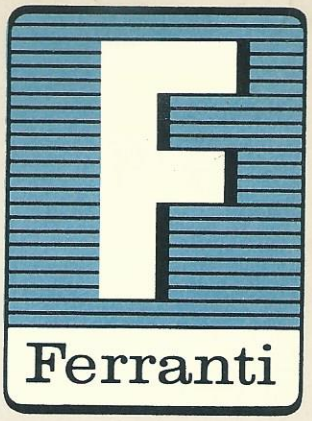
FP6000
computer system
PROGRAMMING MANUAL



Ferranti ELECTRONICS

A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED

INDUSTRY STREET, TORONTO 15 - ONTARIO - CANADA

MANUAL Q.144

FP6000 COMPUTER SYSTEM

PROGRAMMING MANUAL

Ferranti ELECTRONICS
A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED
Industry Street, Toronto 15, Ontario, Canada

FP6000 PROGRAMMING MANUAL

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 COMPUTERS

This book is concerned with automatic electronic digital computers. A digital computer is a machine which can perform arithmetical operations on numbers represented in digital form. This way of representing numbers is the one with which we are most familiar, since it is in everyday use. For example, the number 53 might be represented (in a mechanical digital machine) by two gear wheels, each with 10 teeth, one turned through 5 teeth and the other 3 teeth, relative to some standard position. In an electronic machine this number might be represented by two trains of pulses containing 5 and 3 pulses respectively. The ordinary way of writing numbers is digital.

By contrast, in analog computers numbers are represented by some physical quantity, such as length, angle or electrical potential. In an analog machine arithmetical operations are performed by using some law of Physics, eg., Ohm's law, and then making a measurement to find the answer. The most familiar analog computing device is a slide rule, which uses lengths to represent numbers (the lengths are proportional to the logarithms of the numbers they represent); these lengths are added and subtracted mechanically to give lengths corresponding to products and quotients. In an electronic analog computer numbers are usually represented by electrical potentials and the machine contains circuits for producing potentials proportional to sums, products and so on; these potentials can be measured to provide the results.

The precision of an analog machine is limited by the precision with which the physical quantity used can be measured; it is seldom greater than two or three decimal figures. To increase this precision may be very difficult, and certainly expensive. In a digital machine the precision can be increased as much as desired simply by allowing enough digits in the numbers; this is usually quite easy at the time the design of the machine is being laid down.

Most digital computers use numbers having from 6 to 14 decimal digits; this may seem overgenerous since the raw data of a problem may be given to only three digits and this may be enough in the results. But it should be remembered that a computer may perform thousands of operations before arriving at these results, and rounding errors may therefore build up alarmingly. Further, the starting numbers and the intermediate quantities and results may vary over a very wide range.

1.2 AUTOMATIC DIGITAL COMPUTERS

Automatic digital computers originated in the work of Charles Babbage in the early nineteenth century. Babbage proposed a digital machine capable of doing extended calculations without human intervention. This machine was, of course, to be purely mechanical. Its inventor's ideas seem to have been ahead of the technical possibilities of his day and it was never built.

The first automatic digital computer to be made was the Automatic Sequence Controlled Calculator, an electro-mechanical machine which was not finished until 1944. Since this date the situation has been transformed by the speed, flexibility and reliability of electronics.

An automatic digital computer is a digital machine which can perform a large number of arithmetical operations when once set up. Generally such a machine will have a single Arithmetic Unit which is used repeatedly to do different operations on various numbers. This unit is sometimes called the Mill, a term originated by Babbage. Since there is usually only one mill there must be arrangements for storing numbers, for selecting them and passing them to the mill, and for storing the results produced by the mill (since these may be required again at a later stage).

1.2 continued

An automatic digital computer of this kind performs only one operation at any given time; after completing one operation it proceeds to do another, and in general it will perform a long sequence of operations with great rapidity. The machine must therefore have a Store of some sort in which can be placed Orders or Instructions (the term Command is sometimes used); the Control Unit of the machine extracts these orders one by one from its order-store and obeys them. As a rule a single order will cause the computer to carry out a single operation, eg., adding two numbers together or moving a number from one part of the number-store to another. In most digital computers the orders are expressed in a numerical code and it is important to note that they can therefore be stored in the same store as the numbers; in general a part of this store will hold numbers and another part orders. The store is sometimes referred to as the memory of the machine.

The store of a digital computer must be capable of "remembering" numbers and orders until they are required. It must be able to give up any item of stored information and to record new information in place of the old. The use of electronics in the mill means that a pair of numbers can be added in considerably less than a millisecond (a thousandth of a second, often abbreviated to msec). If this speed is not to be wasted the storage devices must also operate at high speed. These requirements have in the past been a major source of difficulty in the design of computers and developments in storage have as a rule lagged behind those in other parts of the computer.

A computer must be able to communicate with the rest of the world. We must be able to "tell" the machine what operations it is to perform, to supply it with the numbers on which to operate, and to extract from it the results of its work. The machine must therefore have input and output devices, and these must be fast and reliable in operation.

Thus, the main parts of a digital computer are as follows:

(a) a Store for holding numbers and orders,

(b) a Control Unit which can extract orders from the store, interpret them, and direct the operations of the rest of the machine,

(c) a Mill, or Arithmetic Unit, which can operate on numbers from the store and send its results back to the store,

(d) Input equipment (eg., punched card or paper tape reader),

(e) Output equipment (eg., printer or card or tape punch).

These parts and their interconnections are shown in Fig. 1.1. It will be seen that the input and output devices are shown as connected to the mill; this is usually the most convenient arrangement. The connection labelled "discrimination" is described later in the next section.

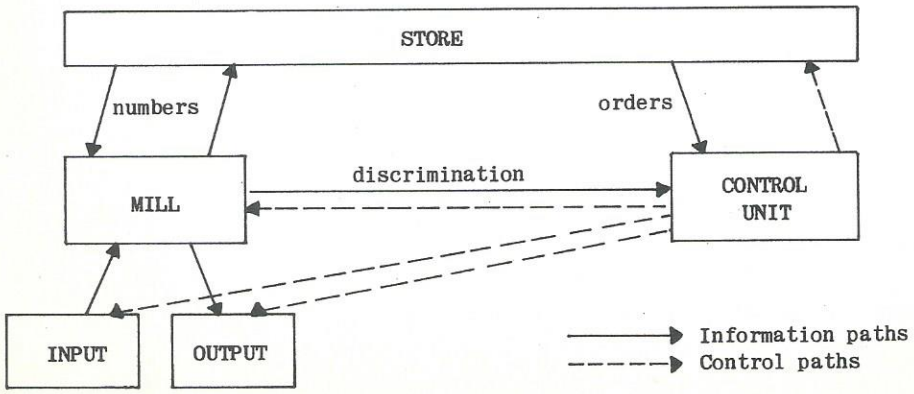


Fig.1.1 Block-diagram of a Digital Computer

1.3 PROGRAMMING

Let us consider how a digital computer is applied to the solution of a problem. The first essential step is to formulate the whole problem precisely and unambiguously; if the problem is a scientific one this formulation should preferably be in mathematical terms, and an equivalent degree of precision should be aimed at in other kinds of problems. This step is usually the hardest part of the whole process and the one where skill and experience are most important; it may well require considerable time and study. We shall not discuss this subject further here since it belongs properly to the field of study which gave rise to the problem.

The next step is the examination of this precise formulation of the problem in order to find the method of solution best adapted to the available computing tools. This step may, in scientific problems, require a knowledge of Numerical Analysis; and it may in any case be combined with the previous step. This subject also is outside the scope of this manual, but there is one point which must be made in this connection. A digital computer can, at least in principle, undertake any task which can be expressed precisely as a finite sequence of finite arithmetical operations. It must be emphasised that it is only problems of this sort which can be handled by a computer; but this is not such a severe restriction as might appear at first sight and there are many non-numerical problems which can be reduced to arithmetic; for example, much work is now being done on the automatic translation of languages by computers. The practical limits on the range of problems which can be dealt with are largely imposed by the storage capacity and speed of the computer and its ancillary equipment.

We shall assume, therefore, that we are presented with a definite numerical process, which we must express in a form assimilable by the computer. That is, we have to program the problem, or prepare the program of orders for the computer. In some circumstances it may be useful to subdivide this process into two stages. In the first stage, programming in a narrower sense of the word, the numerical process is converted into a flow-diagram, or flow-chart, showing what steps the machine is to take and how one operation leads to another.

In the second stage, which is often called Coding, the actual orders which the computer will have to obey are written down in some convenient code; this is relatively simple, once the flow-diagram has been prepared. The exact way in which programming and coding are distinguished (if at all) depends very much on individual preference and no inviolable rules can be laid down - except that great care is needed at all stages. The next step is to punch out the program, order by order, on paper tape or cards and to feed the result into the computer, along with the numerical data (though these may be fed in as a separate operation). At this stage the orders and numbers appearing on the tape or cards are simply read by the computer, converted into its own internal code, and placed in the store. When this input process is complete the store of the machine will contain the whole of the program and some or all of the numerical data; at this point the computer starts to select and obey the orders of the program one by one. Some of these orders will ultimately cause the machine to print or punch the results of the calculation.

It will be clear that the preparation of the program for a particular calculation may well be time-consuming. When it has been written the program will have to be checked and tested on the computer; this stage is usually called the development of the program. When the preparation and development are complete the program may be used repeatedly with different numerical data. This means that it may be uneconomic to write a program for a calculation which has to be done once only; but if a calculation has to be done many times, perhaps at regular intervals, the cost of preparing the program can be distributed and it will usually be found that a digital computer offers by far the cheapest method of doing the work.

The orders which the computer obeys in the course of executing a program are selected from the set of available types of order built into the machine. This set of orders is called the Order-code or Function-code of the computer; it is important for the programmer that a comprehensive set of orders should be available, and that the exact effects of each of them should be known to him.

1.3 continued

It is very helpful if the orders are systematically arranged and are free from objectionable exceptions and omissions. Many of the orders in the order-code of any computer are concerned with simple arithmetical operations; but there are others (for example, those for transferring blocks of numbers from one part of the store to another) which are needed only because the machine is automatic.

The orders in many computers are normally obeyed sequentially; that is to say, they are extracted one after another from adjacent places in the store and are obeyed. Certain orders, called branching orders, may break this regular sequence and cause the machine to start selecting its orders from some other specified place in the store; whether this branch occurs may be conditional on, for example, the sign of some numbers.

In some computers the branch orders are called control-transfer orders since they can be said to transfer control to some other part of the program. Other names are test or discrimination or jump orders. All automatic digital computers have branching orders of some kind and they add enormously to the flexibility of the machine. When a conditional branching order is obeyed certain information is passed from the mill into the control unit; this information is used to determine whether or not the branch occurs (the information passes along the path labelled "discrimination" in Figure 1.1).

Nearly all calculations are, at some stage or other, highly repetitive and the programmer can take advantage of this by writing groups of orders and arranging (with the aid of branching orders) that the machine obeys the orders in each group several times. This is of great importance, as will shortly become evident.

1.4 FLOW DIAGRAMS

We shall now illustrate some of the points mentioned previously with the aid of flow-diagrams such as that shown below. In these diagrams each "box" represents a simple operation or a group of such operations.

(a) Let us first suppose we have in the store of the machine a list of whole numbers, each of which may have any value from 1 to 100, except that the last number in the list is known to be zero. Suppose that the computer is equipped with a printer as an output device and we wish to use this to print all those numbers in the list which are not less than 50.

The flow-diagram of Fig. 1.2 shows a possible sequence of operations. It will be noted that the process consists simply of examining each number in turn; if the number is zero we know we have reached the end of the list, and if it is not we arrange to avoid the "print" operation if the number selected is less than 50. The importance of the test or conditional branch operations is clear.

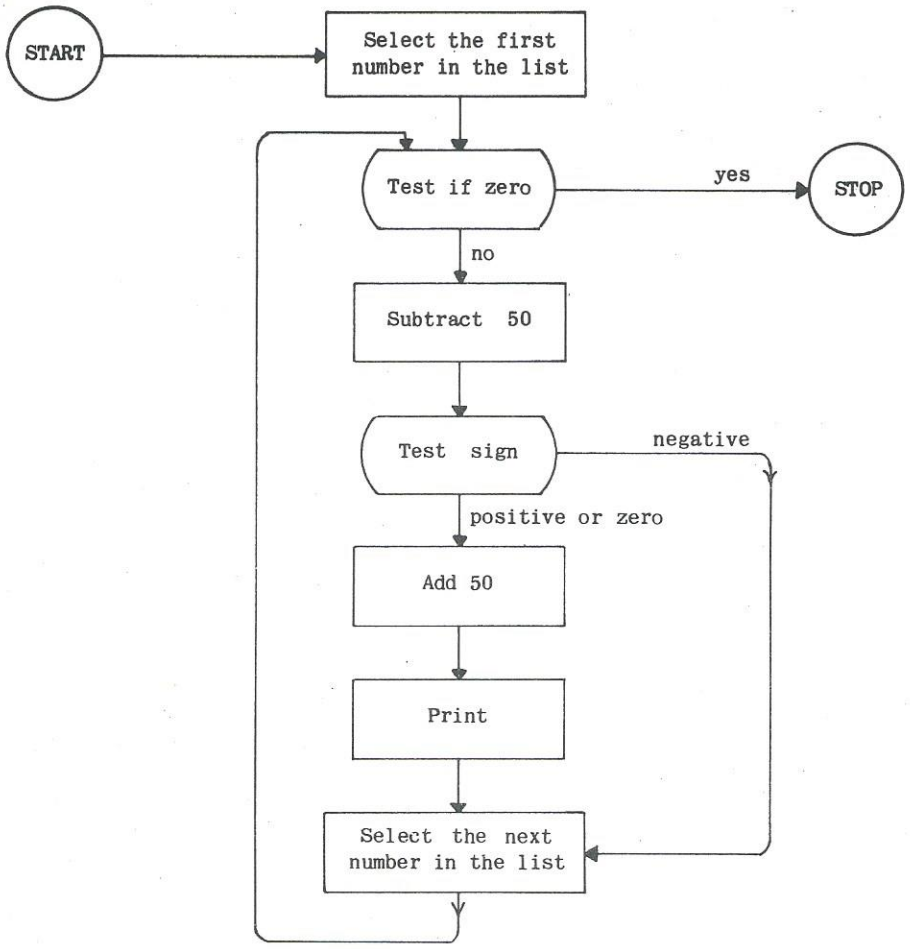


Fig. 1.2 Flow-diagram of a program to print those numbers in a list which are not less than 50.

1.4 continued

(b) In the next example, we shall assume that an input device is provided which can be used to read numbers into the computer one at a time. Suppose that there are 1000 positive numbers waiting to be read and we wish to print the largest of them. It is convenient to use an algebraic notation; the letter x denotes one of the numbers read in from the input device and y stands for the largest number so far read. We read in each number in turn and compare it with y; if it is less than y or equal to it we simply pass on to the next number. If, however, the number is greater than y we must increase the value of y to the new number before reading in the next one. We also have to arrange to count the numbers as they are read in (a quantity c is used for this) so that we know when we have finished, at which point y is to be printed since it is now the largest of all the numbers.

The flow-diagram is shown in Figure 1.3 below.

The technique shown in Figure 1.3 for counting should be particularly noted as it occurs frequently. The main part of the above program consists of a loop or cycle of orders which the computer is to obey exactly 1000 times; after each repetition a counter (in this case c) is reduced by unity until, after the orders of the cycle have been obeyed 1000 times, it is reduced to zero and the computer passes on to the next part of the program. The details of this counting process depend very much on the facilities available in a particular computer; it may, for example, be more convenient on some machines to start with a negative counter and to increase it by unity at each repetition until it is no longer negative.

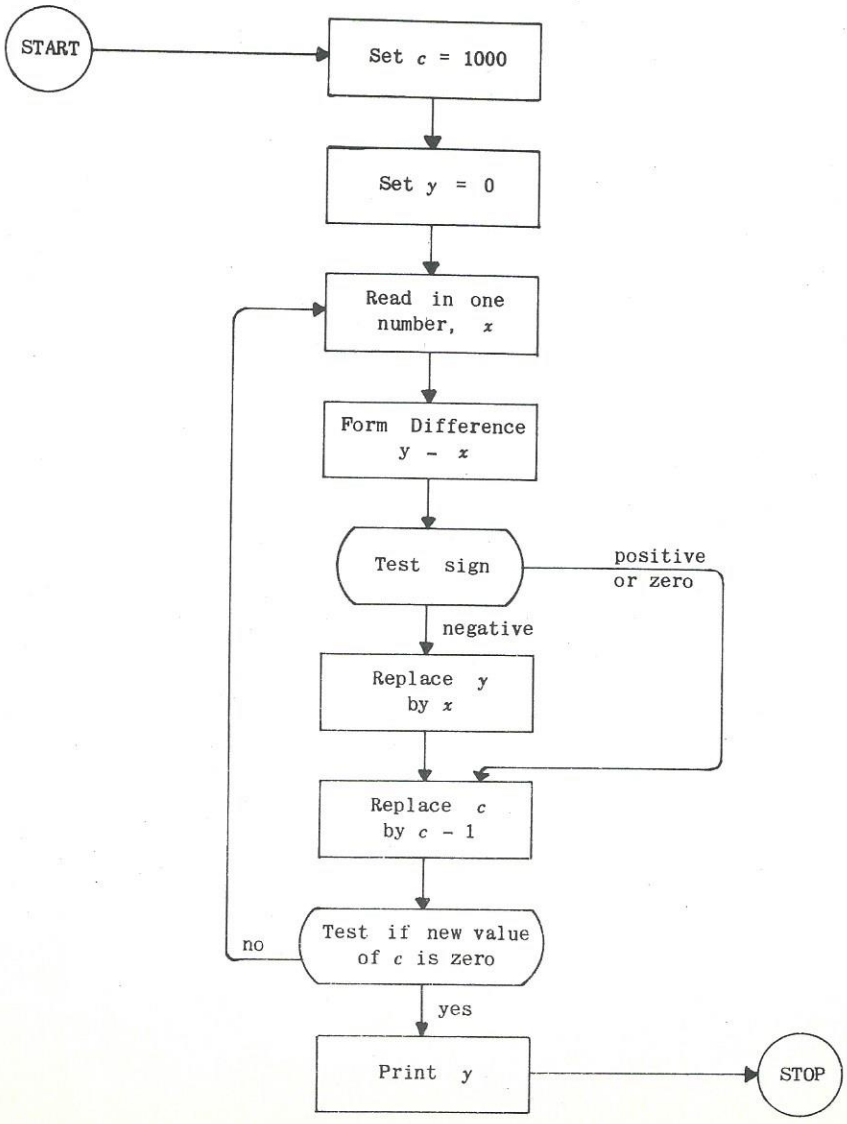


Fig.1.3 Flow-diagram of a program to print the largest of 1000 positive numbers read in one by one.

