Operation of a Simple Microcomputer

Chapter 1

This chapter gives a brief introduction to the basic elements which comprise a microcomputer and to the concept of an instruction set.

INTRODUCTION

A microcomputer (MPU) is a collection of digital circuit elements connected together so as to form an information processing unit. This unit is usually composed of three essential elements as shown in Figure 1-1. The system elements are a program memory which remembers or "stores" the program which the system is to execute, a data memory which is used to store the numbers which are being manipulated, and a microprocessor (MPU) which operates on the data in the sequence dictated by the program. For example, if the circuit is required to find the average value of a set of numbers, then the numbers will be stored in the data memory, the averaging program is stored in the program memory and the actual calculations will be done by the microprocessor. It is important to

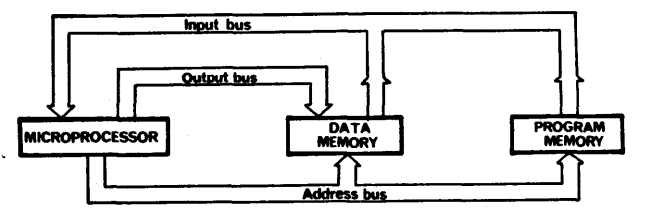


Figure 1-1. Block diagram of a simple microcomputer

recognise at this point that whilst the microprocessor is the focal point of the system, it cannot exist in isolation; there must be facilities for storing the program and there must be some sort of data storage. The program is stored in the program memory as a set of binary characters which are "coded" to represent the various steps which the microprocessor must execute. The microprocessor contains circuits which can decode those instructions and implement the prescribed program steps.

Inside the microprocessor circuit itself there are several "registers" which are used to store binary numbers of particular significance. The most important of these registers are:

(i) *The accumulator:* often abbreviated to acc.; this is the focal point for all data manipulation. Numbers are added to or subtracted from the accumulator; often certain operations such as shift can only be done using the accumulator.

(ii) *The index register:* this is used to store and create data addresses which are particularly important.

(iii) *The instruction register:* in order to make the microprocessor system execute a particular program, the program memory sends a series of commands, or "instructions," to the

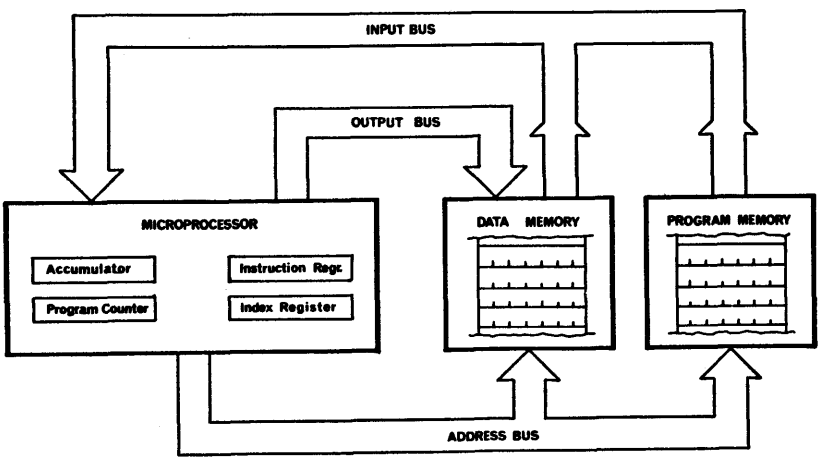


Figure 1-2. Expanded block diagram of microcomputer system showing registers and some memory cells

processor. As each instruction is received by the MPU it is stored in the instruction register. The MPU then carries out the operation required by that particular instruction before receiving the next instruction.

(iv) *The program counter:* often abbreviated to p.c.; this keeps track of the system's progress through the program. Some instructions can modify the way in which the program counter behaves.

Figure 1-2 shows a microcomputer system diagram with some of the internal registers and memory cells drawn in.

