number outputted from the register.

APPLICATION NOTE – DIGITAL-TO-ANALOGUE CONVERSION

You may wish to set up the computer so that it can control smoothly the speed of a motor, the brightness of a light, or the position of a drawing pen. In all of these applications, a smoothly varying voltage, say between 0 and 10, is needed from a computer that has been designed to work on logic levels 0 and 1 of 0 V and 5 V respectively. Some sort of conversion is needed. Consider a *digital-to-analogue converter* which takes in a 4-bit digital number and gives an output voltage between 0 and 15. You will need to know how an operational amplifier works, adding up voltages at its input. Look at figure 4.16. Here an op-amp is shown adding three input voltages.

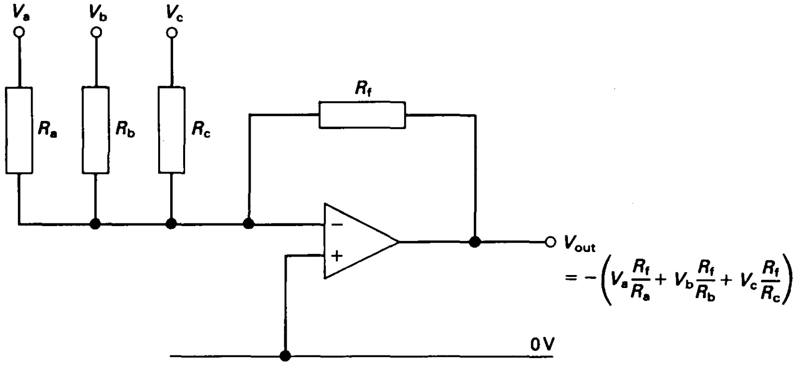

V_a
V_b
V_c
R_a
R_b
R_c
R_f
−
+
V_{out}
$= -\left(V_a\frac{R_f}{R_a} + V_b\frac{R_f}{R_b} + V_c\frac{R_f}{R_c}\right)$
0 V

Figure 4.16
A reminder of how op-amps can add several voltages. Note how the resistances R_a to R_c determine by how much each voltage is multiplied before the addition.

Note how the values of the resistors in each input determine by what amount that input voltage is multiplied. If resistors are used to form a binary sequence *R*, 2*R*, 4*R*, 8*R* and connected to, say, a 1 V supply via switches, then closing the switches in binary sequence will give an output voltage that will rise by equal increments. Figure 4.17 shows the circuit and a table of the output voltages for each binary input from 0000 to 1111.

Of course, the switching is done by logic circuits which could be put together with the resistors and op-amp on a single chip and which could

Switch positions 3 2 1 0	Output voltage
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11
1 1 0 0	12
1 1 0 1	13
1 1 1 0	14
1 1 1 1	15

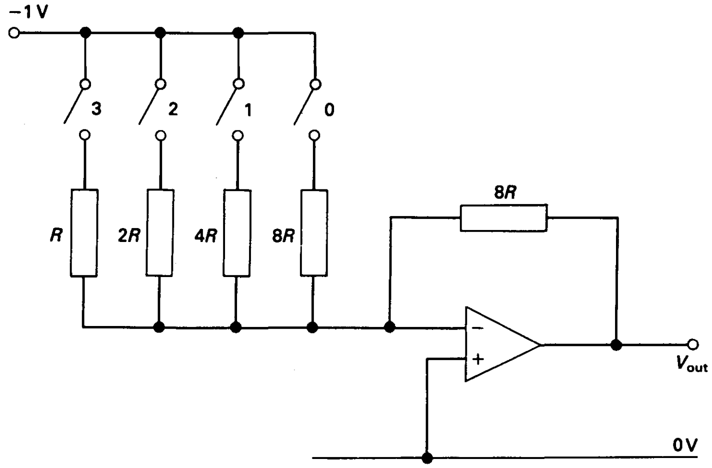


Figure 4.17
A digital-to-analogue converter. The table shows the voltage output for all combinations of switches, open or closed.

then be marketed as an 'integrated D-to-A converter'. Although you could certainly make this circuit work in the laboratory, commercial digital-to-analogue converters, like the DAC 08 8-bit converter, use a slightly different technique called the *R–2R ladder technique.* This method needs only two different resistor values, and so is much more easy to integrate on a chip than the method of resistors in a binary sequence. Eight resistors in a binary sequence would imply using resistors with a wide range of values, 1 kΩ to 128 kΩ for example, and this is hard to achieve.

The *R*–2*R* ladder technique is based on the remarkable behaviour of the *R*–2*R* ladder, like the one shown in figure 4.18.

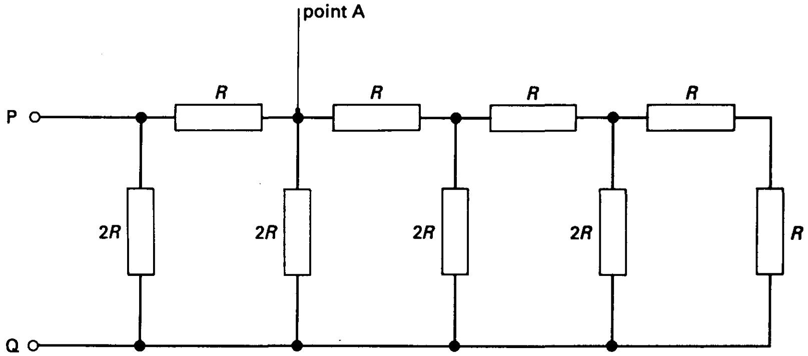


Figure 4.18
The *R*–2*R* ladder. The resistance measured between P and Q turns out to be *R*.

The first feature of this ladder is that its input resistance (measured between P and Q) is just *R*. Check this for yourself by working back from the other end of the ladder. There, two *R*s in series give 2*R*, but this is in parallel with the vertical 2*R*, giving *R* equivalent resistance. Then this R is in series with the next horizontal R, and so on. For a ladder of any length, the input resistance is just R.

The second feature is to do with currents in the vertical resistors. Assume 16 mA flows in through P and out of Q. Look at the resistors around point A, shown in figure 4.19.

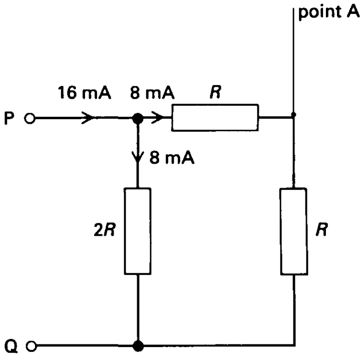


Figure 4.19
Showing how currents split at nodes. The value of the rightmost resistor R is the equivalent resistance of the circuit to the right of point A in figure 4.18.

The vertical resistance, R, at A is of course the equivalent resistance of all resistors to the right of A, according to the first feature mentioned. You can see at once that the 16 mA must divide into two 8 mA currents at point P, since the resistance of both branches from P to Q is the same. The point is that the current is halved at each of the nodes, where vertical and horizontal resistors join together. So, as shown in figure 4.20, the currents flowing in the vertical resistors form a binary sequence.

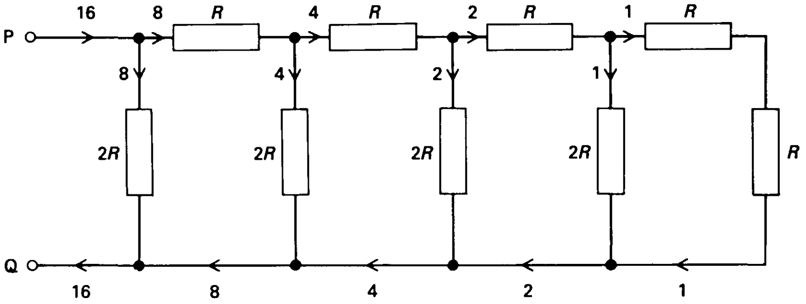


Figure 4.20
All currents shown in this '4-bit' R–$2R$ ladder are in mA. The ladder has the property of dividing the input current of 16 mA into a binary sequence.

This resistor ladder can be used very easily to make a precision digital-to-analogue converter. The circuit is shown in figure 4.21.