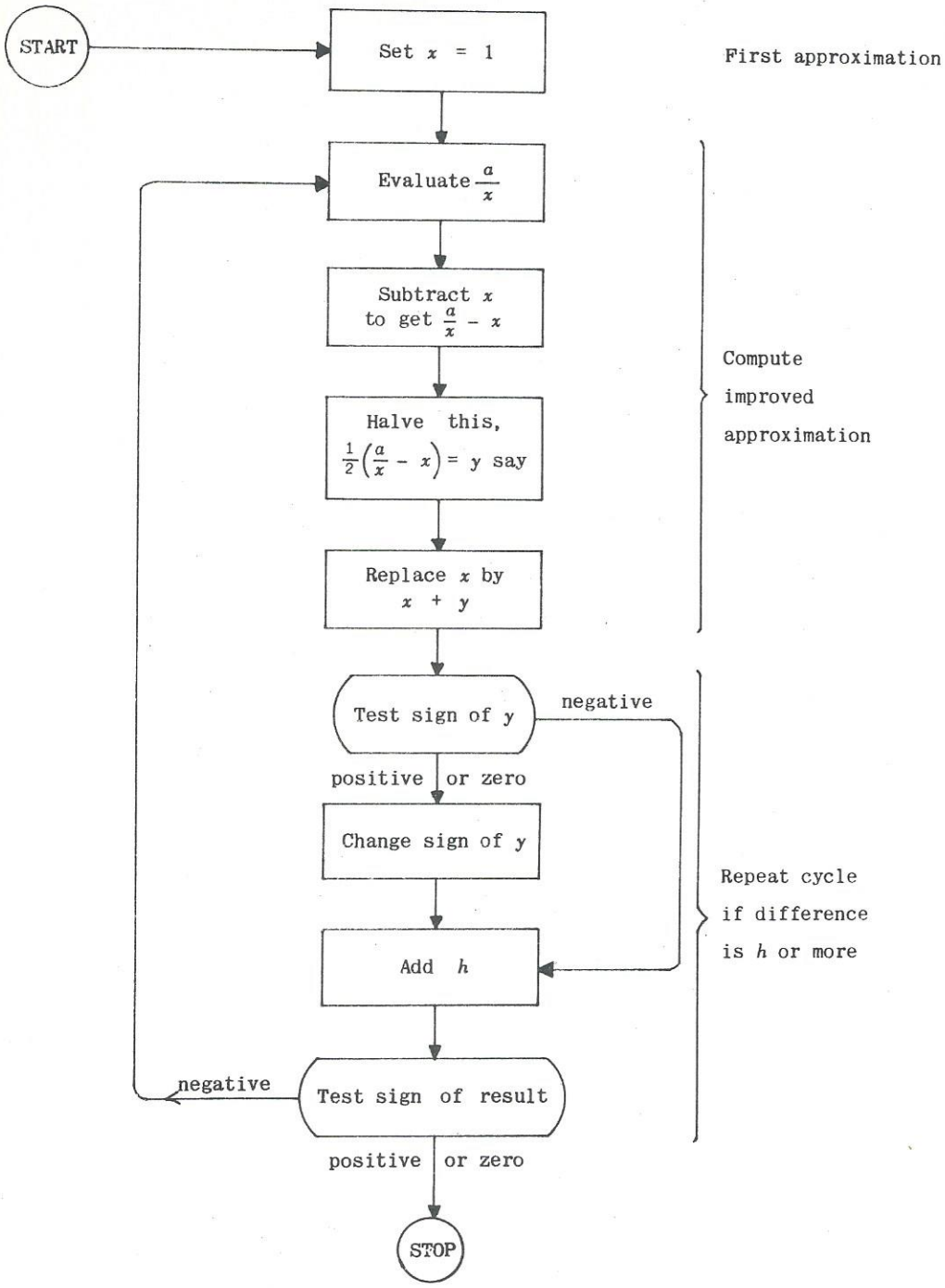


Fig.1.4 A process for evaluating $\sqrt{a}$: the result is x.

1.4 continued

(c) Another illustration is provided by a program to evaluate the square root of a positive number a. We shall use Newton's process; this is based on the fact that if x_1 is an approximation to $\sqrt{a}$, then

$$x_2 = \frac{1}{2}\left(x_1 + \frac{a}{x_1}\right)$$

is a better approximation. This calculation can be repeated with x_2 in place of x_1 to yield a further approximation x_3, and so on. The successive approximations are in general connected by the relation

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

If we are fortunate and if a is a perfect square, we may ultimately arrive at the exact value $x_n = \sqrt{a}$ and the process may be said to have terminated. But as a general rule this will not happen and we shall have to be content with an approximation; for example, we could stop whenever two successive approximations differ by not more than some small preassigned quantity h.

As an illustration let us obtain an approximation to $\sqrt{3}$. We may take $x_1 = 1$ as a first guess; then

$$x_2 = \frac{1}{2}\left(x_1 + \frac{a}{x_1}\right) = \frac{1}{2}\left(1 + \frac{3}{1}\right) = 2,$$

and

$$x_3 = \frac{1}{2}\left(2 + \frac{3}{2}\right) = \frac{7}{4} = 1.75,$$

and

$$x_4 = \frac{1}{2}\left(\frac{7}{4} + \frac{3}{7/4}\right) = \frac{97}{56} = 1.7321428\ldots$$

The correct value to 6 decimal places is 1.732051. It will be seen that this process converges very rapidly - it is in fact an example of what is known as a "second-order" iterative process, in which the number of significant figures is approximately doubled at each step.

Returning to the general process for $\sqrt{a}$, the difference between two successive approximations is

$$x_{n+1} - x_n = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) - x_n = \frac{1}{2}\left(\frac{a}{x_n} - x_n\right)$$

$= y_n$ say.

We can therefore readily compute this quantity from the value of x_n and examine it. If it is greater than h in absolute value we simply add it to x_n to get $x_n + 1$ and then repeat the process; if it is less than h (or equal to it) we have finished.

Any of these flow-diagrams forms a suitable basis for a program or a part of a program. It will be remarked how each process has been broken down into very simple steps and how each step leads unambiguously to the next. Programming has been described as "explaining the problem to the computer in words of one micro-syllable".

The notation used for these flow-diagrams has been chosen to be self-explanatory. If the preparation of flow-diagrams is to be undertaken systematically as a preliminary to the programming or coding of a process it is probably advisable to use a more rigid system to exclude possible ambiguities.

1.5 NUMBERS

The ordinary way of writing numbers may be called a decimal (or denary) system, since it is based on the number 10, which is called the radix of the system. For example the number written 5428 is an integer (or whole number) whose value is

$$5428 = (5 \times 10^3) + (4 \times 10^2) + (2 \times 10^1) + 8,$$

which may alternatively be written,

$$\Big[\Big(\big[(5 \times 10) + 4\big] \times 10\Big) + 2\Big] \times 10 + 8.$$

In this example, 5 is the most-significant (or left-most) digit and 8 is the least significant (or right-most) digit. The contribution made by each digit to the value of the number is just the value of the digit multiplied by a power of 10 determined by the position of the digit in the written form of the number. Fractions and mixed numbers may be expressed by writing digits to the right of a decimal point, for example

$$27.93 = \Big(2 \times 10\Big) + 7 + \Big(9 \times \frac{1}{10}\Big) + \Big(3 \times \frac{1}{10^2}\Big)$$

which may be written

$$(2 \times 10^1) + (7 \times 10^0) + (9 \times 10^{-1}) + (3 \times 10^{-2}).$$

The negative exponent in expressions such as 10^{-3} and 2^{-4} simply means that we have to take the reciprocal. For example 10^{-3} is simply another way of writing $1/10^3$, and $2^{-4} = 1/2^4 = 1/16$.

The decimal point is not usually written in integers, it is understood to lie to the right (eg. 5428 could be written 5428. or 5428.0).

This is not of course the only way of representing numbers. Weights in tons and pounds, Periods of time in hours, minutes and seconds are examples of mixed-radix systems; for example, 3 hours, 24 minutes, 52 seconds represents

$$\big[(3 \times 60) + (2 \times 10 + 4)\big] \times 60 + (5 \times 10 + 2) \text{ seconds.}$$

Various combinations of such radices as 5, 12, 14, 16, 20, 60 are in general use for weights and measures in many parts of the world; such systems are widely understood despite their complexity.

A simple system which finds some application in computers is the Octal system. This is similar to the ordinary decimal system but is based on the radix 8 instead of 10; thus the number written 2736 in the octal system has the following value in the conventional decimal system

$$(2 \times 8^3) + (7 \times 8^2) + (3 \times 8) + 6,$$

$$= (2 \times 512) + (7 \times 64) + (3 \times 8) + 6,$$

$$= 1024 + 448 + 24 + 6,$$

$$= 1502.$$

In the Octal system, only 8 different digits are needed (0, 1, 2, ... 7) instead of the usual 10. Octal fractions may be written by introducing an octal point, for example 3.5 means 3-5/8. This is clearly much simpler than some of the mixed-radix systems with which we are familiar.

The simplest system of all, and one which is used in most digital computers (including FP6000) is the Binary system, which is based on the radix 2. This has the great advantage of needing only two different digits (0 and 1) for writing any number. The number written 1101 in binary has the value of 13 in the decimal system, as shown below.

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2) + 1,$$

$$= 8 + 4 + 0 + 1,$$

$$= 13.$$

We may write fractions and mixed numbers if we introduce a binary point, for example

$$10.01 = (1 \times 2) + 0 + (0 \times 2^{-1}) + (1 \times 2^{-2})$$

$$= (1 \times 2) + 0 + (0 \times 1/2) + (1 \times 1/4)$$

$$= 2 + 1/4 = 2\text{-}1/4$$

It might appear at first sight that any departure from the usual decimal system would introduce many grave disadvantages to the programmer and user of the computer. This is not so however; in fact one need seldom be conscious that the machine is operating in binary. This is because all the numbers fed into the machine and all those which come out of it are in decimal (or in some other convenient system).

1.5 continued

All the necessary conversions to and from the binary system are merely arithmetical operations, which the computer itself does efficiently. When considering the internal operations of the computer it is usually enough to think of them as being carried out simply on "numbers"; one need not usually consider the way in which these numbers are represented inside the machine.

Table 1.1 shows several numbers in binary together with their decimal equivalents. It will be noticed that integers (whole numbers) have more digits when represented in binary than in decimal; as a rough rule 10 binary digits are about equivalent to 3 decimal digits. The word "bit" is often used as an abbreviation of binary digit.

It is, of course, essential for the computer to be able to use negative numbers as well as positive ones. The ways in which negative numbers are represented will be described later.

It should be noted that the octal system described above is closely related to the binary system. In fact, since $8 = 2^3$, a single octal digit is exactly equivalent to three binary digits. An octal number can be converted to binary by simply writing down the binary equivalent to each of its digits, thus 473 in octal is 100 111 011 in binary. Conversely, the binary number, 1011.101, is 13.5 in octal (or 11-5/8 in decimal). Note that one cannot use such a simple process for converting from binary to decimal, or vice versa. It is sometimes more convenient to write binary numbers in octal form since the conversion is easy and the result is more compact and more easily remembered.

While it is as well for the programmer of a binary computer to be familiar with the elements of binary arithmetic, fluency in handling binary numbers is not at all necessary. One point worth noting is that if we move the binary point one place to the right in a binary number we shall effectively double the value of the number, eg.,

10.101 is 2-5/8

and 101.01 is 5-1/4.

In the decimal system such an operation multiplies the number by 10 of course. In the same way if we move the binary point one place to the left we shall halve the value of the number.

Binary	Decimal	Binary	Decimal
1 or 1.0	1	0.1	0.5 = 1/2
10 or 10.00	2	0.01	0.25 = 1/4
11	3	0.11	0.75 = 3/4
100	4	0.001	0.125 = 1/8
101	5	0.101	0.625 = 5/8
1000	8	0.0101	0.3125 = 5/16
1010	10	1.01	1.25 = 1-1/4
10001	17	1.1	1.5 = 1-1/2
11001	25	10.1	2.5 = 2-1/2
100001	33	110.1	6.5 = 6-1/2
1100100	100	101.001	5.125 = 5-1/8

TABLE 1.1 Some Binary Numbers with their Decimal Equivalents

1.6 WORDS

The numbers normally handled in a computing machine contain a certain fixed number of digits. It is convenient to use the term "word" for such a number. A word is a group of digits normally handled together by the computer. Words may be used to represent either numbers or orders, and that is the chief reason for introducing the term. In most computers the words representing numbers and those representing orders are indistinguishable in appearance, each is merely a string of digits. Words of these two kinds are usually stored in the same store, but the programmer often allocates certain parts of the store to numbers and other parts to orders.

In binary computers, words of 18 to 48 or more binary digits are generally used; the word-length is usually fixed by the construction of the machine. FP6000 has a word-length of 24 binary digits, which is equivalent to rather more than 6 decimal digits. The 24 bits are generally handled by the computer in parallel at a clock frequency of one megacycle.

In a parallel computer the time needed for the transfer of a word from one register of the machine to another is called a word-time (the term beat is also used). In FP6000 a word-time is equal to 1 microsecond. The duration of any operation in the computer is a function of the core store speed (2, 4 or 6 usecs) and the clock frequency.

A word may be used to represent a number, in fact this is one of the major uses for words. If the digits of the word are written out side by side in the usual way (with the most-significant digit on the left) we shall get an integer (or whole number). This is one way of interpreting the word, or assigning a value to it, and we shall call it the integer convention. There is an implied binary point just to the right of the least-significant digit.

In FP6000, as in most binary computers, the left-hand (or most-significant) digit in a number-word is used to indicate the sign of the number; it is called the Sign-digit, (or sign-bit) and is 0 in a positive number (or zero) and 1 in a negative number. We shall explain shortly how negative numbers are represented by words.

The digits of a word are numbered from left to right for reference purposes; the sign-digit is digit 0, the digit to the right of it is digit 1, and so on. The last or least significant digit is digit 23. We shall often write B in front of the digit-number so that, for example, B12 means digit or bit 12.

Instead of using the integer convention we could insert a point (a binary point in FP6000 of course) between two specified digits of the word. This point is NOT represented in the word and its position is largely a matter of convention. In FP6000 the binary point is normally placed between B0 and B1, ie., immediately to the right of the sign-digit, so that a typical 24-bit word may be written

0.10001 101111 100100 010100

which is a binary fraction. This way of assigning a value to the word will be called the fractional convention.

The word

0.00000 000000 000000 000000

represents the number zero on either the fractional or the integer convention. The smallest positive number is represented by the word

0.00000 000000 000000 000001

whose value is 1 on the integer convention, and $1/2^{23}$ (or 2^{-23}) on the fractional convention (this is about 0.00000 01192 in decimal). The largest positive number is represented by the word

0.11111 111111 111111 111111

On the fractional convention this has a value just less than unity, in fact $1 - 2^{-23}$ (0.99999 98808 approximately in decimal). On the integer convention it can be shown that its value is $2^{23} - 1 =$ 83 88607, ie., rather more than 8 million. Thus, a word representing a non-negative number can take values from 0 to 83 88607 on the integer convention, or from 0 to $1 - 2^{-23}$ in steps of 2^{-23} on the fractional convention. These numbers are therefore expressed by the equivalent of rather more than 6 decimal digits.

1.6 continued

Let us now consider negative numbers. If we use a desk calculator which handles numbers of 10 decimal digits and we subtract 48 from 0 we shall get the result 99999 99952. This may be regarded as an alternative way of writing the negative number -48. If negative numbers are written in this way they may be added and subtracted correctly on the calculator, provided the results are consistently interpreted. We say that negative numbers written in this way are represented in complementary form. The complementary form of a negative number may be obtained by subtracting the absolute (unsigned) value (in this case 48) from 10^{10}, which is the number lying just outside the capacity of the calculator. The usual way of writing signed numbers is by means of a sign (+ or -) and the modulus of the number.

Either of these ways of representing signed numbers may be used in a digital computer; and other systems are occasionally employed. If a computer uses the sign and modulus representation there may be two different ways of representing zero (ie., as +0 and -0); it is important that these should be treated in the same way.

FP6000 uses the complementary system for negative numbers and we shall now describe this in more detail. The complements are taken with respect to 2^{24}, which plays a role corresponding to 10^{10} in the above example of a decimal desk calculator. Consequently, a negative number which is small in absolute value is represented by a word having a string of 1's on the left (corresponding to the string of 9's above). For example, the integer 44 is represented by the word *

0.00000 000000 000000 101100

We can find the word representing -44 by subtracting the above word from 0. The result is

1.11111 111111 111111 010100

One way of determining the value of a negative number-word is to subtract it from 0 and evaluate the resulting positive word according to the usual rules. The process of subtracting a number from 0 is usually called negating the number, or changing its sign. It should be particularly noted that this is not done by simply changing the sign-digit (as it would be if the sign and modulus representation were used). An easy way of changing the sign of a given word on paper is to start at its least significant digit (on the right) and, proceeding to the left, to copy all the 0's (if any) until we find a 1; this 1 is also copied. The remaining digits are then reversed, ie., we write 1 for 0, and 0 for 1. Tables 1.2 below and 1.3 overleaf illustrate how numbers are represented by words in the two important conventions.

* We shall adhere to the convention of writing the point in number-words and grouping the digits in the way shown, regardless of the convention used to assign a value to the word.

Fraction	Word Representing the Fraction
1/2	0.10000 000000 000000 000000
$1/8 = 2^{-3}$	0.00100 000000 000000 000000
- 1/8	1.11100 000000 000000 000000
$\epsilon = 2^{-23}$	0.00000 000000 000000 000001
$13\,\epsilon = 13 \times 2^{-23}$	0.00000 000000 000000 001101
$-13\,\epsilon$	1.11111 111111 111111 110011
3/4	0.11000 000000 000000 000000
- 3/4	1.01000 000000 000000 000000
- 1.0	1.00000 000000 000000 000000

TABLE 1.2 24-bit Words Representing Numbers (Fractional Conversion)

Integer	Word Representing the Integer
1	0.00000 000000 000000 000001
13	0.00000 000000 000000 001101
- 1	1.11111 111111 111111 111111
- 13	1.11111 111111 111111 110011
$128 = 2^7$	0.00000 000000 000010 000000
- 128	1.11111 111111 111110 000000
$2^{22} = 41\ 94304$	0.10000 000000 000000 000000

TABLE 1.3 24-bit Words Representing Numbers (Integer Convention)

1.6 continued

It will be seen from these tables that the word whose value is 13 according to the integer convention has the value 13×2^{-23} according to the fractional convention. The small quantity 2^{-23} occurs frequently and we shall often denote it by ϵ (the Greek letter epsilon), thus 13×2^{-23} may be more conveniently written $13\,\epsilon$. We shall refer to number-words as integers or fractions according to the convention used to interpret them. Suppose that x_F is the value of some word interpreted as a fraction, and that x_I is the value of the same word as an integer; these are connected by the equations

$$x_I = 2^{23} x_F, \quad x_F = x_I/2^{23} = x_I$$

It is important to realise that x_I and x_F have the same digits in binary, but that their decimal representations would be quite dissimilar.

A word representing a negative fraction may take values from -1.0 to $-\epsilon$ in steps of ϵ. It follows that a number x can be represented on the fractional convention only if

$$-1 \leqslant x \leqslant 1 - \epsilon$$

ie., x must be numerically less than unity, except that the value $x = -1$ is allowed. A word may represent an integer n provided

$$-2^{23} \leqslant n \leqslant 2^{23} - 1.$$

ie., $-83\ 88608 \leqslant n \leqslant 83\ 88607$

A fraction x can be represented exactly by a word only if it is an integral multiple of $\epsilon = 2^{-23}$; in general we shall have to approximate. Any fraction can be represented with an error of at most $\pm 1/2\,\epsilon$ by correct rounding of the last digit. Table 1.4 below shows a few such approximations.

Fraction	Approximate Representation by a 24-bit word
2/3	0.10101 010101 010101 010101
1/3	0.01010 101010 101010 101010
1/5	0.00110 011001 100110 011010
1/7	0.00100 100100 100100 100101
1/10	0.00011 001100 110011 001101

TABLE 1.4 Rounded Approximations to Fractions

1.6 continued

If the numbers occurring in a calculation are integers or fractions, they would normally be represented according to the appropriate convention. Otherwise they must be scaled in some way. This scaling must be done in such a way that all the intermediate quantities formed in the machine during the course of the program are within range; and, if fractions are used, the scaling must be such that accuracy is not lost. This is usually possible; but when it is not, or when it is very difficult to determine the scaling factors, then there are well-established programming techniques, such as floating-point working, which can be used. Occasionally we may want more precision than can be obtained by the normal representation of numbers, ie., we may need more than 6 decimal digits; we can then use double-length (or double-precision) arithmetic, in which each number is represented by two words. These and other techniques will be described later.