An important feature of a microcomputer system is the way in which the three basic system elements, MPU, program memory and data memory, communicate with each other by means of buses. A bus is a set of connections along which parallel binary information can be transmitted. Only one piece of parallel information (usually called a *byte*) can exist on a bus at any one time. For example, in Figure 1-2 it would not be permissible for the program memory and the data memory to simultaneously send information to the MPU via the input bus. Clearly, one of the system elements should assume control of the buses and this task usually falls to the microprocessor. Bus control is done by means of special signals generated by the microprocessor and sent out on various control lines. Sometimes these control signals are grouped together under the heading "Control bus," but this nomenclature is misleading. For clarity, the control signal paths are not shown in Figure 1-2. In the remainder of this chapter, the system connection of Figure 1-2 will be used; it will be assumed that the input bus and the output bus are both 8 bits wide and the address bus is 16 bits wide.

BASIC INSTRUCTION CYCLE

In following a program step-by-step, the microcomputer system uses the bus connections to exchange information in a well-defined manner. The complete sequence of information exchanges which carries out one program step is known as an instruction cycle. A basic cycle consists of just three events:

(i) Fetch the next instruction (i.e., program step) from the program memory to the instruction register.

(ii) Increment the program counter.

(iii) Execute the instruction.

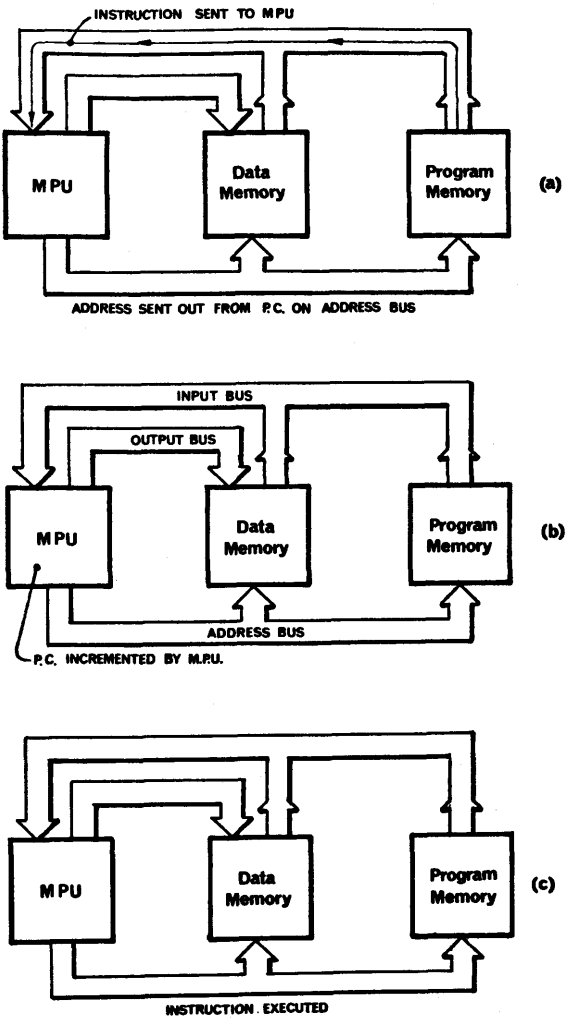


Figure 1-3. Basic instruction cycle of a microcomputer

The instruction cycle may be illustrated by considering the following simple program:

Step 0 Set the accumulator to zero
Step 1 Increment the accumulator
Step 2 Shift accumulator up one place

Assume that the program counter has been set to zero by some external mechanism. First of all, the program counter contents are sent out along the address bus (i.e., address 0 is on the address bus) as shown in Figure 1-3a. The program memory recognises address 0 as one of its own and it sends out the contents of address 0, which contains the instruction "Set accumulator to zero," onto the input bus. The MPU picks up the instruction and loads it into the instruction register. The contents of the program counter are then incremented as shown in Figure 1-3b. Then the instruction is executed, which in this case sets the accumulator to zero, and the state shown in Figure 1-3c is reached. This sequence of events completes Step 0 of the program and therefore constitutes the first instruction cycle. The system carries on to begin the next instruction cycle by sending out the contents of the program counter (now address 1) and the program memory replies by sending the instruction "Increment accumulator" to the instruction register. The program counter is incremented and so the cycle continues.