The operational amplifier in this circuit is connected as a current-to-voltage converter. (If you have not studied op-amps in detail, if the current into the '–' or 'inverting' input is 1 mA, then the output voltage is $-1\,\text{mA} \times 1\,\text{k}\Omega = -1\,\text{V}$.) In figure 4.21, the switches S_0 to S_3 lead their currents to the I-to-V converter, which receives $8 + 4 + 2 + 1 = 15\,\text{mA}$,

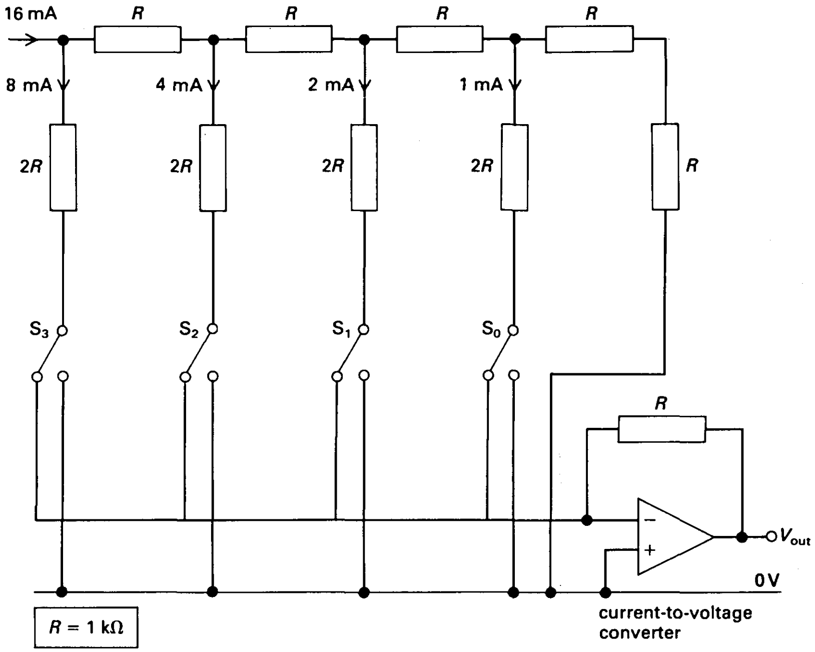


Figure 4.21
The R–$2R$ ladder D-to-A converter. Switches S_0 to S_3 switch currents either into the current-to-voltage converter, or else down to 0 V. These could be high-speed transistor switches. If $R = 1\ \text{k}\Omega$, then the I-to-V converter produces 1 V output per mA of input current.

and so produces an output voltage of −15 V. If only S_0 and S_3 had led their currents to the I-to-V converter, and S_1 and S_2 had been connected to 0 V, then $8 + 1 = 9$ mA would have arrived at the I-to-V converter and been converted to −9 V. So you see that any 4-bit binary number from 0000 to 1111 is converted into its equivalent voltage, from 0 to −15 V. Figure 4.22 gives a table of all outputs.

Switch positions S_3 S_2 S_1 S_0	Output voltage (V)
0 0 0 0	0
0 0 0 1	−1
0 0 1 0	−2
0 0 1 1	−3
0 1 0 0	−4
0 1 0 1	−5
0 1 1 0	−6
0 1 1 1	−7
1 0 0 0	−8
1 0 0 1	−9
1 0 1 0	−10
1 0 1 1	−11
1 1 0 0	−12
1 1 0 1	−13
1 1 1 0	−14
1 1 1 1	−15

Figure 4.22
Table of output voltages obtainable from the converter shown in figure 4.21.

That is the principle of the converter. The commercial DAC 08 chip from Precision Monolithics incorporates transistors as switches. Currents can be switched faster than voltages (the DAC 08 takes only 85 ns to respond to a change in binary number), which is another reason why the *R*–2*R* ladder technique is popular.

Interfacing a digital-to-analogue converter with a microprocessor is very easy; it can simply be hung on the data bus via a latch which is enabled by the usual output signal. A binary number to be converted is put on to the databus, then the latch is enabled and the data allowed in and passed to the converter. When the enable signal is removed, the binary data is of course held in the latch. This has the purpose of holding the D-to-A output voltage steady while other information is on the data bus. This is shown in figure 4.23.

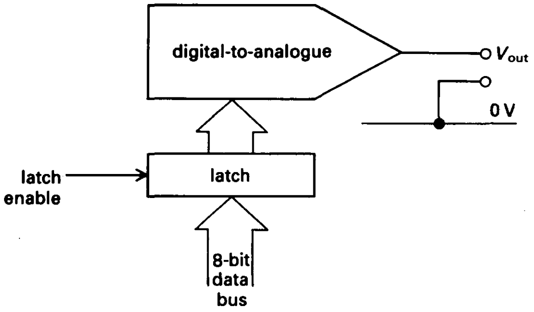


Figure 4.23
Connecting a D-to-A converter to a microprocessor is simple. A single latch is required; the latch enable signal is an output signal generated by the microprocessor.

A rather nice use of the digital-to-analogue converter you might like to try involves graphics: drawing pictures on an *x*–*y* plotter, or on an oscilloscope screen using the *x*-plates and *y*-plates. Two converters are needed, one for the *x*-direction, and the other for the *y*-direction. As shown in figure 4.24, each latch is controlled by its own output signal.

It works like this: data destined for the *x*-position are put on the bus, and the *x*-output signal latches this into the *x*-latch, and the data are converted by the *x* digital-to-analogue converter. The *x*-output signal returns low, and the *x*-data is latched safe. The *y*-position data is then put on the bus, and latched into the *y* converter via the *y*-output signal, and converted. The plotter pen is now at point (*x*, *y*). Then the new *x* position is obtained, then the new *y* position. The plotter pen thus steps from point to point, first along the *x*-axis, then up the *y*-axis. Lines drawn this way are in reality made up of small steps.

The theory of D-to-A conversion sounds nice and simple. When you get around to building a converter, you will learn a lot about electronics, solving many problems. One of the problems involves 'glitches'. These are unwanted spikes which appear at the output of a D-to-A converter when the input number changes. They are due to circuit imbalances when the transistors actually switch the currents. This explanation is incomplete, but the headaches you get trying to deglitch are quite real.

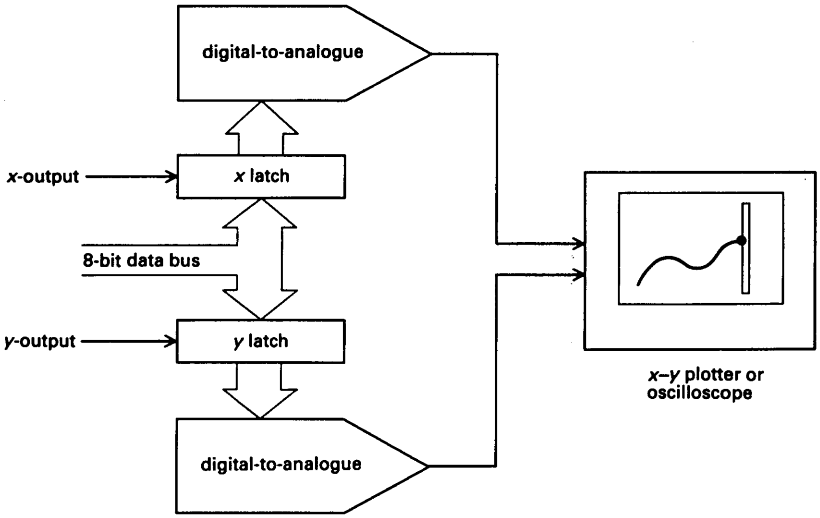


Figure 4.24
Two D-to-A converters, each with its own latch, are able to drive an *x*–*y* plotter or an oscilloscope. This is a cheap and easy way of producing excellent graphics.

APPLICATION NOTE – ANALOGUE-TO-DIGITAL CONVERSION

Perhaps one of the most interesting applications of microprocessors for the scientist and engineer is the A-to-D converter. This device will convert a continuously variable signal into digital bits which may be digested by the computer bus. The signal could be a p.d. measured in some experimental circuit, or it could be the output of a transducer – a device that measures pressure, temperature, magnetic field, pH, or light intensity, for example, and converts these to a voltage. Using an A-to-D converter, an experimenter can build automatic, programmable measuring instruments. Commercial 8 to 16 bit A-to-D converters can be easily obtained fairly cheaply, although price here is usually an indication of speed. How do they work?

There are three well-tried methods of making an A-to-D converter, and each of these involves making a comparison. Figure 4.25 shows the first method, the *linear ramp* technique.