If we say that a word has the value 0.75 then we obviously mean that the word is to be interpreted on the fractional convention; and if we say its value is 94 we intend the integer convention to be used. As a rule the fractional convention is regarded as the standard one.

The word whose value is -1 according to the fractional convention will usually be written -1.0 to prevent confusion with the integer -1; this word has a 1 in the sign digit position (B0) only, and has the value -2^{23} on the integer convention.

Nearly all of this Chapter applies, with only small changes, to other binary computers using the complementary representation (as most of them do). The fractional range $-1 \leqslant x \leqslant 1$ applies to most computers, though other ranges are occasionally used. The integer range depends, of course, on the word-length.

CHAPTER 2

INSTRUCTION CODE

CHAPTER 2

TABLE OF CONTENTS

Table of Contents (continued)

2.1 BASIC DESCRIPTION OF FP6000

2.1.1 Preliminary Details

The central computer consists of a basic arithmetic and control unit to which is added at least one block of 4096 words of core store and some input/output equipment to make up a minimum system. The core store has a cycle time of 2 or 6 microseconds and a capacity of from 4096 to 32,768 words, each of 24 bits. The store is not divided into blocks or sections in any way. The program's core store locations are numbered from L0 onwards. The first eight of these locations are used as the program's accumulators and are also designated X0-X7; they may be used for arithmetic and counting. Accumulators X1-X3 are also available for indexing (sometimes also called modifying). Locations L8-L19 are reserved for special purposes and their use should be avoided by the programmer. These letters L and X are used for convenience in description and are not required to be written in instructions. An automatic parity check is carried out on each word read from or written to the core store. In operation the control unit of the computer selects the instruction to be obeyed from its location in the core store; the address of the location concerned is called the instruction number and this is held in a special instruction number register, INR.

Instructions are normally obeyed consecutively from the core store except that certain instructions, called branch instructions, may interrupt this regular sequential selection and cause the computer to start obeying instructions from some other specified location.

2.1.2 Numbers Represented as Binary Words

The 24 bits of an FP6000 word are numbered from B0 to B23 starting at the left hand (most significant) end of the word. The binary content of the word has several basic interpretations according to the context. These are:

(a) Four 6-bit alphanumeric characters:
The one at the left, which is numbered zero, is the most significant. The notation used is 123.0 for character 0 of the word in L123, 123.1 for character 1 and so on.

The four characters of N are:

	n_0	n_1	n_2	n_3
B	0-5	6-11	12-17	18-23

(b) A signed fraction:
The word is regarded as having a binary point between B0 and B1; a one in the B1 position contributes 1/2 (or 2^{-1}) to the value of the number, B2 contributes 2^{-2}, and so on. B0 in such a word is used to indicate the sign of the number, it is 0 in a positive (or zero) number and 1 in a negative one. Negative numbers are represented in what is known as two's complement form and hence the sign bit can be considered as an integral part of the number; B0 contributes -1.0 to the value of the number. Hence while

0.010000-----0 represents +0.25

1.110000-----0 represents -0.25
(-1.0 + 0.5 + 0.25)

One method of changing the sign of a binary number on paper is to start from B23 and copy all the bits until the least significant 1 bit is reached; then all the subsequent bits are reversed. In the fractional convention a number can only lie in the range $-1 \leqslant \text{fraction} < 1$ and to bring numbers within this range scaling factors may need to be used.

(c) A signed integer:
This has an implied binary point at the end of the word, to the right of the B23 position, hence with a positive number, B23 contributes 1 to the value of the number, B22 contributes 2, B21 contributes 4, and so on. B0 contributes -2^{23}. In the integer convention a number can only lie in the range $-2^{23} \leqslant \text{integer} < 2^{23} - 1$. In other respects the integer and fractional representations are similar.

Two adjacent FP6000 words can be used to represent a single number, this is double length working. The numbers normally handled in FP6000 have 24 bits which may be considered as 23 bits plus a sign bit. This corresponds to rather more than 6 decimal digits in either fractional or integer representation because $2^{-23} = 0.119 \times 10^{-6}$ or $2^{23} - 1 = 8,388,607$. Occasionally the need arises to work to a higher degree of precision; this is obtained by working double length, which gives a precision of more than 13 decimal digits. A double length number has 46 bits plus sign. The sign bit and the first 23 following bits comprise the more significant word; a zero sign bit and the remaining 23 bits comprise the less significant word. Thus the sign bit of the less significant half is unused and, by convention, must be zero. This procedure can be generalized to multiple length numbers consisting of more than two words.

2.1.2 continued

Certain operations deal directly with double length quantities contained in adjacent accumulators, with the more significant half in the lower numbered accumulator. X7 and X0 are considered adjacent in this context. The notation x: is used to represent the double length word in X and X + 1.

2.1.3 Instructions;
Format and Binary Representation

A single FP6000 instruction consists of four parts,

F X M N

of which a typical example is:-

001 7 3 169

Here the function (F) of the instruction, which specifies which operation the instruction performs, is 001, which, as will be seen later, calls for addition.

The second part (X) of the instruction is called the X-address. This normally specifies an accumulator which contains one of the operands which the instruction uses, in this case X7.

The M-address specifies an accumulator whose contents are used to index N before the instruction is obeyed. The process of indexing will be described in Section 2.10. If M = 0 no indexing takes place.

The fourth part (N) of the instruction is the N-address. This is normally the address of any storage location, which contains the other operand on which the instruction acts, in this case L169.

Hence the basic overall effect of the instruction

001 7 0 169

is to take the word in location 169 and add it to the word in accumulator 7 leaving the result in X7 without in any way affecting the word in L169. After obeying this instruction, the computer will proceed to the next instruction. A convenient way of writing the effect of the various instructions is to use the notation n, x and m to represent the contents of the words whose addresses are N, X and M respectively. Thus the 001 function can basically be defined as

001 $x + n \longrightarrow x$

The arrow means 'replace', thus the symbols to the left of this represent contents of locations before the instruction is obeyed and to the right, contents afterwards. Only the quantity after the arrow is altered by the operation of the function.

A number of other symbols are used in the descriptions of the instructions:

x^* denotes the contents of X + 1

x: denotes a double length number

c denotes the contents of the carry register, C

x_a denotes bits B12 to B23 (the N-address)

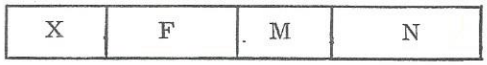
x_c denotes bits B0 to B8

x_e denotes bits B15 to B23 (the floating-point exponent)

x_m denotes bits B9 to B23

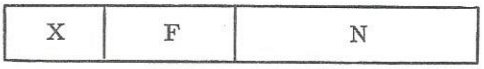
(In general the subscripts are equally applicable to x, m or n)

The usual representation of an instruction in the store in binary is:



B 0-2 3-9 10-11 12-23

except that for a branch instruction this becomes



B 0-2 3-8 9-23

F is written as three octal characters of which the first is constrained to be 0 or 1; F normally comprises B3 to B9 inclusive. Branch instructions use B9 as an N-address bit, hence these functions are represented by B3 to B8 only.

X comprises B0 to B2 which enable any one of the 8 accumulators to be specified.

M comprises B10 to B11 except in branch instructions, which cannot contain M.

N comprises B12 to B23 normally and B9 to B23 in branch instructions and is written in decimal on the coding sheet.

2.1.4 Symbolic Address

While a program is being written it is difficult to allocate actual store addresses to all the quantities that are being used, so that it is a considerable hardship to have a programming notation which demands that only absolute addresses be written in instructions. ASSEMBLER is a program which is used to translate programs as written by the programmer, into a form which is convenient for future reading by FP6000. One facility of ASSEMBLER that is of great value in this context is the ability to refer to the N-addresses of instructions or constants by code-words. Absolute addresses need only be associated with these code-words when the whole program has been written.

Suppose that L169 and X7 are being used in a program to contain a rounding factor and the number to be rounded respectively, and that the instruction:

001 7 0 169

(which was discussed in 2.1.3) is being used to add the rounding to the number, then this instruction can be written with a symbolic address as:

001 7 0 RDG

Ultimately it will be necessary to define an actual address to be associated with RDG but this need not be done until all the instructions have been written and it is clear where the variables need to be stored. The form of the definitions, which are read in before the program, is dictated by ASSEMBLER.

ASSEMBLER will accept as an address:-

a Codeword which consists of up to four characters of which the first must be alphabetic.

an Octal Integer which must be proceded by *

a Single Length Decimal Integer

a Combination of the above three types of elements with + and - (eg., SUM +27 - *10)

For example, in the case of the multiplication of two factors the double length result is stored in two consecutive locations by the multiplication instruction. Symbolic addressing allows these locations to be referred to, for example, as

PROD and PROD + 1

or as

RSLT - 1 and RSLT

Thus if RSLT is later defined to be L179 then RSLT -1 would refer to L178.

Certain codewords have special preassigned meanings and are not otherwise available. These codewords are listed below and described in greater detail in Chapter 7.

T is the transfer address and in the present context can be regarded as the address of the location containing the instruction in which T appears, hence the address 'T + 1' in one instruction refers to the next instruction in sequence.

C, L, A, S designate the four types of shift, Cyclic, Logical, Arithmetic and Special, that are available with all shift functions.

CP, CR, LP, MT, TP, TR, TY designate the types of peripheral units.

2.2 FUNCTION CODE

2.2.1 Function Groups

The functions in the FP6000 function code fall naturally into groups which are denoted by the first two digits of the function (eg., the 001 function belongs to group 00).

The effect of the functions in the various groups may be summarized as follows:

Group 00	Normal arithmetic operations of addition, subtraction, negation, etc. for single and multiple length numbers. The result of the operation is always left in one of the accumulators.
Group 01	Substantially the same as Group 00, except that the answer is left in a core store location, thus providing 'add to store', 'subtract from store', etc.
Group 02	Logical operations and part-word operations; the result is always left in an accumulator.
Group 03	Similar to Group 02, but the result is left in a core store location.
Group 04	Multiplication, division and conversions between binary and decimal.
Group 05	Simple branching operation.
Group 06	Branching operation involving indexing and counting.
Group 07	Miscellaneous branching instruction used with subroutines.
Group 10	Substantially the same as Group 00, except that the numeric value of a constant is specified instead of an address in the store.
Group 11	Single and double length, left and right shifts, in cyclic, logical and arithmetic forms. Single and double length normalizing functions are also provided.
Group 12	A group of miscellaneous functions providing internal block transfers, facilities for check-sums and other logical operations.
Group 13	The floating-point group, providing facilities for addition, subtraction, multiplication, division, conversions, and square roots.
Group 14	Spare
Group 15 16	A miscellaneous set which provides communication among the master programs, its sub-programs, and the computer operator.
Group 17	Input - Output operations. These all allow for variable length transfers between peripheral devices and the store.

2.3 ARITHMETIC FUNCTIONS

2.3.1 The Overflow Register, V

During arithmetic operations it may happen that numbers are produced which exceed the capacity of the computer word; they are then said to overflow. In fact, overflow can be thought of as a loss of significant bits at the left end of a word. This is usually caused by insufficient scaling down of the numbers concerned in the calculation and is generally a serious matter unless detected; in contrast the loss of bits at the right end of a word is less serious and can easily be compensated for by rounding. When overflow occurs the result obtained is arithmetically incorrect and a special overflow register, V, is provided to give warning of this condition.

For example, if a computer word is regarded as representing a fraction then this must, by the standard fractional convention, satisfy the inequality.

$$-1 \leqslant \text{fraction} < 1.$$

Hence if in this convention 0.45 is added to 0.69 the result will exceed the permitted range and hence an incorrect result will be given in the word but V will be set.

V is a two state device which is normally 'clear' but which will be 'set' when overflow occurs. Once V is set it remains set, regardless of all intervening instructions performed by the computer until cleared by one of certain special instructions being obeyed.

2.3.2 The Carry Register, C

Performance of arithmetic operations by hand often necessitates carry over from one figure to the next most significant figure when adding, or borrow from the next most significant figure when subtracting; similarly carry to or from the next most significant word is required when a computer performs arithmetic operations on multiple length numbers. Facilities in the function code enable this carry to be incorporated automatically and operations on multiple length number to be performed just as easily as on single length ones, and using the same functions.

The functions of the arithmetic groups 00 and 01 form eight complementary pairs in all, one function of each pair has application to operations on single length numbers or on the most significant word of a multiple length number and may set V, while the other has application to the less significant parts of the number and may set C. To obtain the correct carry it is essential that the functions that set C are only used on words which are in standard form with zero sign bits.

Unlike V, C has only a transient setting and its normal state is clear; whenever it is set by one instruction it will be cleared by the completion of the next instruction obeyed unless that instruction itself sets C. The lower-case letter c denotes the contents of C and may be regarded as a word which is zero when C is clear and which is all zeros except for a B23 when C is set.

2.3.3 Elementary Arithmetic. Function Groups 00 and 01

000	$n + c \rightarrow x$	010	$x + c \rightarrow n$
001	$x + n + c \rightarrow x$	011	$n + x + c \rightarrow n$
002	$-n - c \rightarrow x$	012	$-x - c \rightarrow n$
003	$x - n - c \rightarrow x$	013	$n - x - c \rightarrow n$
004	$n + c \rightarrow x$	014	$x + c \rightarrow n$
005	$x + n + c \rightarrow x$	015	$n + x + c \rightarrow n$
006	$-n - c \rightarrow x$	016	$-x - c \rightarrow n$
007	$x - n - c \rightarrow x$	017	$n - x - c \rightarrow n$

The action of the functions of these two groups is readily seen from the above symbolic definitions; for example the 001 function adds to the contents of accumulator X both the carry, c, and the contents of location N.

Functions 000-003 and 010-013 can set V but not C and are used with single length numbers or the most significant word of a multiple length number.

2.3.3 continued

Functions 004-007 and 014-017 can set C but not V and are used with the less significant words of a multiple length number; they always produce a result which has a zero as the sign bit.

As an example, the calculation of (p + q - r) where p, q, r are double length numbers in P, P + 1; Q, Q + 1; R, R + 1 respectively requires;-

004	5	0	P + 1	
000	4	0	P	
005	5	0	Q + 1	may set C
001	4	0	Q	may set V
007	5	0	R + 1	may set C
003	4	0	R	may set V

2.3.4 Arithmetic With Small Integers.
Function Group 10

Most programs contain a number of instructions concerned with the simple arithmetic operations of addition, subtraction and copying, using numbers which are predetermined small constants. The functions of this group closely resemble those of group 00 but the quantity specified in the N address, instead of being the address of an operand, is used as the operand itself. Since the N address occupies 12 bits, any integer up to 4095 is available.

100	N + c → x	104	N + c → x
101	x + N + c → x	105	x + N + c → x
102	- N - c → x	106	- N - c → x
103	x - N - c → x	107	x - N - c → x

The action of the functions of this group is readily seen from the above symbolic definitions; for example the 100 function replaces the contents of accumulator X with the sum of the integer N and the carry, c.

Similar to the corresponding functions of groups 00 and 01, the 104-107 functions are designed only to be used with less significant parts of multiple length words and consequently set C rather than V and always produce a zero sign bit. Functions 100-103 set V but not C.

As an example of the use of these functions, the addition of the integer 29 to the triple length integer in X5, X6, X7 is performed by:

105	7	0	29	may set C
105	6	0	0	may set C
101	5	0	0	may set V

2.4 BRANCH INSTRUCTIONS

Normally instructions are obeyed sequentially by the computer. The branching functions apply certain tests. If the test finds that the appropriate conditions are satisfied, the sequence of instructions is interrupted and a branch occurs to cause the computer to obey instructions starting at some other location. If the test conditions are not met, the next instruction is obeyed in the usual way. For all the branch instructions if a branch takes place, the location of the next instruction is specified in the N address of the branch instruction. In this context ASSEMBLER's T (Transfer Address) codeword is particularly useful, for example 'T-4' refers to the fourth instruction before the current one. It will be noted that all branch functions are even numbered, this is because bits B9 to B23 are used to provide a 15 bit N-address so that any location in the core store can contain the instruction to which the branch takes place.

2.4.1 Branch on State of Accumulator.
Function group 05

These functions all carry out a test on the contents of the accumulator which is specified in the X address. The specific functions are shown in the table below.

It will be noted that these functions occur in pairs of opposites, if the contents of the accumulator are such that an 050 function causes a branch then the corresponding 052 function will not branch.

For example, to replace y in a location symbolically identified by ITEM by its modulus necessitates changing the sign if y is negative; this could be performed by the following sequence. It will be found helpful, when writing programs, to mark in by means of arrows the possible paths through the program, as is shown.

Example:

F	X	M	N	Notes:
000	2	0	ITEM	y to X2
054	2	0	T + 2	branch if y is non-negative
012	2	0	ITEM	replace y by -y if y is negative

2.4.2 Unconditional Branch.
Function 074 with X = 0

074 0 0 N

An 074 instruction with zero in the X address will always branch to the instruction specified in the N address. For example, the instruction

074 0 0 T + 3

will branch to an instruction three words on and hence cause the next two FP6000 words to be omitted from the sequence of instructions obeyed.

2.4.3 Branch on State of Overflow.
Function 074 with X = 1,2,3,4,7

There are five branch functions which test the state of V (2.3.1); they enable any desired action to be initiated should overflow occur.

Two of these functions always clear V irrespective of whether or not a branch occurs, two always leave the state of V unchanged and the fifth always inverts the state of V, ie. if V was set, it clears V and if V was clear it sets V.

The action of these five branch functions is provided by the 074 function with appropriate values in the X address.

Branch on State of Accumulator				
050	X	0	N	Branch to N if contents of accumulator X are zero (x = 0)
052	X	0	N	Branch to N if contents of accumulator X are not zero (x ≠ 0)
054	X	0	N	Branch to N if contents of accumulator X are positive or zero (x ⩾ 0)
056	X	0	N	Branch to N if contents of accumulator X are negative (x < 0)

2.4.3 continued

The specific functions under this heading are listed below:

Branch on State of Overflow				
074	1	0	N	Branch if V set and leave V unaltered
074	2	0	N	Branch if V set and clear V
074	3	0	N	Branch if V clear and leave V unaltered
074	4	0	N	Branch if V clear and clear V
074	7	0	N	Branch if V clear and invert V

2.4.4 Branch on State of Carry.
Function 074 with X = 5, 6

Two complementary branch functions are provided to test the state of C; C is cleared by the performance of either of them. They are obtained by the use of the 074 function with appropriate values in the X address.

The specific functions under this heading are:

Branch on State of Carry				
074	5	0	N	Branch if C set and clear C
074	6	0	N	Branch if C clear and clear C

2.5 LOGICAL OPERATIONS

The functions which perform logical operations are characterised by the fact that the numerical values of the words on which they operate are only of secondary significance and that these words are primarily thought of simply as strings of binary digits. Strictly speaking, the logical shift functions also come into this category, but it is more convenient to describe these under the heading of Shift Functions (q.v.).

Facilities are provided for the performance of the three logical operations: 'And', 'Inclusive Or', and 'Exclusive Or'; each instruction operates on two words and produces one new word as a result. V cannot be set by any of these functions.

2.5.1 The 'And' Operation. Functions 020, 030, 120

It is sometimes necessary to pick out a part of a word for separate treatment or examination as, for example, when conserving space in the computer by packing several small numbers into a single word; the 'And' (or collate) function provides facilities for unpacking such a word into its component parts. The new word produced by collating two words has 1's only in those binary positions where both of the operand words have 1's, elsewhere there are zeros. The collating operation is sometimes called 'logical multiplication' since the result may be obtained by a digit by digit multiplication.