The instruction cycle given above is sufficient for the operation of a fairly simple microprocessor. However, since the input bus is only 8 bits wide, all instructions would have to be encoded into 8 bits. This only allows for 256 different instructions. This is inadequate for general use, and in order to overcome this problem, the concept of a multiple byte instruction is used. In a multiple byte instruction the complete instruction is encoded into a multiple of the byte size of the machine. For the 8-bit example used here, multiple byte instructions would be encoded into 16, 24, 32 bits, etc. The instruction is then split up into a group of bytes and written into the program as a series of program steps. For example, consider the three program steps:

Step 0 Increment accumulator
Step 1 Store the contents of the accumulator in data memory at address 1010101010101010
Step 2 Increment accumulator

In a particular MPU, the two instructions might be encoded as follows:

Increment accumulator	01001100
Store contents of accumulator in data memory at address 1010101010101010	01110111 : 10101010 : 10101010

In the program memory, the three program steps would then appear as:

Step 0	Address 0	01001100	= Increment
Step 1	Address 1	01110111	= Store
	Address 2	10101010	
	Address 3	10101010	
Step 2	Address 4	01001100	= Increment

Note in particular how the "store" instruction has been split up into three 8-bit bytes.

In order to handle multiple byte instructions it is necessary to modify the instruction cycle a little so that all but the last byte of a multiple byte instruction are treated as instructions requiring no action. Thus the three program steps in the above example would create the interaction shown in Table 1-1.

Note how the 3-byte "store" instruction is transmitted from the program memory to the MPU in three separate instruction sub-cycles using the program counter to address each byte. The MPU recognises from the code of the first byte (Step 1, Cycle 1 that the instruction is a 3-byte instruction and waits until Cycle 3 before executing the full instruction. During the first two cycles it rebuilds the 24-bit instruction from the separate bytes. The time taken to execute two do-nothing commands is wasted. Thus some of the more sophisticated machines miss out the do-nothing commands so that the full instruction cycle for a 3-byte instruction

TABLE 1-1. MULTIPLE BYTE INSTRUCTION CYCLE EXAMPLE

Step 0	Cycle	Send out p.c. contents (0) and fetch instruction Increment p.c. Execute instruction (increment acc.)
Step 1		Send out p.c. contents (1) and fetch instruction
Instruction Cycle	Sub Cycle 1	Increment p.c. Execute instruction (do-nothing)
		Send out p.c. contents (2) and fetch instruction
	Sub Cycle 2	Increment p.c. Executive instruction (do-nothing)
		Send out p.c. contents (3) and fetch instruction
	Sub Cycle 3	Increment p.c. Execute instruction (send contents of acc.)
Step 2		Send out p.c. contents (4) and fetch instruction
	Cycle	Increment p.c. Execute instruction (increment acc.)

would be:

Send out p.c. contents and fetch instruction
Increment program counter
Send out p.c. contents and fetch instruction
Increment program counter
Send out p.c. contents and fetch instruction
Increment program counter
Execute instruction.

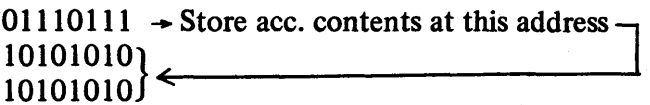
In practice, the system timing can become even more complicated because different instructions can take different times to execute. This topic will be covered in more detail in Chapters 2, 3 and 5.

Multiple byte instructions nearly always arise when the instruction contains two pieces of distinct formation:

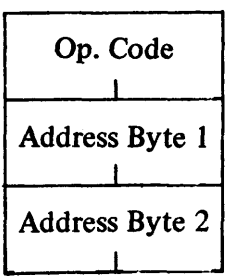
(1) The operation to be performed (e.g., store contents of accumulator)

(2) An address relating to the data on which the operation is to be performed.