Here, the input voltage V_{in} is compared with the output of a digital-to-analogue converter, using a chip called a 'comparator'. This chip compares its two inputs and produces a logical high output if V_+ is greater than V_-. If, on the other hand, V_- is greater than V_+, then the output is logical low. (If both inputs are equal, the comparator could flip either way, but it is biased to flip one way according to its particular use.) Its action is summarized in figure 4.26.

In the linear ramp circuit, the comparator receives the input voltage V_{in}, on one input; the other comparator input receives a voltage from the D-to-A converter which is driven by a counter. Now suppose that the voltage to be converted, V_{in}, has binary value 0101. The counter, which has been reset to binary 0000, starts counting upwards, 0001, 0010, and so on. After each count up, the D-to-A converts the binary

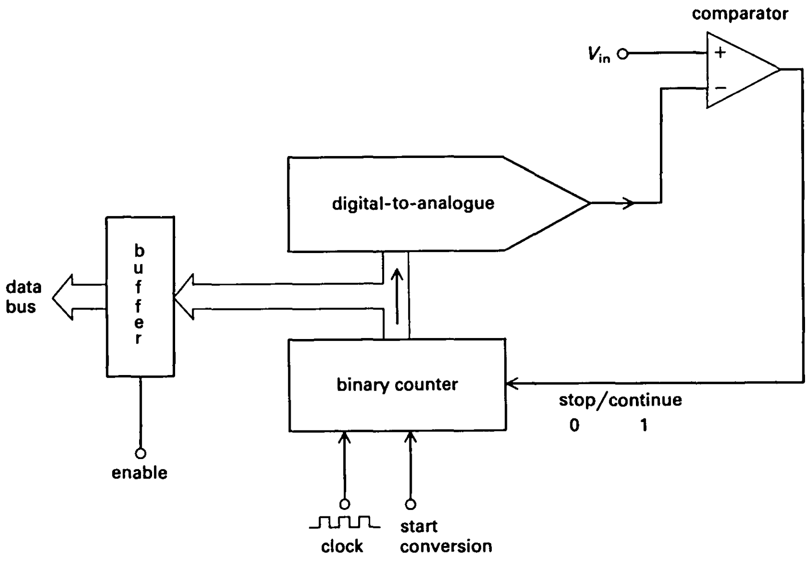


Figure 4.25
Components of a linear ramp converter. A D-to-A converter is driven by the binary counter. The voltage produced, which is a staircase waveform, is compared with the input voltage. When the two are equal, the counter stops, and then holds the binary value of the input voltage.

number to a voltage which is compared with V_{in}. Assume the counter has reached 0011. The comparator receives voltage 0011 from the D-to-A converter, and compares it with V_{in}, 0101. It sees that the D-to-A converter's voltage is too low, and so outputs a logical high. This is fed back to the counter, which commands it to continue counting. It continues from 0011 to 0100 to 0101. At this point the comparator receives V_{in}, 0101, and also 0101 from the D-to-A converter. The inputs are equal, and the comparator's output goes logical low (it has been

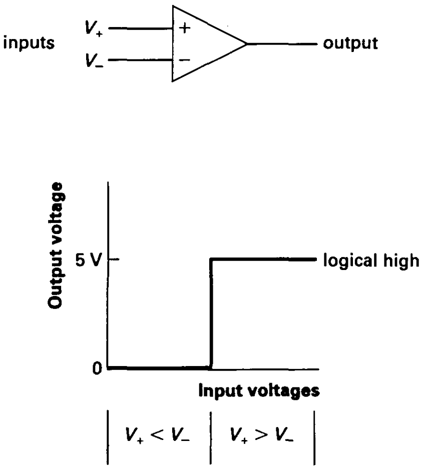


Figure 4.26
Action of a comparator chip. The graph shows the output is 5 V (logical high) when the voltage V_+ is greater than V_-. When V_+ is less than V_-, the output is 0 V (logical low).

biased that way). This low, when fed back to the counter, tells it to stop counting. The counter now holds 0101, the binary value of the input voltage. The conversion is complete, and the 4-bit number can be enabled on to the computer's data bus and read into memory.

Figure 4.27 shows this conversion; the step output of the D-to-A converter as the counter increments until the D-to-A voltage equals V_{in}.

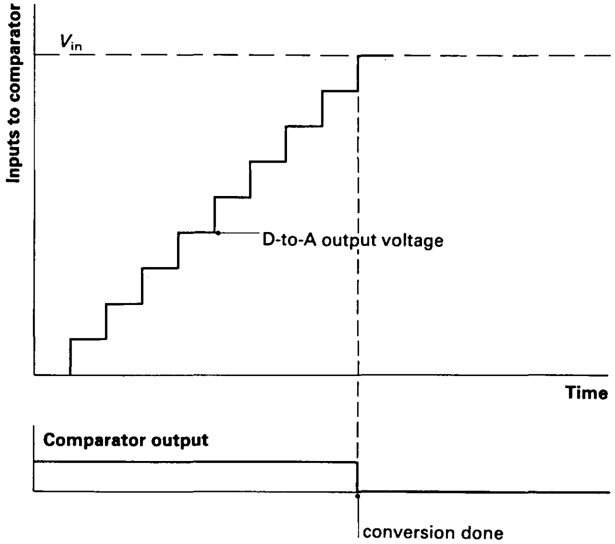


Figure 4.27
The technique of linear ramp conversion. The D-to-A output increases in equal steps until it is equal to the input voltage. Then, the comparator signals that the conversion is done. The counter, which had been driving the D-to-A converter, is stopped, and holds the binary value of the input voltage.

The comparator's output is also shown dropping low at this point. In a working circuit, a little more logic is needed, to reset the counter before each conversion is started and to signal the CPU when the conversion is done, so the result can be enabled on to the bus.

It is quite a straightforward procedure, but rather slow. For a large input voltage, the counter may need to count up to a large number. An 8-bit converter would need a maximum of 255 counts. A typical clock driving the converter may run at 1 MHz, needing 1 μs per count, or up to 255 μs for a single conversion. Such pedestrian behaviour may be fine if you want to measure the height of water in a river once per hour, but totally inadequate to record the behaviour of a millisecond pulsar.

Microprocessor-controlled measuring systems are usually used in the two extremes of either very infrequent (once per hour, day...) or very frequent (thousands or millions per second). These are areas where humans would not perform very well. The problem with quick or transient measurements is illustrated in figure 4.28.

You know that a certain signal has a large initial peak and then decays down. You start the ramp converter at the instant the signal appears (you cannot anticipate), but by the time the counter has counted a hundred or so pulses, the input voltage has long since

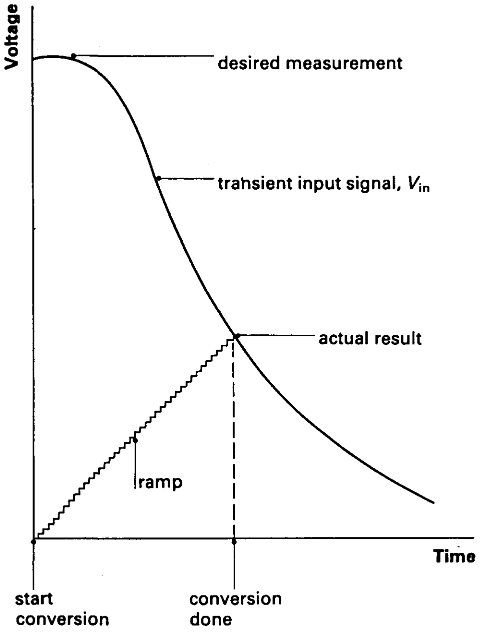


Figure 4.28
What happens when a ramp converter tries to capture a transient signal. By the time the conversion is complete, the desired voltage has passed. Ramp converters are useful only for slowly changing signals.

changed, and the output number is nowhere near the peak value you wanted. It is a big problem: a faster conversion method is needed.