The result of collating

j = 0011

with k = 0101

is h = 0001

A bit of h is a 1 if the corresponding bits of both j and k are 1's, otherwise it is a 0. The collate operation is written j & k = h and is commutative (k & j = j & k) and associative p & (q & r) = (p & q) & r, hence p & q & r has 1's only where p and q and r all have 1's.

Most of the actual applications of the 'And' functions are to extract a sequence of bits from a word; to do this, the other operand word will have a block of consecutive 1's corresponding to the required sequence of bits, and zeros elsewhere. For example: to extract B12 to B20 from a word, we set up an operand word (sometimes called a mask) as follows:-

u = 000000000000 111111111 000

then if v = 101110010100 110001110 000

u & v = 000000000000 110001110 000

In the result the required nine bits of v have been left unaltered and the remaining bits have been replaced by 0's.

The functions 020, 030 and 120, are analogous to the functions of groups 00, 01 and 10 respectively. They are defined as:

020 x & n → x

030 n & x → n

120 x & N → x

Note that in the 120 functions, N is a 12 bit integer and is therefore only useful in extracting data lying within positions B12 to B23.

As an illustration, suppose we wish to skip 7 instructions if the integer (k) in address INT is odd; to do this we examine the last bit of k, this will be 1 only if k is odd.

F	X	M	N	Notes:
000	3	0	INT	k to X3
120	3	0	1	Leave only B23 of k in X3
↓052	3	0	T + 8	Branch if B23 ≠ 0

A further example: suppose we wish to split the word in ITEM into two parts, retaining B0 to B18 in ITEM and storing B19 to B23 in ITEM + 1.

F	X	M	N	Notes:
000	6	0	ITEM	Word to X6
120	6	0	31	B19 to B23 to X6
013	6	0	ITEM	B0 to B18
010	6	0	ITEM + 1	B19 to B23

2.5.2 The 'Inclusive Or' Operation.
Functions 021, 031, 121

The 'Inclusive Or' operation produces a 1 in the result where either (or both) operand has a 1. The result of this operation

on j = 0 0 1 1

and k = 0 1 0 1

is h = 0 1 1 1

The term "or" is represented by the symbol "v" which is derived from the Latin word "vel" meaning "or". Thus, for example, x v n should be interpreted as x or n.

The "Inclusive Or" operation is written j v k = h and is commutative (j v k = k v j) and associative p v (q v r) = (p v q) v r, hence p v q v r has 1's in all positions except those in which p, q and r all have 0's.

The functions 021, 031, 121 are analogous to the functions of groups 00, 01 and 10 respectively: they are defined as

021 x v n → x

031 n v x → n

121 x v N → x

One application of the 'Inclusive Or' function is to set a marker bit in a word, say B20 in X6 irrespective of whether it was already set and without disturbing any other bits in the word. This will be done by the instruction:

121 6 0 8

2.5.3 The 'Exclusive Or' Operation.
Functions 022, 032, 122

The 'Exclusive Or' operation produces a 1 in the result only in positions where the corresponding bits of the operand words differ. The result of this operation

on j = 0 0 1 1

and k = 0 1 0 1

is h = 0 1 1 0

This process is also called the 'not-equivalent' operation, or another alternative is 'logical addition' since the result may be obtained by a bit by bit addition (or subtraction) with no carry. When the bits of k are 0's, the bits of h are the same as those in j, where the bits of k are 1's, the bits of h are the reverse of those of j.

The 'Exclusive Or' operation is written (j ≠ k) = h and is commutative (j ≠ k = k ≠ j) and associative p ≠ (q ≠ r) = (p ≠ q) ≠ r, hence p ≠ q ≠ r has a 1 bit where either all 3 operands have 1's or just one of them has a 1.

The functions 022, 032, 122, are analogous to the functions of groups 00, 01 and 10 respectively; they are defined as:

022 x ≠ n → x

032 n ≠ x → n

122 x ≠ N → x

One of the standard applications of the 'Exclusive Or' operation is the comparison of two words to see whether or not they are equal (j ≠ k = 0), with FP6000 this operation is performed by the 026 function, which is described later. However, there are other applications of the 'Exclusive or' function; for example, to invert the state of a marker bit in a word, say B20 in X6, without disturbing any other bits in the word. This will be done by the instruction,

122 6 0 8

Again the fact that p ≠ (p ≠ q) = q can be utilized to interchange two numbers, p in X7 and q in STR7, with no possibility of setting overflow and without using any working space.

F	X	M	N	Notes:
022	7	0	STR7	p ≠ q to X7
032	7	0	STR7	q ≠ (p ≠ q) = p to STR7
022	7	0	STR7	p ≠ (p ≠ q) = q to X7

2.6 MULTIPLE LENGTH COMPARISON

In the same way that performance of the arithmetic operations on multiple length numbers necessitates incorporation of carry between one word and another, comparison of the relative sizes of multiple length numbers in either binary or character form also required communication between one word and the next. Tests to determine which of two numbers is the greater, or if they are equal can be performed by a subtraction instruction followed by a branch instruction of group 05, but to facilitate multiple length working, the 026, 027 functions are provided.

2.6.1 Test Equality, Function 026

026 X M N

This instruction will leave C set if it is already set and set C if the contents of accumulator X and location N are not identical. The second of these provisions effectively remembers any non-equality until the complete multiple length number has been tested; care must be taken that C is not already set when the first 026 instruction is obeyed. The instructions

```
026    1    0    ITEM
026    2    0    ITEM + 1
074    5    0    T + 7
```

will cause a branch if the double length numbers in X1, X2 and ITEM, ITEM + 1 are not completely identical.

The advantage of using the 026 function rather than a subtraction is that the state of the accumulator X is not altered nor is there any risk of setting V; Also the 074 branch function is faster in operation than one of the 05 group. These considerations apply equally well to testing the equality of two single length numbers and an 026 function is often the best function to use in this context also.

This function attaches no special significance to the sign bit of a word and may equally well be used to test the equality of words of alphanumeric characters.

2.6.2 Test Relative Size. Function 027

027 X M N

This instruction will set C if the sum of c and the contents of location N is greater than the contents of accumulator X. It is designed specifically for use with words of alphanumeric characters, for example in table look-up, and should only be used with numbers in binary representation if all the numbers are positive. The advantages of using the 027 function rather than doing a subtraction are the same as for the 026 function. The 027 function is, in fact, identical to the 007 function except that the value of x is left unchanged.

2.7 SHIFT FUNCTIONS

2.7.1 Types of Binary Shift

The purpose of the shift functions is to take the bits of a binary word and move them to the left or right. Since FP6000 is a binary machine, the shifts correspond to multiplication and division by powers of 2; it is probably in this context that the binary nature of the computer's operation is most apparent. These functions are particularly useful when numbers have to be doubled, halved, or scaled by a power of 2, or when performing logical operations on words whose contents are not single numbers.

Considering the shift operation applied to a decimal number, a shift of 5.73 one place to the left past the fixed decimal point yields 57.3 so that effectively the number has been multiplied by 10; similarly shifting a binary number one binary place to the left multiplies it by 2 and shifting it r places to the left multiplies it by 2^r. Conversely, a shift to the right produces a reduction in the size of a number, a shift of a binary number r binary places to the right has the effect of dividing it by 2^r, (or multiplying it by 2^{-r}). For example, the binary integer 1100 (=12) when shifted two places to the right becomes 0011 (= 3). The FP6000 function code provides for left or right shift of single or double length numbers and in each case gives the choice of one of four types of shift.

In all shift instructions both the number of places of shift and the type of shift are specified in the N-address which is regarded as split into two portions, N_S and N_t as shown:-

X	F	M	N_t	N_S
(3)	(7)	(2)	(2)	(10) bits

ASSEMBLER provides special codewords to facilitate specification of the N_t bits in a shift instruction; these are C, L, A, S which correspond to N_t = 0, 1, 2, 3 respectively.

As a safeguard against inordinate wastage of time, the maximum number of places of shift which FP6000 will carry out is not many more than the maximum meaningful shift (ie., the number of bits in the operand) but specification of an excessive shift will waste some time and, in the case of cyclic shift, produce a resultant word which is not simply related to the original operand. A shift of zero places will leave the operand unaltered in all circumstances.

The four types of shift available with each shift function are:-

Cyclic Shift (N_t = 0)

The single (or double) length word being shifted is considered as a 24 (or 48) bit binary pattern with no special significance accorded to any bit. With this type of shift all the bits are shifted around cyclically so that after each place of shift, the bit shifted off at one end of the word is replaced in the space created at the other end of the word. V cannot be set by this type of shift.

A left cyclic shift of 3 places will have an effect thus:-

before 1011101----------01

011101----------011

11101----------0110

after 1101----------01101

This type of shift is particularly useful when unpacking digits from the top portion of a word; if the result is required at the lower end of the accumulator, fewer places of shift (and hence less time) are required to carry out the operation by a left cyclic shift than by a right shift, and also no digits are lost from the word.

Logical Shift (N_t = 1)

The single (or double) length word being shifted is considered as a 24 (or 48) bit binary pattern with no special significance accorded to any bit, it will not normally represent a single number. With this type of shift, bits shifted off at the end of the word are lost and zeros are fed in at the other end of the word to replace them V cannot be set by this type of shift.

A left logical shift of 3 places will have an effect thus:-

before 1011101----------01

011101----------010

11101----------0100

after 1101----------01000

2.7.1 continued

Arithmetic Shift ($N_t = 2$)

The single (or double) length word being shifted is considered as a signed number which can equally well be an integer or a fraction. With this type of shift the precise action depends on the particular shift function being used and is described under these functions in the following sections.

Special Shift ($N_t = 3$)

The action of this type of shift depends on the state of V and the direction of shift; it is described under the individual shift functions.

2.7.2 Left Shift, Single Length. Function 110

110 X M N

The effect of this instruction is to shift the bits of the single length word in accumulator X to the left N_s places, the type of this shift being defined by N_t. The action of Cyclic and Logical Shifts is the same for all shift functions and has already been described.

Arithmetic Shift ($N_t = 2$) with this function produces a resultant word which is the same as that produced by Logical Shift in that bits are shifted off at the left of the word and zeros fed in at the right to replace them. However as the word is regarded as a signed number, V is set when capacity is exceeded, which is as soon as the sign digit is altered.

Special Shift ($N_t = 3$) with this function has exactly the same effect as Arithmetic Shift.

2.7.3 Left Shift, Double Length. Function 111

111 X M N

The effect of this instruction is to shift the bits of the double length word in accumulator X and X + 1 to the left N_s places, the type of shift being defined by N_t. The action of Cyclic and Logical shifts is the same for all shift functions and has already been described.

Arithmetic Shift ($N_t = 2$) with this function regards the operand as a double length signed number and hence only 47 bits are shifted as B0 of X + 1 is ignored and the other bits skip over this position. In fact B0 of X + 1 will always be set to zero after a shift with $N_s \neq 0$; a shift of zero places leaves the contents of X and X + 1 unaltered. Bits are shifted off at the left of the word and zeros fed in at the right to replace them; V is set when capacity is exceeded, that is if the sign digit is altered at any step of the shift.

The instruction 111 2 0 A + 3
will alter the contents of X2 and X3 as follows:-

11100----------110 and 0'11110----------011

1100----------1101 " 0'1110----------0110

100----------11011 " 0'110----------01100

00----------110111 " 0'10----------011000

Special Shift ($N_t = 3$) with this function has exactly the same effect.

2.7.4 Right Shift, Single Length. Function 112

112 X M N

The effect of this instruction is to shift the bits of the single length word in accumulator X to the right N_s places, the types of shift being defined by N_t. The action of Cyclic and Logical Shifts is the same for all shift functions and has already been described.

Arithmetic Shift ($N_t = 2$) with this function regards the word as a signed number and hence feeds in copies of the sign bit each time the word is shifted. As shifting takes place, 1's may be lost from the right of the word to the detriment of accuracy. In order to minimize this effect, the resultant number is rounded by the addition of a 1 in the B23 position if a 1 was lost when the number was shifted down for the last time; this produces a result correct to within $\pm 2^{-24}$ in the fractional convention.

2.7.4 continued

The instruction 112 2 0 A + 6

will alter the signed number in X2

from 101----------10001110000

to 111111101----------10010

Unrounded arithmetic shift can be obtained by using logical shift, provided the operand is non-negative.

Special Shift ($N_t = 3$) with this function has exactly the same effect as $N_t = 2$ if V is clear initially. However if V is set initially, the action is to clear V and shift the word right with rounding just as for $N_t = 2$ above except that it is the inverse of the original sign digit which is propagated.

The instruction 112 2 0 S + 3

will, if V is set, clear V and alter the pattern in X2

from 101----------10000

to 000101----------10

The Special Shift facility has application to the right shifting of a number whose production set V. For example to find the average of two numbers, a, b, even though their sum may overflow, division by Special Shift right will produce the correct answer. Consider the average of -1/2 and -1, addition of these two numbers sets V and produces an arithmetically incorrect result. However a subsequent right single length special shift of this result will yield the correct average, -3/4.

In detail the numbers are:

1.000---(= -1) and 1.100---(= -1/2)

Addition yields: 0.100---(= 1/2) with V set

Special Shift right one place of this sum propagates the inverse of the sign and yields:

1.010---(= -3/4) with V clear

2.7.5 Right Shift, Double Length. Function 113

113 X M N

The effect of this instruction is to shift the bits of the double length word in accumulators X, and X + 1 to the right N_S places, the type of shift being defined by N_t. The action of Cyclic and Logical Shifts is the same for all shift functions and has already been described.

Arithmetic Shift ($N_t = 2$) with this function regards the operand as a double length signed number and hence only 47 bits are shifted, as B0 of X + 1 is ignored and the other bits skip over this position. In fact B0 of X + 1 will always be set to zero after a shift with $N_S \neq 0$; a shift of zero places leaves the contents of X and X + 1 unaltered. As the number is signed a copy of the sign bit is fed in each time the word is shifted. There is no rounding to compensate for 1's shifted off the right end of the number.

The instruction 113 2 0 A + 3

will alter the contents of X2 and X3 as follows:

10----------0111 and 0'011----------0111

110----------011 " 0'1011----------011

1110----------01 " 0'11011----------01

11110----------0 " 0'111011----------0

Special Shift ($N_t = 3$) with this function has exactly the same effect as $N_t = 2$ if V is clear initially. However, if V is set initially the action is to clear V and then carry out double length right shift just as for $N_t = 2$ (above) except that it is the inverse of the original sign digit that is propagated. Thus the relationship between Special and Arithmetic Shift with this function is the same as with the 112 function and Special Shift has the same application to the right shifting of a number whose production set V.

2.8 MULTIPLICATION

The multiplication of two decimal numbers, either integers or fractions, each with the same number of digits yields a product which contains up to twice as many digits (eg. the product of 0.246 and 0.456 is 0.112176). Likewise the multiplication of two binary numbers, each of 23 bits yields a 46 bit product which becomes 47 bits with the addition of the sign bit. This product requires two computer words for its representation and such a number is called double length. All three FP6000 basic multiplication functions multiply two single length words together to form a double length product; the actual operation of the multiplication functions is unaffected by whether the operands are interpreted as fractions or integers.

The number of bits after the binary point in the product is equal to the sum of the numbers of bits after the binary point in the factors. Hence after multiplication of two integers the result will be an integer and similarly the product of two fractions is a fraction. Multiplication of an integer by a fraction yields a mixed number with the binary point lying between X and X + 1, that is to say X contains the integral part of the product and X + 1 the fractional part, the latter is always positive.

All multiplication functions take note of the signs of the factors and give a correctly signed product in all cases. The only circumstances in which overflow can occur as a result of multiplication is by squaring -1.0.

2.8.1 Unrounded Multiply.
Function 040 (n.x → x:)

040 X 0 N

This instruction takes the number in location N and multiplies it by the number in accumulator X to produce a double-length product in accumulators X and X + 1.

2.8.2 Rounded Multiply.
Function 041 (n.x + 2^{-24} → x:)

This function is mainly used when dealing exclusively with fractions to provide the single length rounded product that is often required. Although the error in using the 040 function and regarding X alone as a single length product is less than 0.00000012 this error is biased and will be magnified in an extended calculation, hence the 041 function is provided.

041 X 0 N

This instruction takes the number in location N, multiplies it by the number in accumulator X to produce a double length product in accumulators X and X + 1 and then adds a 1 in the B1 position of X + 1. This is equivalent to adding 1/2 to the contents of X + 1 (ie., 2^{-24} when multiplying two fractions) and will cause a carry into X if necessary so that a rounded single length result is given in X.

The 041 function is sometimes useful when multiplying an integer by a fraction, X will then contain the integer nearest to the value of the product, whereas, had an 040 function been used X would have contained the integral part of the product, that is the largest integer not exceeding it. For example when 9 is multiplied by 3/4 the 041 function gives x = 7 and nothing meaningful in X + 1, while the 040 function gives x = 6 and x* = 3/4.

2.8.3 Semi-Cumulative Multiply.
Function 042 (n.x + x* → x:)

This function facilitates the accumulation of products, especially when working with multiple length words.

042 X 0 N

This instruction multiplies the contents of location N by the contents of accumulator X, adds the contents of accumulator X + 1 to the least significant half of this product (with carry into X) and stores the resultant sum at X and X + 1.

This function has a ready application with small positive integers. For example, if 4 positive integers K, P, Q, R, (all less than 2^{11}) are stored in identically named locations the sequence of instructions shown overleaf will form (KP + QR) in RSLT

2.8.3 continued

F	X	M	N	Notes:
000	2	0	K	K to X2
040	2	0	P	KP to X3 (X2 clear)
000	2	0	Q	Q to X2
042	2	0	R	KP + QR to X3 (X2 clear)
010	3	0	RSLT	KP + QR to RSLT

As an example which is applicable to fractions or integers of either sign consider double length multiplication; if x: (c,d) is stored in X4, X5 and n: (a,b) is in N, N + 1 and using X1 as working space the quadruple length product of x: and n: is given in X4, X5, X6, X7 by the sequence listed below.

Note that the interpretation of the result requires knowing whether the original factors were integers or fractions.

F	X	M	N	Notes:
000	6	0	5	d to X6
040	6	0	N + 1	bd to X6, X7
042	5	0	N	ad + bd to X5, X6, X7
010	5	0	1	store most significant part (Q) in X1
000	5	0	4	c to X5
042	5	0	N + 1	bc + ad + bd - Q to X5, X6, X7
042	4	0	N	ac + bc + ab + bd - Q to X4, X5, X6, X7
005	5	0	1	Add Q to contents of X5
101	4	0	0	Add carry to contents of X4

2.9 DIVISION

It is desirable that the computer operates in such a manner that if the product of two numbers (ab) is divided by one of them, the resultant quotient should be the other.

For example: $\frac{a.b}{a} = b$

As has been shown in Section 2.8, the product of two single length numbers when formed by the computer is a double length number. Since it is often of value to know the remainder left after the division process, the process generally starts with a double length dividend and a single length divisor and produces a quotient and a remainder, both single length. To make the definitions clear, consider the division of 69 by 11; here 69 is the dividend and 11 the divisor, the resultant quotient is 6 and remainder is 3. Negative numbers are catered for and a correctly signed quotient produced in all cases.

The remainder is always numerically less than the divisor and is a number such that if it is treated as a new dividend, it will enable the division to continue with the original divisor to produce an extension to the length of the quotient (an example is given under the 045 function).

All three basic division functions will set V if an attempt is made to divide by zero (in which case the dividend is destroyed) or if the quotient exceeds capacity, eg., division of -2^{23} by -1 gives a result 2^{23} which is out of range.