In the example given above, the first byte of the instruction specified the instruction to be performed and the last 2 bytes specified the address as follows:



Generally, the first byte of the instruction is known as the "op-code" (short for operation code) and the other bytes are known quite naturally as the "address." So a multiple byte instruction may be depicted as follows:



Multiple byte instructions are not restricted to 3 bytes. Sometimes a 2-byte instruction is adequate, and occasionally 4 bytes are used.

INSTRUCTION TYPES

There are 4 basic types of instruction that are used in microprocessors; these are:

(1) Arithmetic and logical instructions – such as add, subtract, shift, AND, OR, etc.

(2) Instructions that work with memory. This can be data memory or program memory.

(3) Jump instructions. These make it possible to jump from one portion of a program to another.

(4) Input and output instructions. These make it possible for the system to communicate with the outside world.

It is quite common for a single instruction to have several of the properties listed, but it is convenient to separate them as above for explanatory purposes.

ARITHMETIC AND LOGICAL INSTRUCTIONS

These instructions operate upon data using the Arithmetic and Logical Unit (ALU) within the MPU. The ALU is a circuit which allows binary numbers to be added, subtracted, shifted, AND-ed, etc. For operations requiring two operands (e.g., addition), the accumulator is often the source for one of the operands and some other memory location or register is the source for the other. One or more flip-flops (often called flags) are associated with the ALU. These flip-flops relate to the overflow and underflow bits occuring during an ALU operation. Where only one flip-flop is used it may be considered as an extra bit added to the ALU, so that if an overflow takes place during an addition, then the flip-flop (often called carry-link flag) is set. In shift operations the operand can be shifted through the carry-link flag. It is important, when carrying out arithmetic and logical operations, to note whether the carry-link flag affects the outcome of the operation and also whether the carry-link flag is affected by the operation. This varies from processor to processor and is usually specified in the detailed explanation of each instruction.

Some MPU's have several flags (flip-flops) associated with the ALU. The various conditions that they can signify include:

Carry – from an arithmetic operation
Overflow – overflow as a result of an arithmetic operation
Link – used in shifting
Sign – usually the sign of the number in the accumulator
Parity – a parity bit relating to the number in the accumulator
Auxiliary – signifies a carry from bit 3 to bit 4 of the adder;

(or Half carry) – this is useful in BCD arithmetic operation

A typical set of arithmetic instructions would include some of the following:

Add
Subtract
Shift left (i.e., multiply the number by 2)
Shift right (i.e., divide the number by 2)
AND
OR
Exclusive-OR
Complement
Clear accumulator (i.e., set accumulator to zero)
Increment
Decrement
Compare
Various instructions for manipulating the flags associated with the ALU

MEMORY REFERENCE INSTRUCTIONS

A memory reference instruction is one which works with data that is stored in either the program memory or the data memory. The instruction "store accumulator contents at the address 1010101010101010," is a memory reference instruction because it works with a binary number which is stored at the specified address. Very often a memory reference instruction includes an arithmetic or logical command. Examples of this are:

(1) Add contents of specified memory address to accumulator

(2) Subtract contents of specified memory address from accumulator

(3) AND contents of specified memory address with accumulator

(4) OR contents of specified memory address with accumulator

(5) Exclusive-OR contents of specified memory address with accumulator

(6) Shift contents of specified memory address one place to the left

(7) Shift contents of specified memory address one place to the right

(8) Increment contents of specified memory address

(9) Decrement contents of specified memory address.

It should be noted that instructions 1 to 5 given above have two operands. One is held in the accumulator and the other is in the specified memory address. The final result is placed in the accumulator. However, instructions 6 to 9 begin with a single operand in a specified memory address. The operand is brought to the ALU in the microprocessor, and after performing the required task the result is placed back into the specified address. The accumulator is not affected by the operation. The above list is by no means exhaustive; some microprocessors do not have all the features given above, others have far more. Memory reference instructions are discussed in more detail in the next chapter.