The second method of conversion is called *successive approximation*, and does rather better. Here the input voltage, V_{in}, to be converted is again compared with the output of a D-to-A converter. Instead of being driven by a counter, the D-to-A converter is driven by a special logic circuit called a 'successive approximation register' (SAR). A comparator is used, and its output is fed back to the SAR, telling it what to do. The circuit is shown in figure 4.29, and the process is best illustrated by an example. Think of the SAR as a 4-bit register with an input coming from the comparator:

Assume the analogue input voltage V_{in} has binary value 0010. The following sequence occurs:

1 The SAR outputs 0111, bit 3 being set to zero, all the others to 1. The D-to-A converts this and sends the equivalent voltage to the comparator, which receives V_{in}, 0010, on its other input. The comparator says 0111 is too big, and so outputs a logical 0 to the SAR. The SAR latches this 0 into bit-3 position.

2 The SAR now makes bit 2 zero. This, with the previous latched zero, makes the output 0011. The D-to-A converts this to a voltage which the comparator says is too big, and so outputs a logical 0 to the SAR. The SAR latches this into bit-2 position. The SAR now holds 0011.

3 The SAR now makes bit 1 zero, and so outputs 0001, bits 2 and 3 remaining latched at zero. The D-to-A converts, and the comparator

says that this voltage, 0001, is smaller than 0011, V_{in}. So the comparator outputs a logical 1 which is latched into the SAR. The SAR holds 0011, bits 3, 2, and 1 being latched at 0, 0, and 1 respectively.

4 Finally, the SAR makes bit 0 go low, outputting 0010, bits 1 to 3 remaining latched. The D-to-A converts, and now the comparator sees two equal voltages and outputs a logical 0 which is latched into the SAR (note that the comparator is biased).

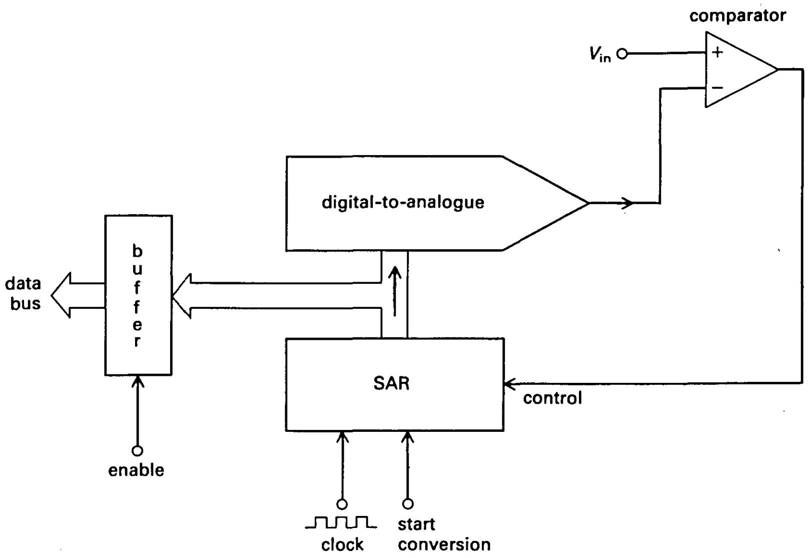
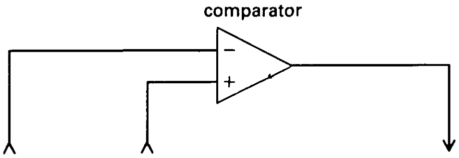


Figure 4.29
Components of a successive approximation converter. The successive approximation register (SAR) drives the D-to-A converter, which passes on the equivalent voltage to the comparator. This compares the approximation voltage with V_{in}, the input voltage. The outcome of this comparison tells the SAR what to do next.

The conversion is done, and the SAR's output, 0010, can be enabled on to the data bus and read into memory. Commercial SARs have a 'conversion done' signal to tell the CPU it may transfer the digital information. Figure 4.30 shows the comparator's input for the above four stages of conversion and the logic signal sent back to the SAR with its meaning.

Note also that the output of the comparator is 0, then 0, then 1, then 0, as the conversion proceeds. This is the correct binary value for V_{in},



	D-to-A output	input voltage	control to SAR	
1	0111	0010	0	too high
2	0011	0010	0	too high
3	0001	0010	1	too low
4	0010	0010	0	just right – both voltages equal

Figure 4.30
The input to and output of the comparator for the four steps of successive approximation.

except that it is sent out bit after bit. This is an example of a *serial* format of data which you will meet in the next section.

It took just four comparisons to convert the input signal into a 4-bit binary number. For a standard 8-bit conversion, only 8 comparisons are needed. That is much faster than the maximum of 255 comparisons for the linear ramp technique. The AD571 SAR-type converter comes with the D-to-A converter, SAR, comparator, and buffer on an 18-pin chip taking just 25 μs for a conversion. For several thousand US dollars, you can even buy a SAR-type converter which will manage 10 million conversions per second.

Here is how the SAR converter handles the transient signal.

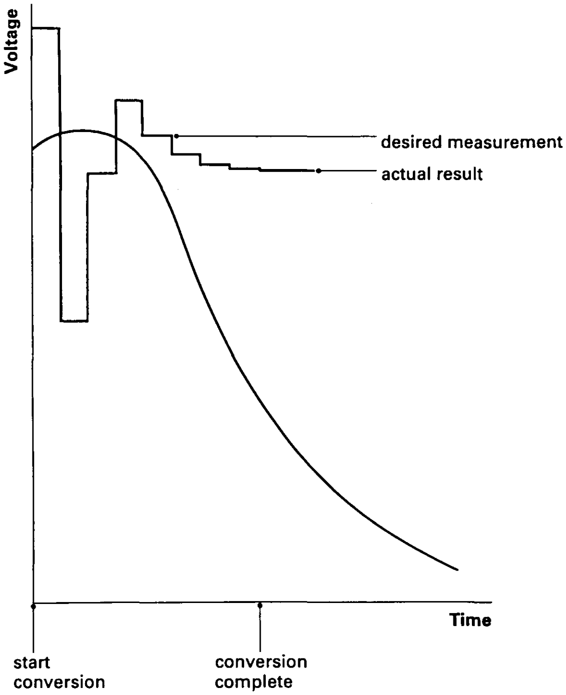


Figure 4.31
Successive approximation conversion technique. Note how each D-to-A output increment is half of the previous increment. Here an 8-bit converter makes its eight comparisons, but only the first four are able to home in on the input voltage.

It gets much closer to the desired value, but there is an error of another sort. Unlike the ramp converter which stops when it has reached the correct code, the 8-bit SAR must always execute its 8 comparisons. In this example, the converter fails to work properly after the fourth approximation. Its final result is a little too low. This problem is not major – in the example here, the converter was not given a fair chance. The engineer needs only to remember that this may happen, and to avoid it. In any case, the SAR technique is the most popular today, and for most laboratory uses represents the best trade-off in terms of price versus speed.

The real winner for speed is the *flash converter*, which takes only nanoseconds to convert; the Plessey SP9754 is a 4-bit flash converter which can convert at 100 million conversions per second. That is fast enough to digitize video signals, and that is the area where you will find flash conversiòn at work. Perhaps you would expect such fantastic performance to require complex circuitry. It turns out that these are the simplest of all converteŕs to understand. It is the actual manufacture which is difficult, and expensive. Figure 4.32 shows the line-up. It is literally a line of resistors of equal values which divide, here, a 10 V reference into equal intervals.