As with multiplication the action of the division functions is unaffected by whether the operands are interpreted as fractions or integers. The number of bits after the binary point in the quotient is equal to the number of bits after the point in the dividend less the number of bits after the point in the divisor. It is sometimes useful to use a double length dividend in which the binary point is between the two halves; if such a number is divided by an integer then the quotient must be interpreted as a fraction and vice versa.

2.9.1 Unrounded Double Length Divide. Function 044 (x:/n → x*)

044 X 0 N

The double length dividend is the number in accumulators X and X + 1 and this is divided by the number in location N. In general the quotient resulting from this division will be an infinite binary number, however the division process is terminated when the quotient is a single-length word, at which time the quotient is in accumulator X + 1 and the remainder is in accumulator X. Overflow will be set in the circumstances applicable to all the division functions.

The single length quotient is always less than the true quotient unless the remainder is zero; it is in fact the integral part of it. The remainder always has the same sign as the divisor (since if division were to continue the quotient could only increase).

As an example of unrounded division, the following program will calculate $\frac{D \times E}{F}$ and will store the quotient in location QUOT and the remainder in REM. All quantities are integers.

F	X	M	N	Notes:
000	6	0	D	D to X6
040	6	0	E	D x E to X6, X7
044	6	0	F	Divide by F
010	6	0	REM	
010	7	0	QUOT	

2.9.2 Rounded Double Length Divide.
Function 045 ($x:/n + 2^{-24} \rightarrow x^*$)

When dividing fractions, it is the best single word approximation to the true quotient that is usually wanted in order to avoid the bias in the error that unrounded division produces. The situation is analogous to that arising using the 041 function and similar rounding is applied to produce a single length number. The 045 function can equally well be used with integers if a rounded off quotient is required.

045 X 0 N

The double length dividend is the number in accumulators X and X + 1 and this is divided by the number in location N. In general the quotient resulting from this division will be an infinite sequence of binary digits. In order to obtain the best possible single word approximation to this true quotient, the division process is terminated when the quotient is one bit longer than single word length and a 1 is then added in this position to produce a rounded quotient in accumulator X + 1. The remainder, in accumulator X, will not necessarily have the same sign as the divisor, although its absolute value will not exceed half that of the divisor. However, when dealing with fractions the remainder is not often required. Overflow will be set in the circumstances applicable to all division functions.

As an example of rounded division consider the calculation of $RES = \frac{NUM}{DEN}$ where NUM and DEN are both fractions and it is known that DEN is larger than NUM. This is obtained by the instructions:-

F	X	M	N	Notes:
000	6	0	NUM	NUM to X6
100	7	0	0	0 to X7
045	6	0	DEN	Divide
010	7	0	RES	

As an example of dividing the remainder to continue the division process, the instructions to calculate the nearest double length approximation to 1/7 (as a double length fraction) are:

F	X	M	N	Notes:
100	6	0	1	1×2^{23} to X6, X7
100	7	0	0	
100	5	0	7	7 to X5
044	6	0	5	Form first half of quotient
010	7	0	QUOT	
100	7	0	0	0 to X7
045	6	0	5	Form second half of quotient
010	7	0	QUOT + 1	

2.9.3 Single Length Integer Divide.
Function 046 (x*/n → x*)

046 X 0 N

The process used may be considered to be equivalent to clearing X and then performing an 044 function except that provision is made for the case where x* is negative.

The single length dividend is the number in accumulator X + 1 (NOT in X), and this is divided by the number in location N. In general the quotient resulting from this division will be an infinite sequence of binary digits, however the division process is terminated when the quotient is a single length word at which time the quotient is in accumulator X + 1 and the remainder in accumulator X. V will be set in the circumstances applicable to all the division functions.

The single length quotient is always less than the true quotient unless the remainder is zero, it is in fact the integral part of it. The remainder always has the same sign as the divisor (since if division were to continue the quotient could only increase). The operation of this function can be thought of as similar to the 044 function with zero contained in X. An integer dividend will most frequently be single length and the use of this function saves both time and the necessity for ensuring that accumulator X is zero before using the 044 function.

For example, the remainder after division of 69 by 11 is produced by the sequence shown in the adjacent column.

F	X	M	N	Notes:
100	3	0	69	69 to X3
100	2	0	11	11 to X2
046	2	0	2	69/11 to X3
010	2	0	REM	Remainder

The 046 function can equally well be used if the dividend is a single length fraction but if rounding is required it must be done by a programmed consideration of the size of the remainder.

As a further illustration of the use of the 046 function, suppose that the positive integer in DIST represents a distance expressed in inches and it is required to convert this to yards, feet and inches. If we denote these three quantities by y, f, i respectively and d is the distance in inches then

$$d = 36y + 12f + i$$

$$= 12\,(3y + f) + i$$

so that i is the remainder when d is divided by 12 and (3y + f) is the quotient; if this quotient is in turn divided by 3 we can find y and f. Thus the sequence of instructions shown in the table below can be used:

F	X	M	N	Notes:
100	2	0	12	12 to X2
000	5	0	DIST	d to X5
046	4	0	2	d/12 to X5
010	4	0	INS	Remainder (i)
100	2	0	3	3 to X2
046	4	0	2	(3y + f)/3 to X5
010	4	0	FEET	Remainder (f)
010	5	0	YDS	Quotient (y)

2.10 INDEXING

Accumulators X1, X2 and X3 are also called "Index Registers" and the least significant 15 bits of the contents of one of these accumulators is called an 'index'. The M address of an instruction may be used to specify one of these index registers, in which event the index is added to the N address of the instruction before it is obeyed; this process increases the time taken to obey the instruction by one core store cycle. The notation m_m is used to describe the index stored in accumulator M. It is only the instruction as obeyed that is indexed in this way; the original instruction is left unaltered in the store.

For example, if the index in accumulator X2 represents the integer 27, when the instruction:

000 6 2 513

is obeyed, the effect of this order is to transfer the contents of location 540 (= 513 + 27) to accumulator 6.

The advantage of indexing is that an instruction may effectively have different values of N at different times depending on the quantity in the index register. For example, indexing facilitates access to a table where the point of entry is not known at the time of writing the program.

Note also that branch instructions (function groups 05, 06 and 07) cannot be indexed but that their N-address has 15 bits.

Although the N address of an instruction of any other group contains only 12 bits, the index is 15 bits which provides a method of reaching storage locations above 4096. When the index is added to the N address of an order, carry beyond the 15 bits is discarded and does not set V or C.

It is often desired to obey a sequence of instructions several times before continuing to the instruction following the sequence. Such a sequence of instructions is called a 'loop', and to obey these instructions more than once it is necessary to have a means of counting the number of times a loop of instructions is obeyed. The last instruction in the loop must be a branch instruction which branches back to the start of the loop if the loop has not been traversed the desired number of times.

This process is required very often and special functions are provided to facilitate it. These functions operate in conjunction with the most significant 9 bits of an accumulator. This part of an accumulator is called a counter (denoted by X_c).

In addition to counting it is usually desired that an index be increased each time the loop is obeyed so that successive quantities in the store may be used in successive traverses of the loop.

2.10.1 Set Counter.

Function 124 ($N \rightarrow x_c$, $0 \rightarrow x_m$)

124 X M N

This instruction sets up a counter equal to the integer N in accumulator X and clears the index part; if N exceeds 511 the least significant 9 bits of it will be taken for the counter.

For example: 124 3 0 287

will store in X3 a counter of value 287 with the remainder of the word clear.

In most cases the desired initial setting of x_m is zero as provided by the 124 function. If this is not the case, x_m may be set at any other desired value by following the 124 function by a 101 function or alternatively the value of the counter and index setting may be stored with the program as a constant. ASSEMBLER provides convenient facilities for setting such constants.

2.10.2 Single Word Count. Function 060

($x_m + 1 \rightarrow x_m$, $x_c - 1 \rightarrow x_c$, branch if new $x_c \neq 0$)

060 X 0 N

This instruction operates on a counter and index, x_c and x_m contained in accumulator X, partitioned thus:

x_c (9 bits)	x_m (15 bits)

The action of the instruction is to increase x_m by 1, decrease x_c by 1 and, if after this, x_c is non-zero, branch to location N. In common with all other branch instructions this instruction cannot be indexed. There can be no carry from x_m to x_c; the maximum value of the counter is 512 and this is obtained by having x_c initially zero because carry from x_c is discarded. The function can also be used purely for counting without utilisation of x_m.

2.10.2 continued

As an example of the use of this function consider the problem of forming the sum of 100 numbers contained in successive locations starting at NUMB. One way of programming this process would be to write a separate instruction for each addition. However, the repetitive nature of this calculation enables indexing and the loop technique to be used very effectively, thus:-

F	X	M	N	Notes:
124	2	0	100	100 to $X2_c$, 0 to $X2_m$
100	6	0	0	0 to X6
→ 001	6	2	NUMB	Add one number
060	2	0	T - 1	Count
074	2	0	OVR	To error procedure

The 001 and 060 instructions are obeyed a number of times and constitute a "loop" of program.

The first time the 001 instruction is encountered $x2_m = 0$ and the address to which it refers is therefore NUMB. The 060 instruction increments $x2_m$ from 0 to 1 and decrements $x2_c$ from 100 to 99. $x2_c$ is not zero so a branch back to the 001 instruction takes place.

This time $x2_m = 1$ and this is added to the address to give NUMB + 1. The procedure is repeated until the 060 instruction decreases $x2_c$ to zero when it is obeyed for the 100th time. In this circumstance the 060 instruction does not cause a branch and instead the next instruction obeyed is the 074 which tests overflow.

2.10.3 Alternate Word Count. Function 062
($x_m + 2 \rightarrow x_m$, $x_c - 1 \rightarrow x_c$, branch if new $x_c \neq 0$)

062 X 0 N

This function is similar to the 060 function with the sole exception that the value of x_m is increased by 2 each time so that alternate words can be selected by the index.

An example of the use of this function in a loop is given in the table below which considers a set of 25 items each stored as a double length number starting at ITEM. Obtaining a total of all the items requires two accumulators into which the words are added alternately.

The loop technique is particularly applicable to this calculation because of its repetitive nature. In this case one index is used to index two different instructions.

F	X	M	N	Notes:
124	2	0	25	25 to $X2_c$, 0 to $X2_m$
100	6	0	0	Sum initially zero
100	7	0	0	
→ 005	7	2	ITEM + 1	Add least significant half
001	6	2	ITEM	Add more significant half + carry
062	2	0	T - 2	Count
074	2	0	OVR	Test overflow

2.10.3 continued

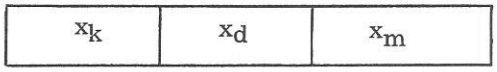
The first time the 005 and 001 instructions are encountered $x2_m = 0$ and the addresses which they refer are therefore ITEM + 1 and ITEM respectively. The 062 instruction increments $x2_m$ from 0 to 2 and decrements $x2_c$ from 25 to 24; $x2_c$ is not zero so a branch back to the 005 instruction takes place. This time $x2_m$ is 2 and this is added to the address to give ITEM + 3. The procedure is repeated until the 062 instruction decreases $x2_c$ to zero when it is obeyed for the 25th time. This time no branch takes place and the 074 instruction is obeyed next.

2.10.4 Character Count. Function 064

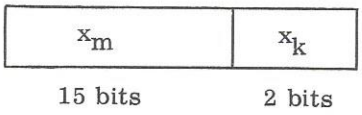
($x_k + 1 \rightarrow x_k$, carry from x_k added to x_m, $x_d - 1 \rightarrow x_d$ branch if new $x_d \neq 0$)

064 X 0 N

This instruction operates on a counter, x_d, and index, x_m and x_k, contained in accumulator X, partitioned thus:-



The characters are counted in x_d and the index consists of 17 bits which are effectively arranged thus



The value of x_k dictates which character of the word is selected. There is normal carry from x_k to x_m and hence addition of unity to x_k increments the word indexed by 1/4 so that if $x_k = 3$, x_k then becomes zero and x_m is increased by 1 (eg., character 3 of L126 is followed by character 0 of L127). The action of the function is to increase the value of the index by one character, to reduce the value of the counter by 1, and, if after the subtraction, this is non-zero, to branch to location N. The maximum value of the counter is 128 characters and this is obtained by having x_d initially zero; there is no carry between x_m and x_d. N is treated as a 15 bit quantity and hence cannot be indexed.

It will be seen that the functions in the next section make use of this type of index to select characters from storage.

2.11 PART WORD MANIPULATION

In addition to the logical operations of Section 2.5, which provide general facilities for the packing and unpacking of part words, the FP6000 function code provides separate functions to handle certain various standard sub-divisions of a word more easily and quickly.

2.11.1 Extract Character, Function 024 ($n_j \rightarrow x$)

As we have seen the 24-bit FP6000 word can contain four 6-bit characters and in these circumstances B0 is not treated as a sign bit.

024 X M N

This instruction, if not indexed, extracts the least significant 6 bit character of the word location N (ie. n_3) and places it in the least significant 6 bit positions of accumulator X, the remainder of X being cleared. In use, however, the instruction will often be indexed by a character index and in this case the instruction extracts from the word in location N the character specified by the m_k bits from the index register and stores this in the least significant 6 bit positions of accumulator X with the remainder of X being cleared. In particular, extraction of character n_0 requires using an actual index register which has $x_k = 0$ because setting M = 0 means the instruction is not indexed and hence n_3 will be extracted.

The 024 instruction, if used in a loop terminated by an 064 instruction, may be used to select successive characters from storage. For example, to form in X7 the sum of nine characters from 123.3 to 125.3 inclusive requires the sequence listed in the table below.

Note in particular the use of the 124 instruction to set a non-zero value of x_k (here x_k = 3) so that a character other than character 0 is the first selected from the word.

The first time the 024 instruction is obeyed $x3_k = 3$ and character 123.3 is extracted. The 064 instruction adds 1 to $x3_k$ with carry to x_m so that the result is now $x3_k = 0$, $x3_m = 1$; also $x3_d$ is decreased from 9 to 8. As $x3_d$ is non-zero a branch takes place back to the 024 instruction. The index has $x3_k = 0$ so that character 0 is extracted, and $x3_m = 1$ so that this character is 124.0. When the 064 instruction increases $x3_k$ by 1 this becomes 1 and there is no carry to $x3_m$. The process continues until $x3_d$ is reduced to zero and the loop is not then repeated again.

2.11.2 Insert Character. Function 034 ($x_3 \rightarrow n_j$)

This function is the converse of the 024 function

034 X M N

This instruction if not indexed, takes the least significant six bits (B18 to B23) of the word in accumulator X and inserts them into character position 3 of the word in location N without altering the other 18 bits of the word. In use however the order will often be indexed by a character index and in this case the instruction takes the least significant six bits (B18 to B23) of the word in accumulator X and inserts them in that character position of the word in location N which is specified by the m_k bits of the index register, the remainder of n being unchanged. For example, the following instructions insert bits B18 to B23 of X4 as character 174.0.

100	2	0	0	Set index
034	4	2	174	

Note than it is necessary to use here an actual index register that has $x_k = 0$ to insert character n_0 because writing M = 0 in the instruction would mean that this is not indexed and hence the bits could be inserted as 174.3 as stated above.

F	X	M	N	Notes:
124	3	0	393	Set $x_k = 3$, $x_d = 9$, $x_m = 0$
100	7	0	0	Clear for sum
→024	6	3	123	Extract character
001	7	0	6	Add character
064	3	0	T - 2	Character count

2.11.3 Extract Exponent. Function 025
($n_e \rightarrow x$)

This function is used principally for extracting the exponent of a number represented in floating point form as will be described in Section 2.13.

025 X M N

This instruction extracts the least significant 9 bits (n_e or B15 to B23) of the word in location N and places them in the corresponding position in accumulator X, the remainder (B0 to B14) of X being cleared.

2.11.4 Insert Exponent. Function 035
($x_e \rightarrow n_e$)

This function, the converse of the 025 function, is used principally for inserting the exponent when representing a number in floating point form.

035 X M N

This instruction replaces the least significant 9 bits (n_e or B15 to B23) of the word in location N by the corresponding bits of the word in accumulator X without disturbing the other bits of n.

It is important to note one essential difference between these complementary instruction pairs. When extracting a character from a storage location to an accumulator, the rest of the accumulator is cleared, whereas this is not true when the character is inserted into a storage location from an accumulator, since the remainder of the word is left intact.

2.11.5 Insert x_a. Function 036
($x_a \rightarrow n_a$)

036 X M N

The action of this instruction is to replace the least significant 12 bits (B12 to B23) of the word in location N by the corresponding bits of the word in accumulator X without disturbing the other bits of n. For example, suppose numbers less than 4096 are being stored two to a word, in B0 to B11 and B12 to B23. Then the instructions

```
100   3   0   2000
036   3   0   WORD
```

will alter the number in B12 to B23 of WORD to be 2000 without altering the number stored in the other half of WORD.

2.11.6 Insert x_m. Function 037
($x_m \rightarrow n_m$)

037 X M N

The action of this instruction is to replace the least significant 15 bits (B9 to B23) of the word in location N by the corresponding bits of the word in accumulator X, without disturbing the other bits of n. Thus, if two items, one comprising 9 bits, B0 to B8 and the other comprising 15 bits, are stored in a single word, this instruction can be used to alter the 15 bit item, without affecting the other one.

2.12 CHARACTER CONVERSION

Although FP6000 works internally with quantities expressed in binary, data input and output must be in decimal form. For this purpose information is stored as 6-bit characters, which enable both numeric and alphabetic data to be handled. Each 6-bit character may represent one of 64 symbols which are the decimal digits 0-9, the letters of the alphabet A-Z and certain other symbols. This code is listed in chapter 5.

2.12.1 Decimal to Binary Conversion. Function 043 ($10.x: + n_j \rightarrow x:$)

On input, decimal numbers have to be converted from 6-bit characters into binary in order to perform arithmetic operations easily. These numbers will appear in the store as a sequence of binary coded decimal digits followed by a non-numeric character such as space. The 043 function provides a fast and easy method of converting 6 bit binary coded decimal characters into a pure binary integer but it can also be used for other multiplications by ten.

043 X M N

The effect of this instruction is to take a specified character from N, check that it is numeric, then multiply the double length binary number in accumulators X and X + 1 by 10 and add the character to this. If the character is non-numeric neither the multiplication by 10 nor the addition takes place but instead C is set; this provides a convenient way of detecting either an illegal character or the end of the number being converted. If the 043 instruction is not indexed, the character taken from N will always be n_3; normally however the instruction will be indexed by a character index so that each character of N will be selected in turn. Overflow will be set if the contents of X and X + 1 exceed capacity when multiplied by 10 and an incorrect result will be given if x: is initially negative.

As an example, consider the conversion of a decimal integer stored from character 0 of DATA onwards; this will be converted to a binary integer in X5 and X6 by the instructions shown in the table below.

Note that the first 100 instruction sets the counter portion of the index equal to zero which will be interpreted by the 064 instruction as the maximum possible count of 128 characters. The 064 instruction is not used for counting but merely to increment the index; the exit from the loop is via the 074 instruction when a non-numeric character is encountered and it is essential that the character counter has not been reduced to zero before this, therefore this counter is started at its maximum value.

Conversion of decimal fractions to binary is achieved by applying the 043 function in exactly the same way as for integers and then dividing the binary integer which results by 10^r where r is the number of digits after the decimal point. Thus, 0.0257 is obtained in binary by converting the digits 0257 to a binary integer by obeying an 043 instruction four times and then dividing this integer by 10^4. A decimal fraction stored from character 0 of DATA onwards will be converted to a binary fraction in X6 by the previous program example with the addition of one instruction, see overleaf.