JUMP INSTRUCTIONS

A jump instruction (sometimes called a branch instruction) is one which allows the microprocessor to move from one section of a program to another, without having to sequentially count through all the instructions before getting to the program segment of interest. Suppose a program contains three jobs, A, B and C, which the system has to do and these three jobs are written as three program segments one after the other as in Figure 1-4a. If for some reason it is required to reverse the order in which the jobs are carried out, this can be done by using jump instructions inserted as shown in Figure 1-4b. To carry out a jump instruction the program counter is loaded with the new value specified in the instruction. So in the example of Figure 1-4b, the program counter begins at zero, and the first instruction is a 3-byte jump instruction which commands a jump to program memory address number 43. The program then executes the instruction steps through instruction 44, 45, etc. until it reaches the instruction which is stored in program memory bytes 59, 60 and 61, which commands a jump to the instruction stored at memory location

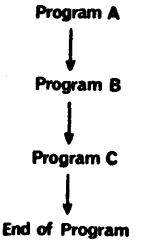


Figure 1-4a. Basic program layout

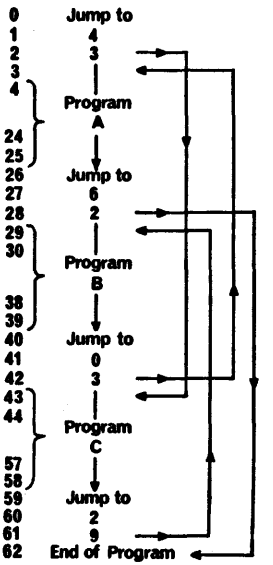


Figure 1-4b. Program flow with jump instructions

Figure 1-4. Illustration of action caused by jump instructions

29. This causes the program counter tó be loaded with 29 and program B will then be carried out. Then program A is followed and the final jump is to the program end.

The type of jump instruction described above is known as an unconditional jump because the jump always takes place. A great deal of the power of a computer stems from its ability to make conditional jumps.

A typical conditional jump instruction would be the following:

"If condition X is true, then jump to a specified instruction number, but if X is not true, continue to the next instruction in sequence." The contents of a program counter, after a conditional jump instruction has been executed, can either be the next instruction in the sequence or a completely new instruction address specified by the conditional jump. An example might be:

> If accumulator is zero, jump to address 1010101010101010 (if accumulator is not zero, go to the next instruction in sequence).

Normally the section in brackets is understood and it is not usually written as part of the instruction definition. Simple microprocessors have a limited range of jump instructions. These might be "Jump if carry-link = 1" and "Jump if accumulator is zero." Other units may have more jump conditions based on the condition of the ALU flags. An example of such a conditional jump would be "Jump if overflow is set." In some microprocessors the user defines the various jump instructions by means of hardware.

INPUT-OUTPUT INSTRUCTIONS

A microcomputer system is of little use by itself because there must be some way in which the user can put information in and get information out of the system. The communication channels which are used for input and output are usually called "ports," and all microprocessors have some instructions which make it possible to take information into the system via an input port and put information out via an output port.

Some microprocessors have special input and output instructions but others adapt normal memory reference instructions for the purpose. This subject is covered in Chapter 3.

SUMMARY OF INSTRUCTION TYPES

The above breakdown of instructions into 4 types is a nominal one because, in practice, many instructions fall into 2 or more of the categories (e.g., a jump instruction implies a memory reference operation). However, most manufacturers try to split up the instruction set in the above manner so as to aid the understanding of the set of instructions that a particular microprocessor will respond to. Inevitably a fifth category of "special instructions" tends to creep in, although with such a loose set of groupings to

begin with, it is difficult to decide whether an instruction is special or normal. Some typical special instructions include things like "Stop," "Delay for n seconds," "Do-nothing," etc. The following chapters will cover the various instruction types in more detail.