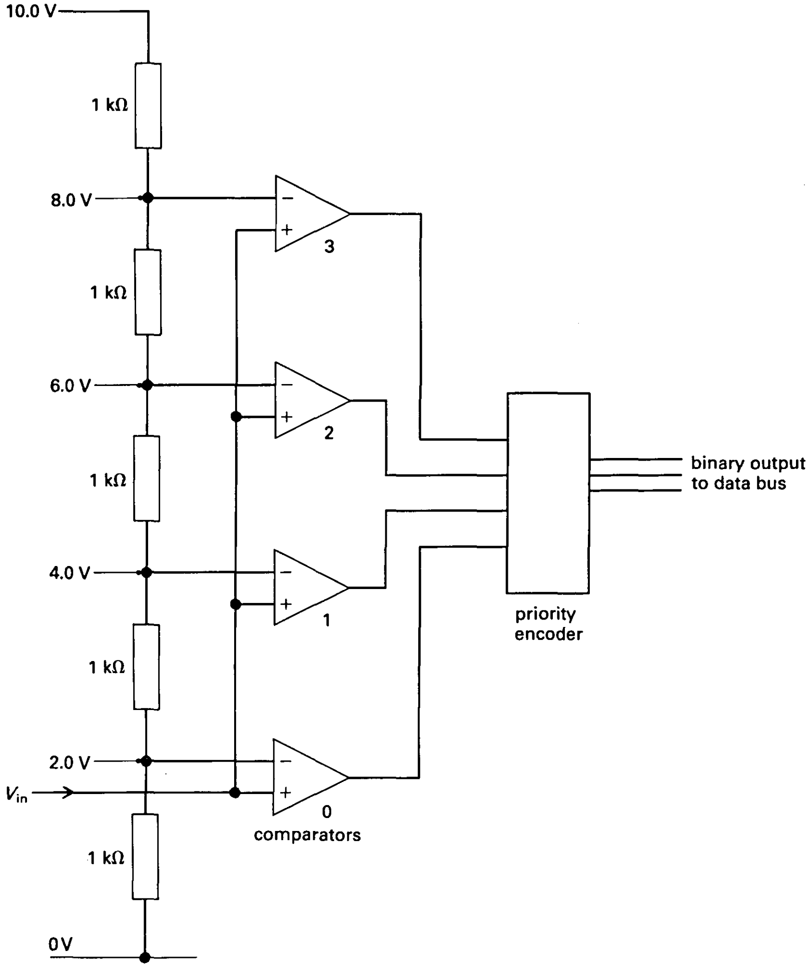


Figure 4.32
Structure of the flash converter. A line of resistors and comparators does the conversion. A priority encoder produces a binary-coded decimal output.

These equally spaced voltages are fed to a line of comparators. The second input to each comparator is connected to a common point, and V_{in}, the signal to be converted, is fed in here. Now, if V_{in} is zero, all the comparators give output logic low, as may be seen from the comparator characteristics shown in figure 4.26. If V_{in} increases to, say, 2.7 V, then comparator 0 will go high, the others remaining low. And if V_{in} lies between 4 and 6 V then *both* comparators 0 and 1 will go high. So as the input voltage rises, each comparator will, in turn, go logical high. Figure 4.33 shows the possible comparator outputs for any V_{in}.

Input voltage range **(*V*)**	**Comparator outputs** 3 2 1 0	**Output from** **priority encoder**
0 to 1.9	0 0 0 0	000
2 to 3.9	0 0 0 1	001
4 to 5.9	0 0 1 1	010
6 to 7.9	0 1 1 1	011
above 8	1 1 1 1	100

'overflow' bit

Figure 4.33
Flash conversion of any positive input voltage. The comparator outputs don't form a particularly nice binary series, but the priority encoder puts this right, as shown by its outputs. Note that the last line in the table is the 'overflow' condition, where the only information provided by the circuit is that V_{in} is more than 8 V.

The only difficulty is that each comparator does not respond to a range of voltages, but is always on if V_{in} is bigger than its own turn-on voltage. That means the four comparator outputs do not form a very nice pattern. This is cleared up by the final part of the flash converter, the 'priority encoder'. When two inputs are high, for example, it chooses the one that came from the comparator with the bigger turn-on voltage. It then produces a regular binary output as shown in figure 4.33. This is then enabled on to the data bus and stored in memory.

Flash conversion involves making comparisons in parallel, and for each V_{in} just one of these comparisons is used in making the output binary number. There is no counting or approximating.

This example using four comparators could classify any input voltage into one of four intervals. Its resolution is pretty poor. The 8-bit resolution which is a standard minimum resolution for most work would require $2^8 = 256$ intervals and hence 256 comparators and some 257 resistors all carefully matched. Even with the use of laser-trimmed integrated resistors, resistor matching and making good comparators is not easy. Hence the high cost of flash conversion.

You may have thought that in the above example with four comparators, there were five intervals of voltage, the last being 8 to 10 V. That is not true, since any signal greater than $7.\dot{9}$ V will turn on the last comparator; 55 V would do equally well. The last comparator (number 3) only tells you if the input signal is outside the interval 6–$7.\dot{9}$ V, and in this way it is an *overflow* indication.

Finally, what about the speed of conversion? This is simply the sum of the time it takes the comparator to respond (25 ns for the NE529) and the time it takes for the encoder to work (10 ns for a 74S148), which comes to around 35 ns. That is why it is a fast method.

The three conversion techniques described in this section all have their uses. The SAR method seems about the most widely used today (1985). If you want to make an A-to-D converter you can build any of the circuits described here and they will work. If you want to work at 8 bits, do not build, buy as it is cheap. For 10 or 12 bits you should build an SAR converter. Use an LM311 comparator, a 74C905 or DM2504 SAR, and a DAC 100 or DAC 10 digital-to-analogue converter for 10 bits. The choice of D-to-A converter is important, but the ones suggested (marketed by Precision Monolithics Incorporated) work well and are not that expensive. Then you will have a 15 μs A-to-D converter accurate to 100 mV in 10 V.

SERIAL COMMUNICATION

The CPU is often transferring large amounts of data to other devices, such as printers, disk drives, cassette tape recorders, often down a telephone line or by a satellite link. You have seen how a CPU can output data in 'lumps' of 4 or 8 bits to D-to-A converters and the like, and this method of communication will also work well for most printers, where the data bus is carried over by a long stretch of 8-or-more-conductor 'ribbon' cable. But it is no good for cassette recorders, which only have one wire for 'record' and another for 'playback' – unless they are stereo. So 4 or 8 bits cannot be sent at once.

The *serial* method of communication sends, for example, 8-bit data over a single wire, not eight wires. Instead of sending all the bits simultaneously, they are sent one after the other, in a series, hence the name. The two modes of communication are shown in figure 4.34, in which the data byte 01001101 is being sent. There are a few ways in use to transmit serial data, but in the interests of uniformity, assume the existence of just one, *ASCII* (American Standard Code for Information Interchange). Every key of a typewriter, plus the control keys of a teletypewriter, have been given a unique 7-bit code which is transmitted over a serial line. In this code, our data 01001101 is the letter 'D'.

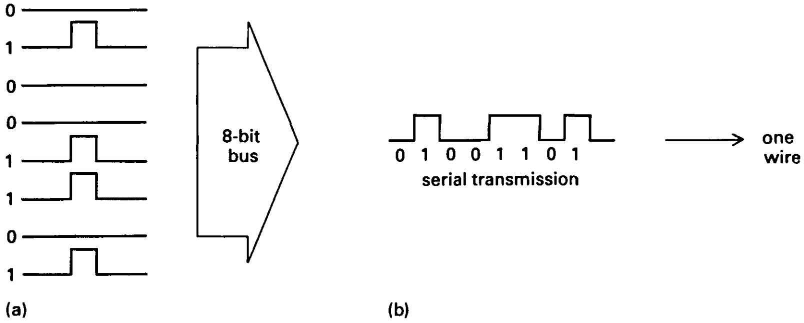


Figure 4.34
A comparison of (a) byte and (b) serial transmission of the same 8-bit number 01001101.

Figure 4.35 shows how the letter 'D' would be sent on a serial line. First a 'start' bit is sent out, and then the 7 bits of the data, starting with the least significant bit. Finally, a 'stop' bit is sent.

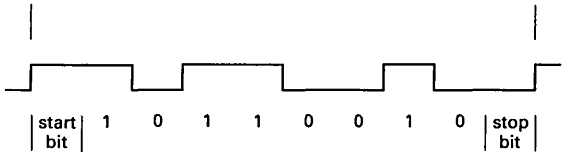


Figure 4.35
Sending a letter 'D' on a single line. The ASCII code for D is combined with serial control bits, start and stop.

You cannot stick this signal straight into a cassette recorder, since it does not have enough bandwidth to cope with the square edges of the pulses. The best thing to do is to pass the pulse train through a modulator (a *modem*). When a logical 1 goes in to the modem, it may produce a burst of sound at frequency 2400 Hz. And when a logical 0 is picked up by the modem, it may produce a 1200 Hz tone burst. These sounds are easily recorded by a typical cassette recorder.

It is quite interesting to see how the CPU converts an 8-bit lump of data into a serial string. The 8-bit data is loaded into the accumulator with a MOV M→Acc instruction. Then a SHR ('shift accumulator right') instruction is performed. This is shown in figure 4.36.