F	X	M	N	Notes:
100	2	0	DATA	Set index
100	5	0	0	Clear locations before accumulating number
100	6	0	0	
043	5	2	0	Convert one digit
074	5	0	T + 2	Test for non-numeric character
064	2	0	T - 2	Return to convert next character

2.12.1 continued

F	X	M	N	Notes:
100	2	0	DATA	
100	5	0	0	
100	6	0	0	
→043	5	2	0	
074	5	0	T + 2	
064	2	0	T - 2	
→100	7	0	0	
044	6	0	TENR	Division by 10^r

F	X	M	N	Notes:
124	2	0	139	Set $x_k = 1$, $x_d = 11$
101	2	0	100	Set $x_m = 100$
→047	5	2	0	
064	2	0	T - 1	

Note that the use of N = 0 in the 047 instruction combined with the setting of $x_m \neq 0$, enables the same loop to be used for the conversion of fractions stored in other locations, by merely setting a different counter and index in X2.

2.12.2 Binary to Decimal Conversion. Function 047 (10.x: → x:, character → n_j)

This function is mainly intended to provide a convenient method of converting a binary fraction into 6 bit binary coded decimal characters. To give a correct result, the binary number must be positive in sign. Since it must also be a fraction, the function is not the exact converse of the 043 function.

047 X M N

The effect of this instruction is to take the binary fraction stored in accumulators X and X + 1 and multiply this by 10. The integral part of the product is stored as a 6 bit character in N while the fractional part is replaced in the accumulators. If the instruction is not indexed the character is stored as n_3; if it is indexed the character is stored as n_j as specified by the m_k bits of the index register. In either case the remaining 18 bits of n are unaltered.

A facility is provided for replacing initial zeros by spaces in the resultant decimal number; this is described under the 125 function following.

As an example, consider a binary fraction in X5 and X6; this will be converted to a fraction with 11 places of decimals (including non-significant zeros) and stored from 100.1 to 102.3 by the instructions shown in the following table.

If binary integers are to be converted to decimal, it is necessary to precede each conversion by program instructions to divide the integer by 10^r (where r is the maximum number of decimal characters in any one integer). Conversion by the 047 function of this quotient as a binary fraction will then produce the correct decimal integer. For example the binary integer 11001 if divided by 10^2 will give decimal 25 when converted whereas if divided by 10^4 it would give 0025.

The following instructions will convert a binary integer, INT, into a decimal integer comprising r digits from 100.0 onwards.

F	X	M	N	Notes:
000	5	0	INT	
000	6	0	TENR	
044	5	0	TENR	Division by 10^r
124	2	0	R	Set counter = r
→047	6	2	100	
064	2	0	T - 1	

F	X	M	N	Notes:
001	3	0	5	$(B_e + C_e)$ to X3
040	4	0	2	$B_r . C_r$ to X4, X5
103	3	0	512	Subtract 512 from sum of exponents
114	4	3	256	Normalize $B_r . C_r$ using $(B_e + C_e)$ as exponent

2.13.2 continued

Hence the final result is $1/2.2^1$. Initially the exponent in X3 is (256 + 2) and the final exponent set in X5 by the 114 instruction will be (256 + 2 - 1).

(b) Suppose $B = 1/2.2^2$ and $C = 3/4.2^2$, then $(B_r + C_r)$ as produced in X4 by the 001 instruction will be 1+1/4, which is outside the permitted range for the argument, and V will be set. The effect of the 114 normalize instruction will be to clear X5 and then, because V is set, to shift this $(B_r + C_r)$ down one place and increase the exponent by 1. Hence the final result is $5/8.2^3$. Initially the exponent in X3 is (256 + 2) and the final exponent set in X5 by the 114 instruction will be (256 + 2 + 1).

As a second example the floating point product in X4, X5 of two non-zero floating point numbers in the 24 bit argument form, B in X4, X5 and C in X2, X3, will be obtained, assuming V is clear, by these instructions listed in the table above.

If the 114 normalize instruction needs to instigate shift of $B_r . C_r$, this will be up one place, eg. $B_r = C_r = 3/4$ yields $B_r . C_r = 9/16$ which requires no shift while $B_r = C_r = 1/2$ yields $B_r . C_r = 1/4$ which requires one place of shift up. Only when both B_r and C_r equal -1.0 will V be set and the shift be down one place. N = 256 is required in the 114 instruction because when the exponents are added by the 001 instruction the 256's contained in B_e and C_e become 512 which overflows 9 bits and is lost because the 114 instruction only stores 9 bits as the final exponent in X3.

2.13.3 Double Length Normalize. Function 115

115 X M N

This instruction will normalize a floating point number whose argument is given in accumulators X and X + 1 (B15 to B23 of X + 1 are ignored) and whose exponent is N. It is designed to be used with 38 bit arguments but will work with 24 bit ones. The action and use of the function are the same as for the 114 function except that none of X + 1 is cleared at the start and the shifts are double length. As an example the sum in X4, X5 of two non-zero floating point numbers with 38 bit arguments, B in X4, X5 and C in X6, X7, will be obtained, assuming $C_e \geqslant B_e$ and V is clear, by the instructions listed in the table below in which the subscripts e and r denote exponent and argument respectively.

Compare this with the first example under function 114, noting the increase in complexity; the effect of normalization as shown by the numerical examples is the same.

F	X	M	N	Notes:
025	1	0	7	C_e to X1
025	3	0	5	B_e to X3
002	3	0	3	Negate B_e
001	3	0	1	$(C_e - B_e)$ to X3
113	4	3	A	Shift B_r
005	5	0	7	$B_r + C_r \rightarrow B_r$
001	4	0	6	
115	4	1	0	Normalize $(B_r + C_r)$ using C_e as exponent

2.13 FLOATING POINT NUMBERS

The problem of scaling arises in many programs. The numbers entering into a calculation often cannot be conveniently represented directly as integers or fractions and must then all be scaled in some way to bring them in range or prevent loss of accuracy. Quantities are usually scaled by powers of 2 rather than powers of 10 because multiplication or division by a power of 2 can be performed easily by shifting.

In some problems scaling is difficult because certain numbers may vary over a wide range or because it is practically impossible to determine accurately the size of all the intermediate results; making the scaling factor large enough to avoid overflow in all circumstances may lead to great loss of accuracy, even with double length working. The best approach is to use floating-point working which automatically takes care of all scaling; in contrast, the usual mode of operation is called fixed-point.

A common way of writing very large or very small numbers in scientific work is by introducing a power of 10 as a scaling factor, eg. -3.457×10^{12} or 1.234×10^{-4}. Here the decimal point is written immediately after the first significant digit of the number and the power of 10 makes the overall value of the number correct. A similar technique is used in FP6000 except that, since it is a binary machine, powers of 2 are used.

A quantity y is represented in the form:

$$y = R.2^e$$

where e (called the exponent) is a signed integer which is chosen to bring R (called the argument) within a convenient fractional range. This means that the quantity y is represented inside the computer by two numbers, R and e, the base 2 being implicit. R is stored as a fraction. In order to preserve the greatest accuracy, R must lie in one of the ranges $1 > R \geqslant 1/2$ or $-1/2 > R \geqslant -1$ unless R = 0. A number which has a zero argument is taken as having zero value regardless of what the exponent is.

In practice it would be wasteful to allocate a complete word to store the exponent of a floating point number because this is seldom very large in absolute value. It would be unusual to have a number as big as 2^{255} (which corresponds to about 10^{76}) or as small as 2^{-256}, hence 9 bits can adequately represent the exponent; this must be signed, so values can be taken in the range

$$-256 \leqslant e \leqslant 255$$

which corresponds approximately to

$$10^{-76} < y < 10^{+76}$$

In fact, the exponent store of an FP6000 floating point number contains not e but e + 256, which is never negative and lies in the range 0 to 511. To avoid loss of accuracy in arithmetic it is desirable that the exponent associated with floating point zero is the smallest one available in the allowed range and the advantage of the "e + 256" representation is that this smallest value 2^{-256}, is represented by zero in the exponent store, and hence standard floating point zero (0.2^{-256}) has both argument and exponent zero. (0.2^{-256}) is detected by the same "branch on zero" functions as fixed point zero.

There are two alternative ways of storing the argument and exponent of floating point numbers in FP6000, both require two words and the 9 bit exponent store is always B15 to B23 of the second word. The difference is that in one case the argument is contained solely in the 24 bits of the first word, in the other case the argument is 38 bits in length (B0 to B23 of the first word and B1 to B14 of the second word). The former yields considerable economy in time in floating point multiplication and division but with a reduction in accuracy.

2.13.1 Normalization

Arithmetic operations on floating point numbers require a sequence of instructions and are easily performed using the functions already described. The final step of shifting the argument into standard form and adjusting the exponent is however more difficult and two special functions are provided to accomplish this; one for each of the FP6000 floating point formats. This operation is called "normalizing" a floating point number. If during normalization the exponent of a number becomes greater than 255, floating point overflow is said to occur and V will be set. Similarly if the exponent becomes less than -256 the number is equated to floating point zero.

2.13.2 Single Length Normalize. Function 114

114 X M N

This instruction will normalize a floating point number whose single length argument is given in accumulator X and whose exponent is N. In normal use the original exponent of the number to be normalized will be contained in an accumulator which will be used to index N. The 114 function cannot be used with floating point numbers with 38 bit arguments.

The instruction first clears X + 1, subsequent action depends on the state of V: if V is set the instruction assumes that the argument has overflowed by one bit position only. If V is clear, a single length left arithmetic shift (see function 110, N_t = 2) of accumulator X occurs and at the same time N is reduced by 1 for each place shifted. The shift terminates when either:

(a) The next shift would change the sign bit of X

(b) N is counted down to zero

(c) The argument of the number is shown to have been zero from the start.

When condition (a) is reached the argument of the floating point number will be in normal form; if the number has then been shifted s places, where s may be zero, (N-s) will be inserted as the exponent in the exponent store (B15 to B23 of X + 1). The result when condition (b) or (c) is reached is floating point zero with both argument and exponent cleared. If (N-s) is greater than 511 so that floating point overflow occurs, V is set. (N-s) cannot be negative because N must be positive and the count down ceases at zero.

If V is set originally, the shift, after X + 1 has been cleared, is an unrounded single length right special shift (see Function 112, N_t = 3, but without rounding) one place of accumulator X with the addition of 1 to N. The resultant exponent is the placed in bits B15 to B23 of X + 1 as described above.

When the exponent of the number being normalized is contained in an index register, N will be written as either 0 or 256, depending on the form which this exponent takes. If it is in the same "e + 256" form which is used in the exponent store of a standard floating point number then N is written as 0, as in the first example below. However if the exponent in the index register is the sum or difference of two standard exponents, as happens in floating point multiplication or division, then N must be written as 256 to obtain the standard "e + 256" form in the normalized number; the second example shows this.

As a first example, the sum in X4, X5 of two non-zero floating point numbers with 24 bit arguments, B in X4, X5 and C in X2, X3, will be obtained assuming $C_e \geqslant B_e$ and V is clear, by the instructions listed in the table below.

The effect of normalization is shown by considering two numerical cases:

(a) Suppose $B = -1/2.2^2$ and $C = 3/4.2^2$, then $(B_r + C_r)$ as produced in X4 by the 001 instruction will be 1/4, which is outside the permitted range for the argument. The effect of the 114 normalize instruction will be to clear X5 and then, because V is clear, to shift this $(B_r + C_r)$ up until the next shift would change the sign bit, ie. there is a left shift one place and the exponent is reduced by 1.

F	X	M	N	Notes:
000	1	0	3	C_e to X1
013	5	0	1	$(C_e - B_e)$ to X1
112	4	1	A	Shift B_r down $(C_e - B_e)$ places
001	4	0	2	$(B_r + C_r) \rightarrow B_r$
114	4	3	0	Normalize $(B_r + C_r)$ using C_e as exponent

The subscripts e and r denote exponent and argument respectively

F	X	M	N	Notes:
001	3	0	5	$(B_e + C_e)$ to X3
040	4	0	2	$B_r . C_r$ to X4, X5
114	4	3	256	Normalize $B_r . C_r$ using $(B_e + C_e)$ as exponent

2.13.2 continued

Hence the final result is $1/2.2^1$. Initially the exponent in X3 is (256 + 2) and the final exponent set in X5 by the 114 instruction will be (256 + 2 - 1).

(b) Suppose $B = 1/2.2^2$ and $C = 3/4.2^2$, then $(B_r + C_r)$ as produced in X4 by the 001 instruction will be 1+1/4, which is outside the permitted range for the argument, and V will be set. The effect of the 114 normalize instruction will be to clear X5 and then, because V is set, to shift this $(B_r + C_r)$ down one place and increase the exponent by 1. Hence the final result is $5/8.2^3$. Initially the exponent in X3 is (256 + 2) and the final exponent set in X5 by the 114 instruction will be (256 + 2 + 1).

As a second example the floating point product in X4, X5 of two non-zero floating point numbers in the 24 bit argument form, B in X4, X5 and C in X2, X3, will be obtained, assuming V is clear, by these instructions listed in the table above.

If the 114 normalize instruction needs to instigate shift of $B_r . C_r$, this will be up one place, eg. $B_r = C_r = 3/4$ yields $B_r . C_r = 9/16$ which requires no shift while $B_r = C_r = 1/2$ yields $B_r . C_r = 1/4$ which requires one place of shift up. Only when both B_r and C_r equal -1.0 will V be set and the shift be down one place. N = 256 is required in the 114 instruction because when the exponents are added by the 001 instruction the 256's contained in B_e and C_e become 512 which overflows 9 bits and is lost because the 114 instruction only stores 9 bits as the final exponent in X3.

2.13.3 Double Length Normalize. Function 115

115 X M N

This instruction will normalize a floating point number whose argument is given in accumulators X and X + 1 (B15 to B23 of X + 1 are ignored) and whose exponent is N. It is designed to be used with 38 bit arguments but will work with 24 bit ones. The action and use of the function are the same as for the 114 function except that none of X + 1 is cleared at the start and the shifts are double length. As an example the sum in X4, X5 of two non-zero floating point numbers with 38 bit arguments, B in X4, X5 and C in X6, X7, will be obtained, assuming $C_e \geqslant B_e$ and V is clear, by the instructions listed in the table below in which the subscripts e and r denote exponent and argument respectively.

Compare this with the first example under function 114, noting the increase in complexity; the effect of normalization as shown by the numerical examples is the same.

F	X	M	N	Notes:
025	1	0	7	C_e to X1
025	3	0	5	B_e to X3
002	3	0	3	Negate B_e
001	3	0	1	$(C_e - B_e)$ to X3
113	4	3	A	Shift B_r
005	5	0	7	$B_r + C_r \rightarrow B_r$
001	4	0	6	
115	4	1	0	Normalize $(B_r + C_r)$ using C_e as exponent

2.14 SUBROUTINES

It is frequently found that the same sequence of instructions to perform a specific operation is required at many places in a program. While it is feasible to copy such a sequence into the program at each point where it is required, this is wasteful of storage space and effort. Instead it is possible to arrange that this necessary sequence of instructions, which is known as a "subroutine", appears only once and is used, each time it is required, by branching to it. Then when the subroutine has completed its operation, a return must be made to the main program, usually referred to as the "master program". The return location will depend on the point from which the entry to the subroutine was made. Usually, the return will be to the instruction following the subroutine entry branch instruction. In order for this to take place, the master program must provide information to the subroutine specifying the return location. This information is called a "link". Two special functions are provided to facilitate entry and exit to and from subroutines.

Many of the operations commonly required, eg., Square root, binary-decimal conversion, etc. are provided in the form of subroutines in a library. The programmer is thus saved the work of writing and developing his own subroutine for these operations.

It is quite possible that a subroutine may itself have other subroutines of a still lower "level". For example, the library subroutine to calculate the inverse sine of a number uses in turn another library subroutine to calculate square roots.

Note that subroutine A has been entered twice from different points in the master program. Also subroutine C is entered directly both by the master program and by subroutine B and hence is of a lower level than subroutines A and B.

2.14.1 Subroutine Entry. Function 070

This function is used as a connection between two levels of program, that is, it provides the branch from the main program to a subroutine or from one subroutine to another of lower level.

070 X 0 N

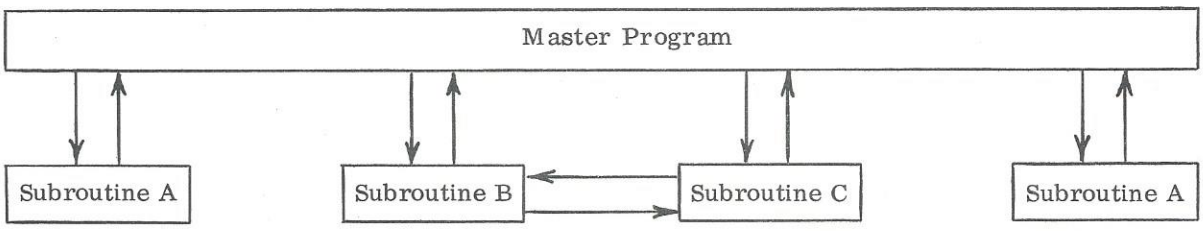
The effect of this instruction is to store in B0 of accumulator X, the state of V and, in B9 to B23, the location of the next instruction; then it clears V and branches out of the normal sequence of instructions to the first instruction of the subroutine, which will be contained in the location specified by N. Sometimes the actual address of the entry point to the subroutine will be known, but, more often, the subroutine's code name, which is attached as a label to the first instruction of the subroutine during input, is used. If a subroutine has more than one entry point a composite label can be used (eg., "SIN + 10", to enter at an instruction 10 subsequent to the original SIN entry), however, it is often easier to assign separate labels. An example of the use of both Subroutine Entry and Exit is given under the 072 function.

2.14.2 Subroutine Exit. Function 072

This function, which terminates a subroutine, is used to provide the connection from the subroutine back to the master program or back to a subroutine of higher level.

072 X 0 N

The effect of this instruction is to branch to a point N locations beyond the address contained in B9 to B23 of X. If B0 of X is a 1, or if V is set when the instruction is obeyed, V is left set by this instruction.



2.14.2 continued

In most cases, the instruction is used to terminate a subroutine and X then contains a link back to the part of the program which was being obeyed before the subroutine was entered, (the 070 function stores the link in an accumulator). With N = 0, this instruction provides a return to the instruction following the subroutine entry instruction.

Occasionally parameters for use in the subroutine have to be provided by the master program and a convenient place to store these is in the locations immediately following the 070 subroutine entry instruction. The subroutine can then fetch these parameters easily by using x as an index. In these cases, the return to the master program will probably be to the location immediately following these parameters. Suppose these are 5 such parameters, the return to the correct location can then be made by setting N = 5 in the 072 instruction. The final result of the 072 instruction is as if the N address is indexed by accumulator X (where X can be any of the 8 accumulators); this fact can be used to provide an unconditional branch instruction which has indexing facilities. The drawback to this is that V may be altered by an 072 function hence the 023 function is preferable.

An example of the use of the Subroutine Entry and Exit functions is given by the following sequence of main program instructions and a three-instruction subroutine.

F	X	M	N	Notes:
001	5	0	ITEM	
070	1	0	DBLA	Subroutine Entry
xxx	x	x	xxx'	
005	6	0	NBR1	
001	5	0	NBR2	
072	1	0	0	Subroutine Exit

The 070 instruction will cause a branch to the subroutine at location DBLA onwards and store the appropriate link in X1. The 072 instruction which terminates this subroutine will take the link from X1 and because N = 0, cause a branch back to the instruction following the 070 instruction in the main program.

2.15 MISCELLANEOUS

2.15.1 Obey n as an Instruction. Function 023

023 0 M N

This instruction causes the contents of location N to be obeyed as an instruction, as though it were in the location occupied by the 023 instruction. The effect is a temporary branch for one instruction because the instruction obeyed after this is the next one in the original sequence, unless the instruction in N is a branch instruction which transfers control to some other location. The instruction obeyed in N may be another 023 instruction. The 023 function does not itself alter C so that the instruction obeyed in N may include carry which is set by the instruction preceeding the 023 instruction.