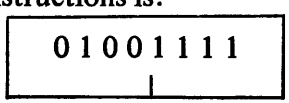
EXAMPLE OF A MICROCOMPUTER PROGRAM

The example here illustrates how a typical microcomputer might add two numbers stored in memory locations 0000:0000:1000:0000 and 0000:0000:1000:0001 and place the result in memory location 0000:0000:1000:0010. A basic program is as follows:

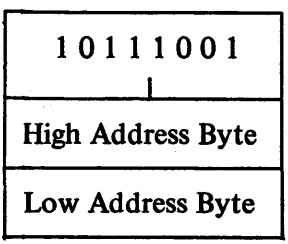
Step 0 Clear accumulator
Step 1 Add contents of 0000:0000:1000:0000 to accumulator
Step 2 Add contents of 0000:0000:1000:0001 to accumulator
Step 3 Store accumulator contents at address 0000:0000:1000:0010

The instruction code for each of the three instructions is:

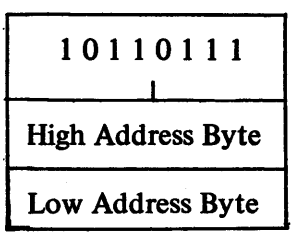
Clear accumulator



Add contents of memory to accumulator



Store accumulator contents at specified address



The program memory would then contain the following bit pattern:

Step Number	Program Memory Address	Contents
0	0	0 1 0 0 1 1 1 1
1	1	1 0 1 1 1 0 0 1
	2	0 0 0 0 0 0 0 0
	3	1 0 0 0 0 0 0 0
2	4	1 0 1 1 1 0 0 1
	5	0 0 0 0 0 0 0 0
	6	1 0 0 0 0 0 0 1
3	7	1 0 1 1 0 1 1 1
	8	0 0 0 0 0 0 0 0
	9	1 0 0 0 0 0 1 0

When the microcomputer begins the program, the program counter will be set to zero. The program counter contents will be sent out as an address (i.e., address 0) and the corresponding memory contents (01001111) will be returned to the instruction register. The MPU increments the program counter and then executes the instruction which is "Clear accumulator." The next instruction is a 3-byte instruction so that it will not be executed until the last byte has been received into the instruction register whereupon it will add the contents of the specified memory address to the accumulator.

This will be repeated for Step 2, but with a different address, and finally on Step 3 the contents of the accumulator will be sent to the specified address. Step 3 is also a 3-byte instruction, so execution of the instruction will not take place until the last byte has been received.

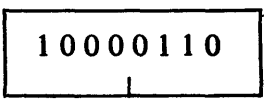
It should be noted that in order to encode four instructions into the program memory it has been necessary to use ten bytes of program memory. Different microprocessor designs try to reduce the program memory size for a given job by using variations on the basic instructions and architecture. Notice also that program step 2 begins at memory address 4, so that should it be necessary at any stage to jump to program step 2, the jump instruction should read "Jump to program memory location number 4."

USE OF THE INDEX REGISTER

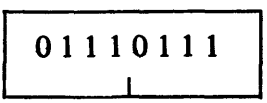
For the above example it is possible to reduce the number of bytes of program memory required by using the index register. Note that the three data-memory addresses are consecutive; this is very common in computer programming. Therefore, if the index register is loaded with the address of the first memory location, there is no necessity to send the address to the MPU again.

Thus the instruction, "Add contents of memory location 0000:0000:1000:0000 to accumulator," becomes "Add contents of memory location specified by the index register." The second instruction can be encoded in one 8-bit byte because it does not specify a 16-bit address. Thus it becomes:

"Add contents of memory location specified by index register to accumulator"

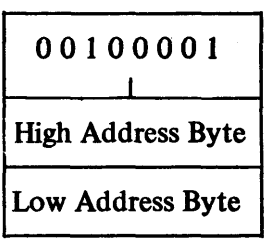


Similarly, the store instruction can be written: "Store contents of accumulator at memory location specified by index register".