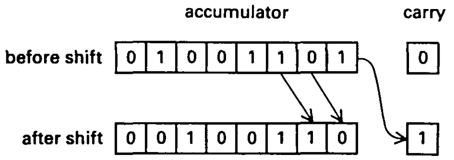


Figure 4.36
How a 'shift accumulator right' (SHR) instruction shifts the lowest bit of the 8-bit number out of the accumulator and into the carry cell.

You can see that all the bits are shifted one place to the right. The rightmost bit is placed into the carry cell. Now the CPU looks at the carry cell, and it makes its serial output line (SOD-line) equal to the carry bit, in this case a 1. If the CPU repeats this another seven times, the 8-bit data will come out of the SOD pin in serial format. This is shown in figure 4.37.

The reverse question, how to convert incoming serial data into 8-bit data, is a little more complex, at least from the point of view of writing the software. Most CPUs will have a SID connection, a serial input line, and the rest is programming.

DIRECT MEMORY ACCESS (DMA)

Have you ever thought about writing programs to make those beautiful animated pictures or coloured processed pictures you normally associate with NASA? Before you start, think about this calculation. To make a medium-resolution picture of 256 by 256 dots needs 256 x 256 = 65 536 bits of information which is 65 536/8 = 8192 bytes. (If you wanted colour, you would need 8192 x 3 = 24 K.) To make a flicker-free

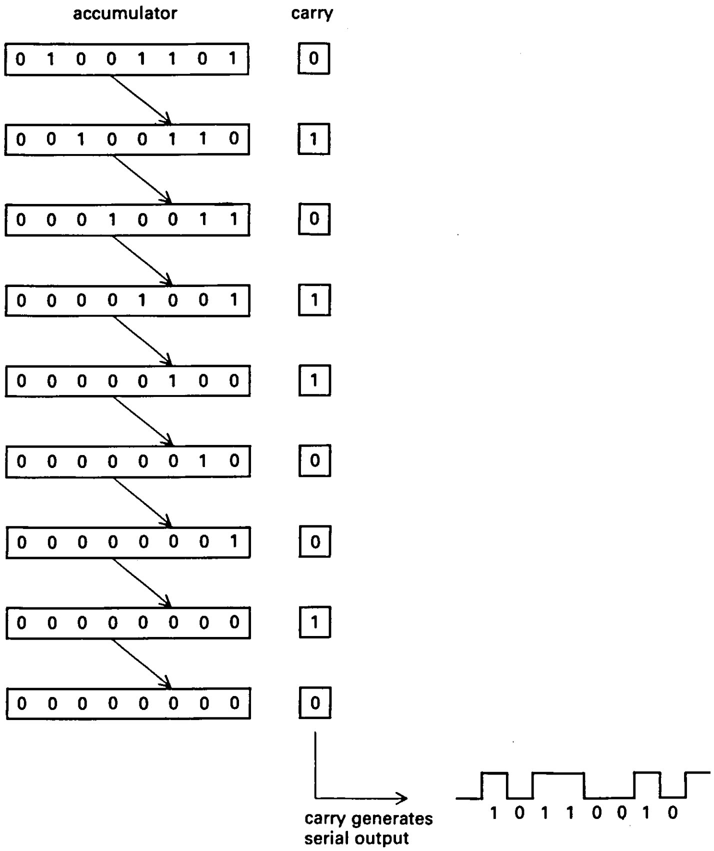


Figure 4.37
Successive shift right operations send each bit of the 8-bit number through the carry cell, where they may be used to produce the serial output via the SOD pin of the CPU.

television picture, it needs to be scanned 50 times a second which is (1/50) s = 20 ms per picture, and is equivalent to (20/8192) ms, or around 2.5 µs per byte. Now an advanced processor like the 8086 or 8088 takes 3.5 µs to move a byte around, using MOV-type instructions. Clearly the processor cannot cope with even medium-resolution graphics, it is too slow, and anyway it is no good having the CPU totally committed to the picture on the screen when other things need doing, like keyboard scanning, data calculations, and so on.

The *DMA controller* could be used to solve this problem. It hangs on the bus, and when activated forces the CPU to relinquish the bus. The DMA controller then transfers data completely on its own, without any programming need. A transfer time of one byte in 0.6 µs can be achieved. DMA transfers are used to communicate with floppy disk drives, fast magnetic tape units, and some television screens. However, for the latter, even DMA is too slow, and the usual solution is to build special CRT interfaces using memory coupled with discrete logic ICs which can transfer data in tens of nanoseconds.

All of these applications are compatible with the computer described in Chapter 2.

A CONTROL AND DATA AQUISITION COMPUTER

Most of the ideas of this chapter can be put to use in building up CONDAC. CONDAC's purpose is Control and Data Aquisition. Before we see how CONDAC could be used in the laboratory, let's look at its circuitry. Don't be put off if it looks complicated, it's not; it's simply a combination of some of the circuits you have just studied.

Here is a description of CONDAC, shown in figure 4.38. To the bottom left of the CPU are the memory boards, and to the bottom right is the keyboard with its sixteen keys. Next to this is the display. This is an eight-digit, seven-segment display. At the top right of the circuit diagram, there are two D-to-A converters, labelled *x* and *y*. To the right of these is the A-to-D converter, a successive approximation type. Finally, in the bottom right of the circuit there is an input buffer which can read the states of eight logical inputs, and also an output latch which can output eight logical signals.

Now for a closer look at some features of the circuit. Firstly, the decoding. In Chapter 2 you learned how different memory boards were selected using a decoder. In the last section of the same chapter, you saw how input and output devices can be selected by using a decoder. CONDAC has two decoders, one reserved for memory, and the other for input–output. The decoders both have 'enable' connections; if enable is low, the decoders' outputs are all low. If enable is high, then the correct output of the decoder will go high. Notice that the $M/\overline{IO}$ signal is used to enable the memory decoder, and the inverse of this signal is used to enable the input–output decoder. So either memory or input–output decoder is selected at any time. It turns out that this use of two decoders saves a handful of gates elsewhere in the circuit.

Two outputs from the memory decoder are used. One selects a RAM board, which also receives both RD and WR signals, and the other output selects the ROM board, which, being read only, needs only the RD signal to work. The ROM in CONDAC contains the *operating system.* This is a chunk of program which actually gets the machine running; scanning the keyboard, allowing you to enter and run programs. It also looks after the multiplexing of the display, allowing you to read results, check memory contents, and the like. In addition, it makes the cassette interface function, and may even contain subroutines to make the D-to-A and A-to-D converters function. (On larger machines, the operating system would run disk drives, and enable you to edit programs shown on a television screen.)

Next, take a look at the D-to-A ports. These are selected by outputs 1 and 2 of the input–output decoder. The *x*-channel is selected by 1, and the *y*-channel by 2. Since these channels are OUTputting data, they are controlled by the WR signal, which is ANDed with the decoder outputs using the two AND gates shown.

The A-to-D converter is selected by decoder output number 7. Two commands must be sent to the A-to-D converter. First, a 'start conversion', which is an OUTput command. The signal is thus generated by ANDing decoder output number 7 with WR. When the conversion is done, the A-to-D converter must inform the CPU of this fact.