As an example, consider a quantity ITEM on which it is required to perform one of four different operations according as X2 contains the integer 0, 1, 2 or 3. The four operations in question are respectively, "no change", "negate ITEM", "clear ITEM" and "branch to a special section of program" This necessitates storing in INST onwards, appropriate instructions viz:

F	X	M	N
000	6	0	ITEM
002	6	0	ITEM
100	6	0	0
074	0	0	ALT

Then the complete program required is:-

F	X	M	N
023	0	2	INST
010	6	0	ITEM

If, for example, x2 = 2 the actual instructions obeyed will be:

F	X	M	N
100	6	0	0
010	6	0	ITEM

2.15.2 Clear Location. Function 033 (0 → n)

033 0 M N

This instruction will write a zero word into location N.

2.15.3 No Operation. Function 123

This function is used in two forms:

The instruction 123 0 0 0 fulfills the function of a dummy instruction and the computer takes no action except to go on to obey the next instruction. During the formulative stages of a program it is useful to be able to incorporate dummy instructions which will be replaced subsequently by active instructions.

On the other hand the instruction below should not be used as it has a special effect described in Chapter 8.

123 7 0 0

2.15.4 Block Transfer. Function 126

This function provides a simple and automatic way of moving a block of N consecutive words from one place in the core store to another.

The instruction 126 X M N will copy the word whose storage location is specified in accumulator X into the location specified in accumulator X + 1. Unity is then added to the addresses of both of these storage locations and the procedure repeated until a total of N words has been transferred. From this it can be seen that overlapping transfers are only possible when the address to which the word is written is numerically less than the address from which it is read. For example, to transfer the 3 words from L88 to L90 into L87 to L89 requires the instructions:-

F	X	M	N
100	1	0	88
100	2	0	87
126	1	0	3

2.15.4 continued

Use of the block transfer function is the fastest method of transfer when more than two consecutive words are involved. The number of words transferred is actually given by the least significant nine bits of N unless these are all zeros. Hence the maximum block length possible is 512 consecutive words and this is obtained with N set as zero or a multiple of 512.

2.15.5 Form Check Sum, Function 127

This function provides a simple and automatic way of adding up the values of a block of N words in consecutive core storage locations.

The instruction 127 X M N will replace the previous contents of accumulator X with the sum of N consecutive words starting with the word whose storage location is specified on accumulator X + 1. Bits will be lost at the most significant end of the sum when this exceeds single word capacity but these will be ignored and in no circumstances will overflow be set by the instruction. This restricts the application of the function for normal addition purposes. The number of words summed is actually given by the least significant nine bits of N unless these are all zeros. Hence the maximum number of words summed is 512 and this is obtained with N set as either zero or a multiple of 512. For example the sum of the 15 words from ITEM onwards will be placed in X3 by the instruction:

F	X	M	N
100	4	0	ITEM
127	3	0	15

Use of the 127 function is the fastest method of forming a check sum when more than 3 consecutive words are involved.

FUNCTION CODE SUMMARY

Function	Description	Time (usecs) 6 usec Core**	Time (usecs) 2 usec Core**	Notes	Section of Chapter 2
000	$n + c \rightarrow x$	18	7	Overflow may be set on exit but Carry is cleared	3.3
001	$x + n + c \rightarrow x$	18	7		
002	$-n - c \rightarrow x$	18	7		
003	$x - n - c \rightarrow x$	18	7		
004	$n + c \rightarrow x$	18	7	Overflow cannot be set Sign of result always positive. Carry set if appropriate.	
005	$x + n + c \rightarrow x$	18	7		
006	$-n - c \rightarrow x$	18	7		
007	$x - n - c \rightarrow x$	18	7		
010	$x + c \rightarrow n$	18	7	As 000-003 but with n and x interchanged	
011	$n + x + c \rightarrow n$	18	7		
012	$-x - c \rightarrow n$	18	7		
013	$n - x - c \rightarrow n$	18	7		
014	$x + c \rightarrow n$	18	7	As 004-007 but with n and x interchanged	
015	$n + x + c \rightarrow n$	18	7		
016	$-x - c \rightarrow n$	18	7		
017	$n - x - c \rightarrow n$	18	7		
020	$x \,\&\, n \rightarrow x$	18	7	Logical AND	5.1
021	$x \vee n \rightarrow x$	18	7	Logical INCLUSIVE OR	5.2
022	$x \not\equiv n \rightarrow x$	18	7	Logical EXCLUSIVE OR	5.3
023	Obey n as an instruction	7	3		15.1
024	$n_j \rightarrow x$	18	7	Extract character	11.1
025	$n_e \rightarrow x$	18	7		11.3
026	Set C if $n \neq x$ or $c = 1$	18	7		6.1
027	Set C if $n + c > x$	18	7	Similar to 007 but result not stored	6.2
030	$n \,\&\, x \rightarrow n$	18	7	As 020-022, result in N	5.1
031	$n \vee x \rightarrow n$	18	7		5.2
032	$n \not\equiv x \rightarrow n$	18	7		5.3
033	$0 \rightarrow n$	18	7	Clear n	15.2
034	$x_3 \rightarrow n_j$	18	7	Part word conversion	11.2
035	$x_e \rightarrow n_e$	18	7		11.4
036	$x_a \rightarrow n_a$	18	7		11.5
037	$x_m \rightarrow n_m$	18	7		11.6
040	$n.x \rightarrow x:$	67	40	Unrounded multiply	8.1
041	$n.x + 2^{-24} \rightarrow x:$	67	40	Rounded multiply	8.2
042	$n.x + x^* \rightarrow x:$	72	41	Semi-cumulative multiply	8.3
043	$10.x: + n_j \rightarrow x:$	58	27	Decimal-binary conversion	12.1
044	$x:/n \rightarrow x^*$	76	45	Unrounded double length division (remainder to X)	9.1
045	$x:/n + 2^{-24} \rightarrow x^*$	79	48	Rounded double length division (remainder to X)	9.2
046	$x^*/n \rightarrow x^*$	71	44	Unrounded single length division (remainder to X)	9.3
047	$10.x: \rightarrow x:$, Char $\rightarrow n_j$	58	27	Binary-decimal conversion	12.2

** Indexing increases these times by 6 or 2 usecs respectively

Function Code Summary (continued)

Function	Description	Time (usecs) 6 usec Core**	Time (usecs) 2 usec Core**	Notes	Section of Chapter 2
050	Branch if x = 0	13	5		4.1
052	Branch if x ≠ 0	13	5		4.1
054	Branch if x ≥ 0	13	5		4.1
056	Branch if x < 0	13	5		4.1
060	Single word modify	13	5	Add 1 to modifier	10.2
062	Alternate word modify	13	5	Add 2 to modifier	10.3
064	Character modify	13	5	Add 1/4 to modifier (subtract 1 from counter, branch if counter non-zero)	10.4
070	Subroutine entry	15	8	Store ONR and V in x, clear V, branch to N	14.1
072	Subroutine exit	12	4	Branch to N + x, V reset as at entry unless set by subroutine	14.2
074	Branch on Condition X:	7	3		
	X = 0 unconditionally				4.2
	X = 1 if V set				4.3
	X = 2 if V set, clear V				4.3
	X = 3 if V clear				4.3
	X = 4 if V clear, clear V				4.3
	X = 5 if C set				4.4
	X = 6 if C clear				4.4
	X = 7 if V clear, invert V				4.3
100	N + c → x	12	5	As 000-003 but N used in place of n	3.4
101	x + N + c → x	12	5	As 000-003 but N used in place of n	3.4
102	- N - c → x	12	5	As 000-003 but N used in place of n	3.4
103	x - N - c → x	12	5	As 000-003 but N used in place of n	3.4
104	N + c → x	12	5	As 004-007 but N used in place of n	3.4
105	x + N + c → x	12	5	As 004-007 but N used in place of n	3.4
106	- N - c → x	12	5	As 004-007 but N used in place of n	3.4
107	x - N - c → x	12	5	As 004-007 but N used in place of n	3.4
110	Left Shift x, N_s places	18 + N_s	6 + N_s	N_t = 0 Cyclic Shift	7.1
111	Left Shift x:, N_s places	42 + N_s	15 + N_s	N_t = 1 Logic Shift	7.2
112	Right Shift x, N_s places	18 + N_s	6 + N_s	N_t = 2 Arithmetic Shift	7.3
113	Right Shift x:, N_s places	42 + N_s	15 + N_s	N_t = 3 Special Shift	7.4
114	Normalize x	37 + s	15 + shift		13.2
115	Normalize x:	42 + s	16 + shift		13.3
120	x & N → x	12	5	As 020-022 but N used in place of n	5.1
121	x v N → x	12	5	As 020-022 but N used in place of n	5.2
122	x ≢ N → x	12	5	As 020-022 but N used in place of n	5.3
123	No operation	7	3	Dummy instruction	15.3
124	N → x_c, 0 → x_m	13	5	Set counter	10.1
125	Set Mode N	8	4		12.3
126	Block Transfer	32 + 12N	13 + 4N	N words from address x to address x*	15.4
127	Check Sum → x	32 + 7N	14 + 3N	Sum N words from address x* ignoring overflow	15.5

Function Code Summary (continued)

Function	Description
130	Convert Fixed to Floating
131	Convert Floating to Fixed
132	x: + n: → x:
133	x: - n: → x:
134	x: . n: → x:
135	x: / n: → x:
136	$\sqrt{n:}$ → x:
	Floating Point Operations
150	Suspend me if my unit X of type N is busy
151	Release my unit X of type N
152	Disengage my unit X of type N
153	Place in my register 9 control registers of my unit X of type N
154	Read more program from my unit X of type N
155	Suspend and dump on my unit X of type N
156	Abolish this program
157	Suspend this program awaiting operator message to EXECUTIVE
160	Suspend this program awaiting an operator's message to EXECUTIVE and print on the console typewriter my message from control word N
161	Spare
162	Suspend me if my subprogram X is active
163	Activate my subprogram X, entering at instruction N
164	Suspend this subprogram awaiting activation by the master, and, if necessary release suspension on the master due to this subprogram
170	Paper Tape Read
171	Paper Tape Punch
172	Print one line
173	Card read
174	Card punch
175	Magnetic tape operation
176	Drum operation
177	Peripheral Typewriter operation
	from unit X with control word N

TABLE OF POWERS OF 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 063
536 870 912	29	0.000 000 001 862 645 149 230 957 031
1 073 741 824	30	0.000 000 000 931 322 574 615 478 516
2 147 483 648	31	0.000 000 000 465 661 287 307 739 258
4 294 967 296	32	0.000 000 000 232 830 643 653 869 629
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407
34 359 738 368	35	0.000 000 000 029 103 830 456 733 704
68 719 476 736	36	0.000 000 000 014 551 915 228 366 852
137 438 953 472	37	0.000 000 000 007 275 957 614 183 426
274 877 906 944	38	0.000 000 000 003 637 978 807 091 713
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601

CHAPTER 3

SUBPROGRAMS

3.1 THE USE OF SUBPROGRAMS

Apart from the facility of being able to perform time sharing among independent programs, FP6000 is able, by virtue of EXECUTIVE, to have a program time sharing with different parts of itself. By use of the instruction to EXECUTIVE "Suspend this program if its unit X of type N is busy" a programmer is able to avoid overwriting information that is involved in a peripheral transfer; however, this may lead to the central computer being held up waiting for a transfer when in fact it could be performing useful work. Consider a program which is producing results which are to be printed on a line printer and suppose that, because of the nature of the computation, the results are produced in groups of ten lines none of which can be printed until all are computed. This situation can easily arise in the production of certain types of table. If the straightforward approach is adopted; the sequence of events will occur as follows:

(a) Compute group of results.

(b) Print group of results.

(c) Return to step (a).

Even if it is assumed that there are other programs in the machine which can use the arithmetic unit while the results are being printed, the printer is still idle for part of the cycle, which is not making the most efficient use of the printer. This situation can be avoided by splitting the program into a master program and subprogram which time share with each other. If the core store originally allocated to the program is divided into an area containing the master program, an area containing the subprogram and an area of working store, and if it is arranged that the datum and limit of the master program are such as to include the whole of the core store area and that those for the subprogram just include its own area and the working store area, then the master program can be made to time share with the subprogram. The working store can be divided into two areas, and while one set of results is being computed by the master the previous set is being printed by the subprogram. Because of the time sharing ability of FP6000 no computation can be timed absolutely, relative to a peripheral transfer. It is therefore necessary for the programmer to take suitable precautions so that it is not possible for overtaking to occur. The master program must not be allowed to overwrite old data with new data before the old data have been completely processed. Similar situations could arise with input subprograms.

Three instructions are provided to enable the programmer to organize the time-sharing of his program and subprograms without difficulty. When the master program reaches a point at which it is ready to initiate the subprogram, for example when it has some results which are ready to be printed, an instruction of the form "Activate my subprogram 1, entering at instruction 10" is obeyed. A master program is allowed to have two subprograms. This instruction causes EXECUTIVE to activate the subprogram, set the subprogram's instruction number equal to 10 and enter the time-sharing routine. Note that the subprogram is not necessarily entered at this time, it is simply activated within the priority list. Usually, because the subprogram is using peripheral equipment, the priority of the subprogram will be higher than that of the master. The master program remains active so that it time-shares with its own subprogram, and EXECUTIVE will pass control initially to the part with higher priority.

When the master program has a new set of results to be printed, but wishes to check that the output subprogram is ready for them, an instruction of the form 'Suspend me if my subprogram 1 is active" is obeyed. If the subprogram is still actively engaged in printing the last set of results, the master program will now be suspended to enable the subprogram to catch up. If, however, the subprogram has completed its task, it will have made itself inactive by the instruction 'Suspend me awaiting activation by the master program and release the suspension on the master program if it is waiting for me". Thus, irrespective of whether the master or subprogram completes its task first, a stage is eventually reached at which the master program is informed that the subprogram has finished and is inactive. The master program may re-activate the subprogram immediately, or it may wish to do some preliminary data transfers before initiating the next set of output.

A more sophisticated programmer may improve on the utilization of the printer by dividing his working store into three areas, thus buffering the supply of data from master to subprogram. By means of markers set within the common area, the master and subprogram can communicate their relative positions, and need only enter EXECUTIVE when it is necessary for one or other to be suspended. However, for most purposes the simpler approach will produce a significant increase in peripheral utilization and hence an improved overall system efficiency.

CHAPTER 4

GROUPS 13 - 17 MACRO-INSTRUCTIONS

GROUPS 13 - 17 MACRO-INSTRUCTIONS

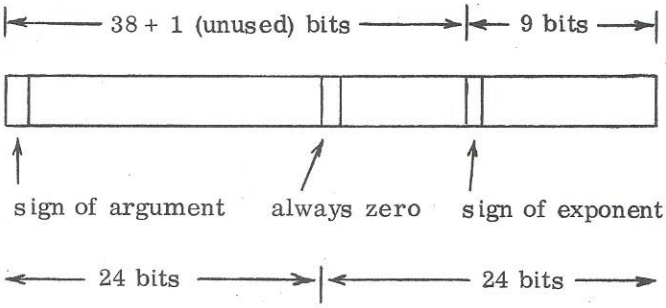
TABLE OF CONTENTS

4.0 INTRODUCTION

As far as the hardware of FP6000 is concerned, the action taken on instructions within the range 130 - 177 is precisely the same. EXECUTIVE is entered at a fixed place and given details of the instruction which cause the entry. EXECUTIVE performs the functions described below by means of routines corresponding to each group.

4.1 GROUP 13 - FLOATING-POINT OPERATIONS

This group is available in the form of either hardware or program within EXECUTIVE. The functions produce the same result irrespective of the machine configuration. A floating-point number is considered to be held in the form of two adjacent words and is divided into a 38 bit argument and a 9 bit exponent. Each part of the number has its own sign digit.



The exponent is held with 256 added to it, so that its "sign digit" is a 1 when the exponent is positive, 0 when negative. There are certain advantages to this form of representation, mainly that floating-point zero is represented as a true zero word in both the most significant and least significant parts, and also that the addition of 256 acts as a guard digit in that the exponent is always represented as a 9-bit quantity, but can readily be formed into a true 24-bit quantity when extracted.

4.1.1 130 Function: Convert Fixed to Floating

This instruction takes a mid-point number in two consecutive accumulators and replaces it in the same accumulators with a binary floating-point number. For example the number +17.92 is held in X6 and X7 so that the integral part (+17) is held in X6 and the fractional part (+0.92) is held in X7. The 130 instruction places in X6 and X7 the true floating-point representation of this number, ie. a fraction 0.56 held in the first 38 bits of the double accumulator and an exponent of 5 in the least significant 9 bits of X7. The guard digit, 256, is added to the exponent. The 130 instruction can never cause overflow.

4.1.2 131 Function: Convert Floating to Fixed

This is the converse of the 130 instruction, in that a standardized floating point number is converted in the same adjacent accumulators to an integral part in the first accumulator and a fractional part in the succeeding accumulator: Overflow will be set if the result cannot be contained.

4.1.3 132-135 Arithmetic in Floating-Point

132 Add the number in X and X + 1 to the number in N and N + 1 and place the result in X and X + 1

133 Subtract the number in N and N + 1 from the number in X and X + 1 and place the result in X and X + 1

134 Multiply the number in X and X + 1 by the number in N and N + 1 and place the result in X and X + 1.

135 Divide the number in X and X + 1 by the number in N and N + 1 and place the result in X and X + 1.

This group of instructions replaces one of the operands, in each case the operand in the specified accumulators, by the result of the arithmetic operation.

4.1.3 continued

Example 1:

Three floating-point numbers A, B and C are contained in consecutive pairs of locations labelled NUMA, NUMA + 1; NUMB, NUMB + 1; NUMC, NUMC + 1. It is required to store in RSLT, RSLT + 1 the mid point version of

$$A + \frac{BC}{B - C}$$

```
000   6   NUMB        ┐
                      ├─ Transfer B to X6 and X7
000   7   NUMB + 1    ┘
134   6   NUMC        Form BC in X6 and X7
000   4   NUMA        ┐
                      ├─ Transfer A to X4 and X5
000   5   NUMA + 1    ┘
133   4   NUMC        Form A-C in X4 and X5
135   6    4          Form BC/(A-C) in X6 and X7
132   6   NUMA        Form A + BC/(A-C) in X6 and X7
131   6    0          Transform to fixed point
074   1   ERR         Branch to ERR if overflow occurs
010   6   RSLT        ┐
                      ├─ Write result to RSLT
010   7   RSLT + 1    ┘
```

Example 2:

One hundred floating-point numbers are held in a table starting at location TABL. Write to TOTL the sum of the 100 numbers, assuming that overflow will not occur.

```
124   2    100        Set a count of 100 in X2
033   0     6         ┐
                      ├─ Clear X6 and X7
033   0     7         ┘
132   6   2 TABL      Add in one number
062   2     T - 1     Repeat 100 times
010   6     TOTL      ┐
                      ├─ Write X6 and X7 to TOTL
010   7     TOTL + 1  ┘
```

4.1.4 136 Function: Floating-Point Square Root

This instruction replaces a floating-point number in the specified accumulators by a floating-point version of that number's square root.

Example:

The location VAR contains a number whose square root is to be written to SD.

```
000   4   VAR      ┐
                   ├─ Place the number in X4 and X5
000   5   VAR + 1  ┘

136   4   0        Replace X4 by the square root of the number

010   4   SD       ┐
                   ├─ Write the square root to SD
010   5   SD + 1   ┘
```

4.2 GROUP 14

This set has been left as a contiguous group of instructions whose meaning is not defined. An individual installation may wish to use this group as an integral part of EXECUTIVE to perform functions not included in the standard list. This practice is not recommended since the value of interchange programs is then restricted, but in the case of semi-special-purpose installations this may be considered a worthwhile limitation.