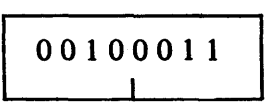


To assist in modifying the program, it is necessary to define two other instructions.

"Load index register with a specified 16-bit binary number"



"Increment index register"



The program now becomes

Step 0 Clear accumulator

Step 1 Load index register with address 0000:0000:1000:0000

Step 2 Add contents of memory location specified by index register to accumulator

Step 3 Increment index register

Step 4 Add contents of memory location specified by index register to accumulator

Step 5 Increment index register

Step 6 Store contents of accumulator at address specified by index register.

The program memory would then contain the following bit pattern:

Step Number	Program Memory Address	Contents
0	0	0 1 0 0 1 1 1 1
1	1	0 0 1 0 0 0 0 1
	2	0 0 0 0 0 0 0 0
	3	1 0 0 0 0 0 0 0
2	4	1 0 0 0 0 1 1 0
3	5	0 0 1 0 0 0 1 1
4	6	1 0 0 0 0 1 1 0
5	7	0 0 1 0 0 0 1 1
6	8	0 1 1 1 0 1 1 1

The necessary data memory addresses are held in the index register and, because the required addresses are consecutive, the next address can be created by incrementing the index register. This saves time and storage because it is not necessary to send a new address each time a memory reference instruction is used. For the example used here the saving is small (just one memory byte), but computer programs often require a sequential scan through large blocks of data and then the saving in the program memory size can be quite significant.

INSTRUCTION NOTATION METHODS

Even with the simple examples used already, there has been some textual difficulty in writing the programs. The description of each instruction is rather long and writing of binary numbers is tedious and prone to error. Therefore, it is convenient to adopt shorthand schemes to assist in presenting the information in a precise and concise manner. As an example consider the instruction:

"Add contents of memory location 0000:0000:1000:0001 to the accumulator."

This can be shortened to:

ADA [0081]

ADA is a mnemonic for "Add to accumulator." The square brackets signify "the contents of the enclosed address," and 0081 is the address written in hexadecimal format.

Each microprocessor has its own unique set of instructions and its own set of mnemonics. There is no international standardisation. Manufacturers usually provide detailed lists of the mnemonics, their formats and their meanings.

Fortunately, there are relatively few shorthand methods of writing binary numbers and there is a definite trend towards the hexadecimal method. With this method the binary number is split into blocks of 4 bits and then each block is written down as its equivalent decimal number. For binary numbers in the range 1010 (ten) to 1111 (fifteen), the first few letters of the alphabet are used. The complete conversion table is as follows:

TABLE 1-2. BINARY TO HEXADECIMAL CONVERSION TABLE

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

An alternative method is to split the binary number into blocks of 3 bits and then write down the equivalent decimal number. Since the largest number using 3 bits is 7, this is known as the octal representation.

e.g. 1010101010101010 becomes 1010:1010:1010:1010 = AAAA in hexadecimal

and becomes 1:010:101:010:101:010 = 125252 in octal code.

It should be understood that the various mnemonics and number codes are purely to assist the user in writing and understanding the program. In the program memory all instructions are stored as binary numbers. Also note that there are two shorthand methods of writing the instruction. The first is a letter-type mnemonic giving the action of the instruction (e.g., ADA), and the second method is the hexadecimal code for the binary form of the instruction.

e.g., ADA [0081] has the binary code

1011:1001 ← Op. code

0000:0000
1000:0001 } Address

which may be written B9
00
81 in hexadecimal code.

Programs which are written either in pure binary code or in hexadecimal (or octal code) are generally referred to as being written in "machine code." Programs written using alphabetic mnemonics are usually called "assembly language programs."

SUMMARY

The function of this chapter has been to explain some of the more fundamental microprocessor principles. It should be remembered that no two microprocessors are the same, and the examples given above are intended purely to illustrate certain points. They do not refer to any particular system and in practice are subject to various refinements.