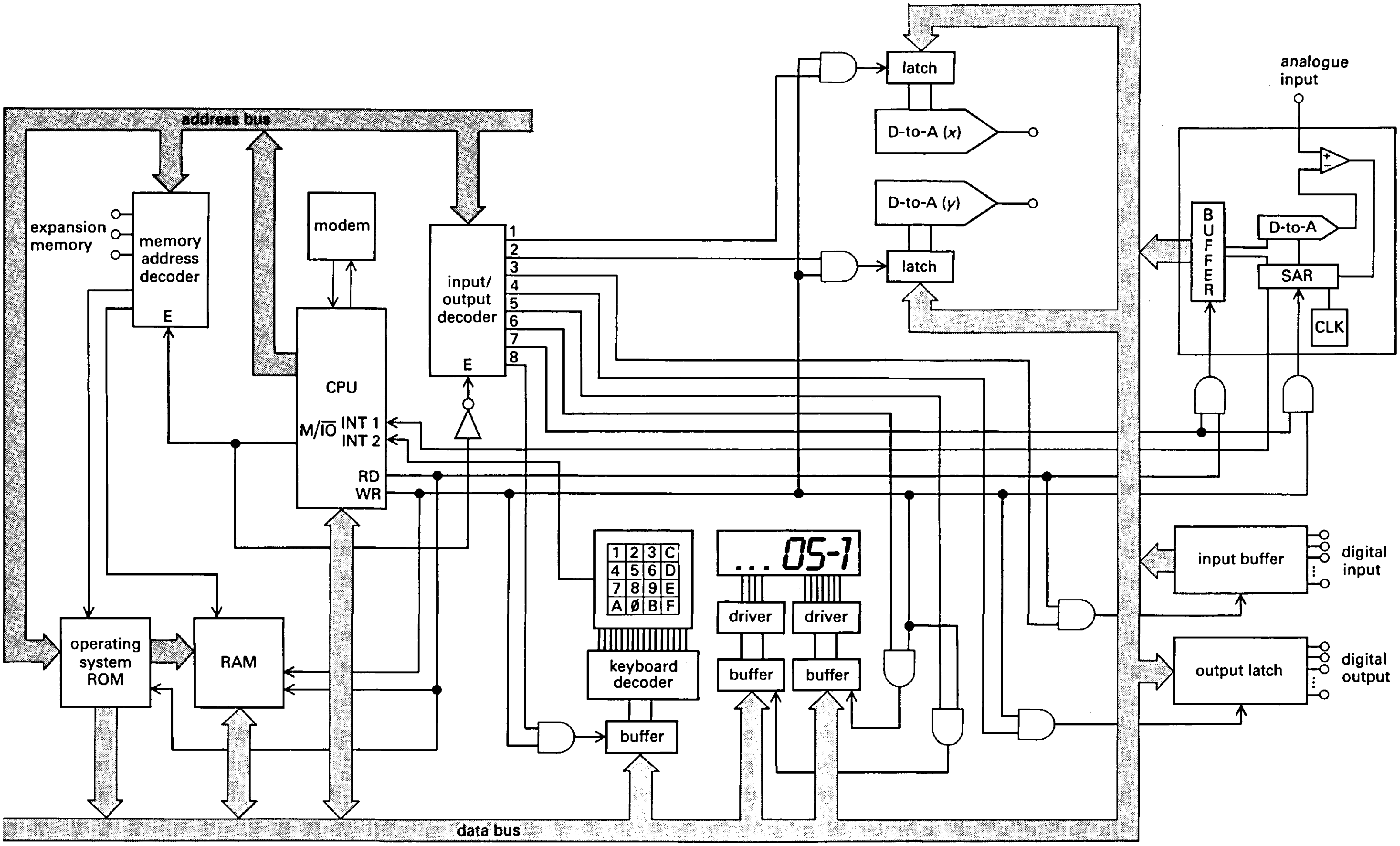


Figure 4.38
Circuit diagram of CONDAC.

It does this by outputting a signal from the SAR to the CPU's interrupt pin, here labelled 'INT 1'. As mentioned earlier, this causes the CPU to jump to a subroutine to input the result of the conversion. The 'interrupt' will interrupt any other task the CPU may be busy with, like scanning the keyboard, or writing a number into the display. Finally, when the conversion is complete, the binary number must be enabled on to the bus, and read by the CPU. So an INput signal must be generated, which is a read; this is done by ANDing decoder output number 7 with RD, and this signal is used to enable the buffer.

You can now think out the remaining chunks of the circuit for yourself, guided by the idea that to switch on, or enable, any chunk of the circuit, you must AND a decoder output with an RD if the chunk is an INput, or AND a decoder output with a WR if the chunk is an OUTput.

Here are a couple of applications of CONDAC in the laboratory. The first, shown in figure 4.39, is a study of animal behaviour. Variables to be measured are temperature, level of food in the trough, and level of water in the bottle. The computer has to control the heating of the cage and the level of light in the cage. The last two are easily arranged: a D-to-A port is used to drive a heater and a second D-to-A to drive a lamp. Power amplifiers will be needed to drive lamp and heater, since both will need more current than a pure D-to-A converter can provide.

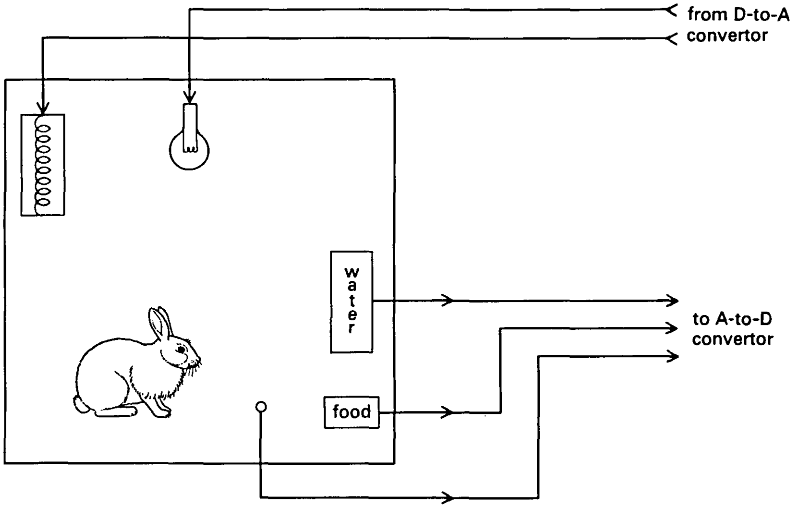


Figure 4.39
Behaviour study of an animal in a cage. Measurements are food level, water level, and temperature. The microprocessor system controls heating and lighting of the cage, as programmed by the experimenter.

Making the measurements: temperature is easy, a transistor or zener diode (LM334 or LM335) are very good, a thermistor not bad. Designing water or food-level detectors is up to the ingenuity of the experimenter. Owen Bishop, in his book *Interfacing to microprocessors and microcomputers*, offers some starting ideas for measurements. But the idea is to convert the level to a voltage which may be measured using the A-to-D converter. Since CONDAC only has one A-to-D input and two are needed, the input must be switched from one signal to the other. A mechanical switch could be used, but there are chips which can do this job very well (CD 4052 or MUX 08). These chips can switch very quickly, and can be driven by CONDAC's digital output port.

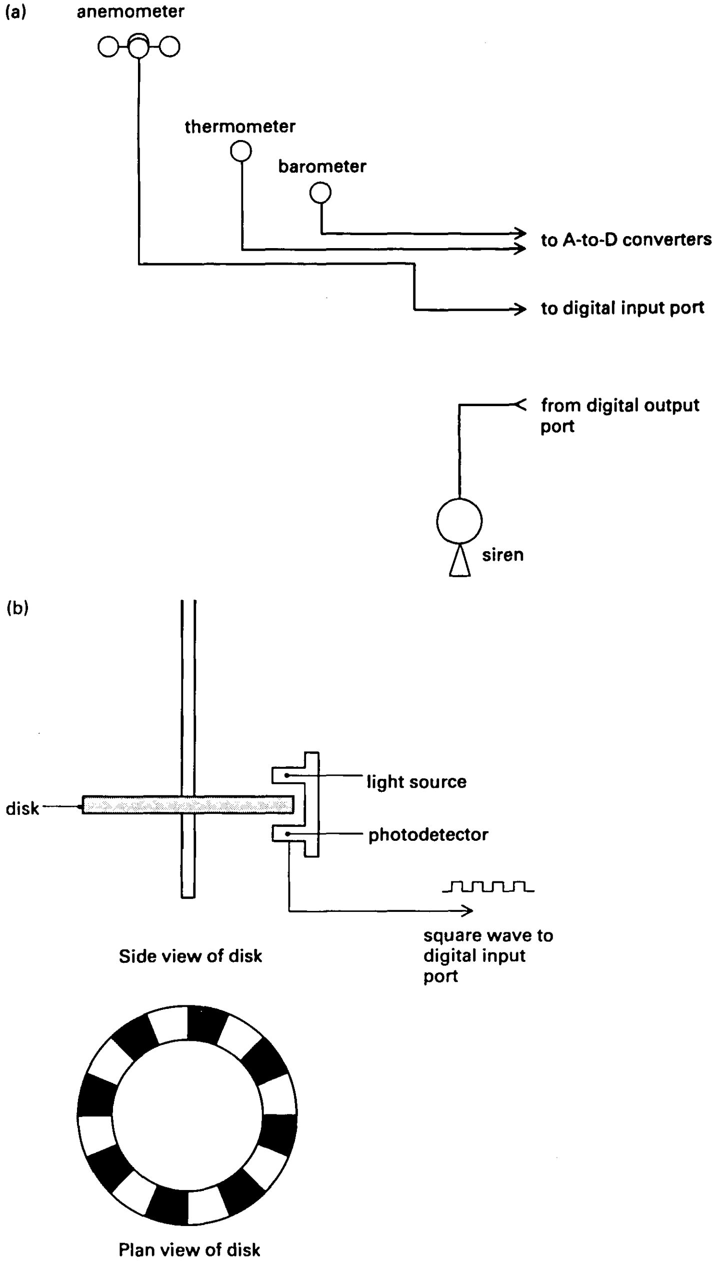


Figure 4.40
(a) A small weather station. Temperature and pressure measurements are made using A-to-D converters. The detailed drawing (b) shows how the speed of the rotating wind-speed shaft is converted to a square wave using a light source and photodetector. This square wave is fed into a digital input of the microcomputer.

The second application, shown in figure 4.40(a), is a small weather station. The measurements are temperature, pressure, and wind-speed, and CONDAC must switch on an alarm siren if the combined measurements indicate a serious storm. Temperature and pressure are analogue signals; converted to voltage, they are both read and recorded using a switched A-to-D converter. It is wind-speed that offers a chance to think. You could connect the anemometer vanes to a dynamo, and measure the output voltage, or current, with an A-to-D converter. But their relation to speed is not simple. A better way is to attach a disk to the vanes. This disk should be made of transparent plastic with opaque , black stripes painted on – figure 4.40(b). Light is passed through the disk and received by a photodetector device. If an opaque strip lies between light and detector, the detector's output is zero. If a transparent strip is in that position, the detector output is high, 5 V. As the vanes and the disk rotate, the output of the detector is a square wave, as alternate opaque and transparent strips cross the light path. The frequency of this square wave must increase as the disk rotates faster, and the increase is linear. So what do you do with the square wave? Feed it to a digital input of CONDAC, and write a program to determine the frequency of the square wave. The instrument is calibrated by recording the frequency at one known wind speed. This is a general method that can be used to measure the speed of rotating shafts, motors, lathes, robot wheels, and so on.

The alarm signal is easy to generate, once the computer has measured the three variables and been programmed to decide that a serious storm exists. CONDAC outputs a binary number to the output port, which then switches on the alarm siren.

In both of these examples the microprocessor is an ideal solution to the how-to-make-the-measurement problem. In both cases measurements have to be made infrequently over long periods of time. Also, the system's response may have to be altered. The animal psychologist may like to study how switching on light and heat with different time periods affects the animal's behaviour, or to have a time lag between heat on and light on. If the control circuit had been made using discrete logic chips, many of the desired changes would have called for massive rebuilding of the electronic circuit boards. With a microprocessor, a simple change in software is all that is needed and the experimenter can do this himself. A microprocessor is not always the best choice; simpler or faster circuits can be made with discrete logic. But in applications where the precise working of the circuit may need to be changed in the future, the microprocessor is the natural choice.

EPILOGUE

Putting microprocessors in their place

You have heard of the enormous power of microprocessors in control functions. You now realize how seriously electronic-component manufacturers take the microprocessor. You may even have experienced the feelings of fun and even satisfaction which come from the creation of your own programs. Now you have read about how microprocessors work you should have, at least, some feelings of awe at the clever designs of controlling signals involved in the machine's architecture. It is all a projection of the power of the human mind. And the parallels between mind and computer are being rapidly researched. The brain is not the most convenient organ to research. Yet thanks to the work of physiologists on cats' and monkeys' brains, brain–computer parallels are becoming quite believable, from the 'and-gate-is-like-a-neuron' level to the 'thought-is-like-a-program' level. Some of the engineers who developed dynamic RAM say they still do not believe that it works. How can dynamic memory be memory when it forgets and has to be reminded? Yet that is also a view held by modern psychologists of the human mind. It does forget, and it has to be refreshed. And this sounds like a fascinating rebirth of *real* natural science, a beautiful study of mathematics, physics, and psychology.

BIBLIOGRAPHY

BISHOP, O. *Interfacing to microprocessors and microcomputers.* Newnes Technical, 1982. This gives some easy interface units to build for any microcomputer, with full constructional details.

CARR, J. J. *Microcomputer interfacing handbook: A-D and D-A.* Foulsham, 1980. This is one of the American 'TAB' series, good for the hobbyist.

HOFSTADTER, D. R. *Godel, Escher, Bach: An Eternal Golden Braid.* Penguin, 1980. This book makes a synthesis of physics, mathematics, art, psychology, religion, and computing. It draws parallels between all of these fields, and moves towards Artificial Intelligence.

HOROWITZ, P. and HILL, W. *The art of electronics.* Cambridge University Press, 1980. This is an excellent text on electronic engineering, from gates to microprocessors, for anyone heading towards the profession.

INDEX

This Index will direct you to explanations of the terms you may look up.

EXECUTE cycle number (These numbers correspond to those in the titles of figures 3.15 to 3.25)	**Mnemonic**	**Op-code or data (*indicates data)**	**Address**	**Effect of instruction**
1	MVI A	0010	0000	Move the number 0111 into register A, ready to be used as an address.
		0111*	0001	
2	MOV M→Acc	0110	0010	Move memory contents into accumulator.
3&4	ADI	1111	0011	Add the number 0101 (5_{10}) to the accumulator, and put result in accumulator.
		0101*	0100	
5	MOV Acc→M	0111	0101	Move accumulator contents into memory.
6	STOP	0000	0110	Stop.
		0110*	0111	Data 0110 (6_{10}) to be added.

(This page may be photocopied for use with the work on pages 34 to 42.)

General editor, Revised Nuffield Advanced Physics
John Harris

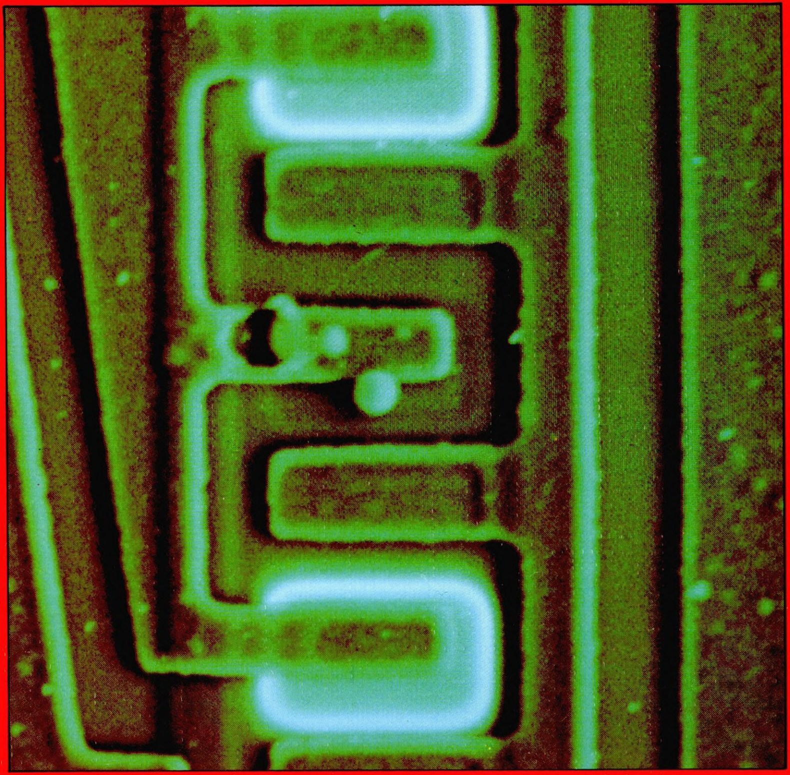
Consultant editor
E. J. Wenham

Author of this book
Colin Price

Microcomputer circuits and processes **is one of the background Readers for the Revised Nuffield Advanced Physics Course.**

This Reader explains how microcomputers work, showing how basic ideas about logic gates and circuits are used in microcomputer systems. It outlines the history of the microprocessor, and looks at the electronics involved, showing how a simple machine is built from scratch, and how the computer program actually runs the microprocessor. It looks in detail at specific parts of microcomputer systems, including memory and analogue-to-digital conversion, and some applications of such systems.

ISBN 0 582 35423 4



ISBN 0-582-35423-4

9 780582 354234

Longman