4.3 GROUP 15

This group of instructions is concerned with the communications between a program and EXECUTIVE whereby the program may receive information and request action of EXECUTIVE regarding the state of its peripherals and the general action required by the program.

4.3.1 150 Function:
Suspend me if my Unit X of Type N is busy

Each programmer is constrained to number the peripherals of each type sequentially from zero. Thus, if a program uses four magnetic tape units, then these are numbered 0, 1, 2 and 3 and reference to any other unit number in a magnetic tape statement will be treated as an illegal instruction.

The peripherals themselves are classed by types as follows:-

TR	Paper tape readers
TP	Paper tape punches
LP	Line printers
CR	Card readers
CP	Card punches
MT	Magnetic tape units
DM	Drums
TY	Special devices including typewriters

Thus "my unit 6 of type MT" is in fact the 7th magnetic tape unit used by an individual program.

The 150 instruction causes EXECUTIVE to test the current state of the named peripheral unit and, if it is still actively engaged in the execution of a transfer, the program requesting the information is suspended. This suspension is released as soon as the peripheral transfer has been completed and checked by EXECUTIVE. If the unit in question is not busy, then an immediate return is made to the program.

4.3.2 151 Function: Release my unit X of type N

This instruction tells EXECUTIVE that the named peripheral is no longer required by the program and may be freed, to be subsequently available for use by any other program. For instance, a magnetic tape sorting program may require the use of card reader for parameter information prior to starting the actual sort. As soon as this information is supplied, the card reader is no longer required but the program itself may be of considerable duration. It would be unwise in these circumstances not to permit the use of the card reader for other purposes, and the 151 instruction is therefore provided.

4.3.3 152 Function:
Disengage my unit X of type N

This instruction should not be confused with the 151 instruction since it does not make the peripheral unit in question available to another program but merely switches it "off-line" (EXECUTIVE records this event on the typewriter) so that any further attempt to use it will result in typed messages to the operator. The peripheral unit can only be brought back "on-line" as the result of some operator action. The purpose of the instruction is to inhibit the use of a peripheral until the operator has taken some action on it - such as a change of paper in the line printer or the replacement of a tape in the paper tape reader.

4.3.4 Function 153: Place in my register 9 the control register of my unit X of type N

Since peripheral transfer instructions are of variable length, it may happen that a program requests the transfer of information in a quantity greater that that which actually exists on the input medium. For instance the block length on tape may be variable, in which case the program may request a transfer of "a block or 512 words, whichever occurs first". If the block is completely read-in and does not contain a full 512 words then the control register for that unit is set to the last active state achieved during the transfer. A control register consists of two parts - an address and a count (this is explained in detail in Section 4.5). As information is passed to the computer during a read operation, the address is incremented and the count is reduced. This process stops when the transfer is completed and, from the state of the control register is not accessible directly to program but is made available through EXECUTIVE by means of the 153 instruction.

4.3.5 154 Function: Read more program from my unit X of type N

Facilities are provided for programmed interludes so that part of a program may be read, some of it obeyed, and then a further section of program used to overlay or supplement the original portion. The instruction may only be used when the program is being read under the control of EXECUTIVE and in the format suitable for EXECUTIVE's binary reading routines.

4.3.6 155 Function:
Suspend and Dump on my unit X of type N

This instruction provides a facility for dumping an entire program, together with its accumulators and all working spaces, onto some external medium in binary form suitable for re-input. It is not intended to provide debugging facilities but merely a convenient method of organizing re-start procedures.

4.3.7 156 Function: Abolish this program

This is intended to be the last instruction obeyed by a program. It is a request to EXECUTIVE to remove it from all lists and make the core store space it occupies and all its peripherals available to any other program. Because of the drastic nature of the instruction in a time-sharing system, EXECUTIVE does not immediately abolish the program but requests confirmation of the instruction from the operator. EXECUTIVE types a message saying

ABOLISH ? # JACK N,n

where N is the numeric quantity specified by the programmer in the N-field of the instruction.

The operator then chooses whether to confirm the abolish query or instruct the program to resume at some restart point.

4.3.8 157 Function: Suspend this program awaiting operator message to EXECUTIVE

This instruction is similar to the 156 instruction except that the operator is expected to take some action or make some decision regarding the continuation of the program.

4.3.9 General

Any attempt by the program to address a non-existent or non-reserved peripheral or to execute an instruction which is not apparently logically sound, will cause a suspension of the program at the point of the offending instruction and a message will be typed on the typewriter in the following form

ILLEGAL 0JACK 2032

thereby identifying the location of the illegal instruction and the particular section or subprogram concerned.

4.4 GROUP 16

This group is a continuation of Group 15 and in general is concerned with the interplay between the various branches of a program. Chapter 3 outlined the necessity or desirability of subprograms and the instructions of group 16 are designed to implement these. Subprograms are sometimes referred to as AUTOROUTINES to avoid confusion with the concept of "subprograms" as used in Fortran.

4.4.1 160 Function: Suspend this program awaiting operator message and type my message on the typewriter

This instruction is essentially the same as the 157 instruction except that EXECUTIVE types the message defined by the control word, whose address appears in the N-field of the instruction. This instruction may be used by the program to indicate some condition which has arisen in the program and which requires some decision or action on the part of the operator. The 160 instruction is the only means of direct communication between a program and the operator's typewriter and the program is always suspended when the instruction is used.

4.4.2 162 Function:
Suspend me if my subprogram X is active

When a master program wishes to activate one of its sub-programs or finds it can no longer proceed if the subprogram is active, it may voluntarily suspend itself. The subprogram number (1 or 2) is specified in the X field and zero is specified in the N-field.

4.4.3 163 Function: Activate my subprogram X entering it at the instruction in location N

A subprogram may only be executed when called upon by the master program. When the complete programs are read in by EXECUTIVE, the subprograms are rendered inactive pending activation by the master. The 163 instruction is treated as a call from the master to place the subprogram in the "active" list and does not necessarily imply that the subprogram is immediately entered. The actual program which EXECUTIVE selects to obey after placing the subprogram in the active list is determined by the priority assigned to each entry in the active list. The master program is free to continue operation after obeying the 163 instruction.

4.4.4 164 Function: Suspend this subprogram and release the suspension on the master program if it was held up pending the completion of this subprogram

This instruction represents the only means of exit from the current subprogram, which can then be re-entered only by the use of a 163 instruction in the master program. Subprograms are normally associated with the use of peripherals in a program whose input-, computing-, and output-times are reasonably in balance.

In the example overleaf, the master program is preparing 200 characters (50 words) for punching, but the output is to be limited to 20 characters per card. It is assumed that the first card cannot be punched until the full twenty have been prepared. The master program transfers the information to the subprogram's area so that it may continue and prepare the next set for output. The Subprogram punches one card at a time and is suspended for long periods between the card punching cycles. However, this suspension applies only to the subprogram and the master is in the meantime free to continue computation. As soon as 10 cards have been punched, the subprogram suspends itself and can only be entered again if the master chooses to activate it.

Example:

Excerpt from Master Program

	F	X	M	N	Notes:
	162	1		0	Suspend if Subprogram 1 is active
	100	6		ADD 1	Set addresses for block transfer
	100	7		ADD 2	
	126	6		50	Transfer 50 words
	163	1		20	Activate Subprogram 1 entering it at instruction 20
	074	0		STRT	Resume computation

Excerpt from Subprogram 1

	F	X	M	N	Notes:
#20	124	1		10	Set count of 10 in X1
	174	0	1	CW1	Punch 1 card with 20 characters
	150	0		CR	Suspend if CR busy
	060	1		T-2	Select next control word and punch another card
	164	0		0	Suspend until re-activated

4.5 GROUP 17 PERIPHERAL TRANSFERS

This group refers to peripherals which are divided into two categories, known as slow and fast:

Character-at-a-time peripherals, including:-

Paper tape readers
Card readers
Paper tape punches
Card punches
Line printers
Typewriters

Word-at-a-time peripherals, including:-

Magnetic tapes
Magnetic drums
Magnetic discs

The format of the peripheral transfer instructions is identical in each case, and specifies the unit number on which the transfer is to be effected (Note: this is the programmer's number, not the absolute number) and the address of a control word to be used to effect the transfer.

The instruction as written by the programmer is as follows:-

Function Unit No. Address of Control Word

The address specified in the N-field may, of course, be indexed.

4.5.1 Control Words

The control word specifies two quantities:

(i) The number of words or characters

(ii) The address in store at which the transfer is to be started.

Thus a magnetic tape read instruction, for example:

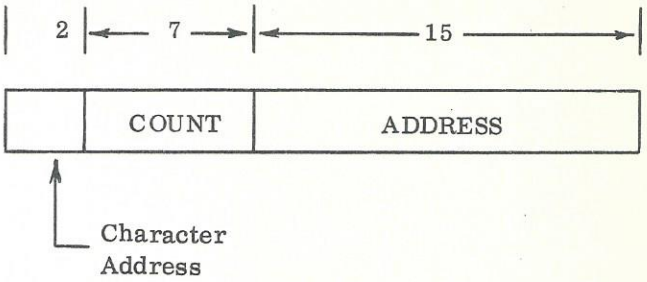
175 2 CONT

will perform a read operation on the programmers magnetic tape unit #2. The address 'CONT' contains two quantities:

Number of Words/Starting Address in Store

The control word is divided in the way described in Chapter 2, whereby 9 bits are used for the count and 15 for the address.

Character-at-a-time peripherals also require a "character address" and, consistent with the specifications in Chapter 2 for character indexing, the control word in this case is divided as follows:-



The character address bits specify at which character position the transfer is to start, the character positions being as follows:

Char 0	Char 1	Char 2	Char 3

4.5.2 Modes

All peripheral transfer instructions may be performed in a variety of ways; for instance, a card reading operation must specify whether the reading is to be in binary or binary coded decimal. The modes applicable to each type of peripheral are described individually in the succeeding paragraphs. A mode remains set until it is re-set; thus it is only necessary to specify the mode at the start of the program if that same mode is required for all peripheral transfers in the program. Mode is set by means of a 125 instruction as described in Chapter 2.

4.5.3 170 Function: Read Paper Tape

A paper tape reader is a character-at-a-time device and as such requires the control word to specify a character count and a character address within the control word. The standard paper tape code uses 7 channels for information and an additional channel for even parity checking. The 7-bit character is transformed in the paper tape reader control unit to a 6-bit form. The various character representations are described in detail in Chapter 8. Because of the transformation it is necessary to introduce "Shift" characters for the internal representation of all 128 external tape characters. These shifts are designated as α, β and δ shifts and are fully described in Chapter 5.

4.5.3 continued

The programmer's count of the number of characters to be read is based on the INTERNAL representation and thus the programmer must be aware of the format on the tape so as to allow for additional shift characters as necessary. In practice α-shift contains all the commonly-used symbols, numbers, and letters; whereas the β and δ shifts are concerned mainly with control and transmission characters.

There are several modes in which paper tape may be read. Six of the seven mode bits may be used in any combination and the effect of each bit is described as follows:-

Bit	State	Effect
1	0	Read N computer characters.
	1	Read N computer characters or to Newline.
2	0	Read starting in α-shift.
	1	Read starting in previously established shift.
3	0	Read BCD mode, inserting control characters as required.
	1	Read binary mode. Note: Newline is outside the binary set and hence may be used as a stop character.
4	0	Ignore erases and blank tape. TC4 (Transmission Code #4) switches reader off-line. TC3 causes a skip to the next TC1.
	1	All codes are entered.
5	0	No effect.
	1	Skip to TC1 then start the transfer.

These bits are arranged in the form of a binary word as follows:-

5	4	3	2	1	0

Thus mode 4 is represented by 000100.

Any combination of modes may be used. Thus, mode 50 (or 110010) implies that the reader is to skip to the next TC1 character and thereafter read all characters into the machine. Shift characters will be inserted as necessary, and reading will continue until N computer characters have been read or a Newline character is encountered.

4.5.4 171 Function:
Punch Paper Tape

The paper tape punch instruction is very similar to the 170 paper tape read instruction, but does not require the number of modes used in reading. The control word is set up in precisely the same way as described in 4.5.3 and specifies the number of computer characters to be punched and the address from which the transfer is to start.

Five of the mode bits are used.

Bit	State	Effect
0,1		Not used, no effect
2	0	Start punch in α-shift.
	1	Start punch in previously established shift.
3	0	Punch N computer characters. (Note: the shift characters are obeyed by the punch control and do not themselves appear on the output tape).
	1	Punch N computer characters in binary mode. The control codes are all punched and all other characters are transformed as for α- shift.
4	0	Punch data.
	1	Punch N run-out characters (blank tape or TC0 codes).

As with the tape reader modes, any of the above modes may be added together to produce the desired effect.

4.5.5 172 Function: Line Printer

There are no modes associated with the operation of the line printer (except in the case of the 160-character wide printer). The standard printer has 120 print positions.

The control word used for the printer is similar in format to that used for paper tape operations. However, the count in the control word is one greater than the number of characters required to be printed, as the first character in the output area is a carriage control character.

The instruction is in the form

172 X N

where X is the relative number of the printer and N is the address in core store of the control word.

The carriage control characters are as follows:

If the fourth bit is a zero, ie., 000XXX, then the lower 2 bits are decoded thus:

0 - Single Line Feed

1 - Double Line Feed

2 - No Line Feed

Associated with the line printer is an 8-channel paper tape loop called the control tape. This tape has a character punched for each line on the printer form. When the control character has fourth bit zero the control tape is advanced one character for each line feed (0, 1 or 2 as the case may be). If, however, the fourth bit is a 1, ie., 001XXX, then the lower 3 bits are decoded as a channel number for the control tape (channels are numbered 0 - 7). The tape is then advanced to the next hole in the specified channel and the printer is made to line feed once for each character the tape moves. By convention, channel 0 is used for Form Feed so that, whatever position the printer is in, the printer and tape will advance to the top of the next form.

4.5.6 Function 173: Read Card

The instruction is in the form

173 X N

where X is the relative number of the card reader and N is the address of the control word. Only the 15 bit word address in the control word is significant. Regardless of the contents of the most significant 9 bits of the control word, 80 columns will always be read from the card, and will be set into the area starting at character 0 of the address specified.

Two modes are available, and these are specified by the 125 (set mode) instruction prior to the card reading instruction. The modes are:

0 - Read BCD, ie. read 80 columns into 80 character positions, converting from the external 12 bit representation to the standard internal 6-bit representation. (See Chapter 5, Table 3).

2 - Read Card Image, ie. read each column to successive half-words performing no conversions. The top of the card appears at the most significant end of the half-word.

4.5.7 Function 174: Punch Card

No modes are applicable to the card punch and the conversion is made from standard 6-bit internal form to the standard 12-bit (1 character per column) external form. If a routine were required by an individual installation to produce a code other than the standard external representation, then such conversion would be mode controlled, but this is not part of the standard FP6000 system.

The instruction is in the form

174 X N

where X is the relative number of the card punch and N is the address of a control word. The counter may be specified in the same way as for paper tape and partial punching is permitted. The starting address of the area to be punched may specify any individual character within a word.

4.5.8 Function 175: Magnetic Tape Operation

The operations described here with reference to magnetic tape explain the interface between the programmer's program and Executive. In general, all magnetic tape operations are in the form

175 X N

where X is the relative number of the tape unit and N specifies the starting address of a control area. The 125 (set mode) instruction is not used. Instead the mode is specified as the first word in the control area.

4.5.8.1 Reading & Writing

For normal read and write operations the control area consists of 4 words as follows:-

(1) Mode

(2) Reply Information

(3) Number of Words

(4) Address in Store

The modes applicable to reading and writing are as follows:-

0 - Read Binary (odd parity)
1 - Write Binary
8 - Read BCD (even parity)
9 - Write BCD

In BCD mode the character 'octal 74' is used as a stop character and writing operations cease when this character is encountered in the output area. Similarly, reading in either mode results in the last word being filled out with the same stop character and the next word being filled with stop characters if the area specified by the programmer permits this.

Reply information is placed in the second word of the control area by EXECUTIVE during and after the transfer of data. The word is held negative by EXECUTIVE during the transfer and set to zero when the transfer is completed correctly. If End of File, End of Tape or a long block is encountered, EXECUTIVE places a small positive integer in the reply word as follows:-

End of File	64
End of Tape	1
Long Block	16

The number of words specified in the control area must be less than 32768.

4.5.8.2 File Handling

EXECUTIVE handles the tape system by means of dynamic allocation, which means that tape units are allocated to the programmer when they are required during the running of a program.

This feature is used by the programmer when an instruction is written using one of the following three modes:-

Open file and check label	128
Open scratch tape and label it	384
Open scratch tape	256

In some cases the programmer must allow for a 9-word area which consists of one mode word, one reply word and 7 words associated with the label. The nine word area takes the form:

Word 1	Mode
2	Reply information
3 - 5	Programmers identification
6	Reel number
7	File serial number
8	Retention cycle
9	Date written

In mode 128, the programmer specifies the name of the file (3 words) and the reel number (one word). When Executive has found the appropriate file, it supplies the file serial number (one word), the retention cycle (one word) and the date on which the file was written (one word).

In mode 384, the programmer is requesting Executive to find a scratch tape (one which is out of date) and label it with the information supplied in words 3 - 9.

Mode 256 uses only the first 2 words of the specified area, since no label is required.

4.5.8.3 Simple Operations

Further modes of tape operation cover the simple actions required of EXECUTIVE by the programmer. Each operation requires the address of a 2-word control area, the first of which is MODE and second is reply information. The modes are as follows:-

Backspace	2
Forward to End of File	4
Write End of File	5
Back to End of File	6
Rewind	7
Close File	519

The reply information is precisely the same as that for normal reading and writing operations.

CHAPTER 5

INTERNAL AND EXTERNAL CODES

TABLE OF CONTENTS

CHAPTER 6

EXECUTIVE-OPERATOR COMMUNICATIONS

TABLE OF CONTENTS

6.1 INTRODUCTION

EXECUTIVE is a special program which, when the computer is in normal use, is always present in the first few hundred words of core store, and which controls the other programs in the system at any time.

The main functions performed by EXECUTIVE are:

(1) Control of peripheral transfers and provision of information about peripherals.
(2) Communication between the operator and any individual program.
(3) Allocation of the time of the central processor among programs, so that even the slowest peripherals do not cause wastage of time or require expensive buffering.
(4) Monitoring.
(5) Macro-instructions.

This chapter is principally concerned with the second of these functions. The macro-instructions, including the peripheral transfer group, were described in Chapter 4. EXECUTIVE performs tests on these macro-instructions, and any instruction which fails will cause the program to be suspended and a message to be typed out, eg.,

ILLEGAL 1JACK 200.

where 1JACK means subprogram 1 of JACK, and 200 is the instruction number of the illegal instruction. Such an instruction might, for example, ask for a transfer on a peripheral not assigned to the program, or attempt to use store locations outside the program's area. A similar message will be typed if a program attempts to obey an instruction with an unassigned function code, such as 066.

In the event of a peripheral failure, such as a parity failure on a Paper Tape Reader, EXECUTIVE will be informed. The program which was using this peripheral is suspended, and a message is typed out.

FAIL UNIT A #JACK.

In the event of a condition requesting the operator's attention, a message will be typed.

FIX UNIT A #JACK.

Such events as a card reader being disengaged or the card punch hopper getting low in cards will cause such a message to the operator.

6.2 COMPUTER-OPERATOR COMMUNICATIONS

The operation of the system is controlled externally by the operator and internally by EXECUTIVE. All communications between the two take place via the console typewriter. In addition to the typewriter keyboard, the operator has a set of three control buttons, labelled as follows:-

INPUT
ACCEPT
CANCEL

In order to inform EXECUTIVE that the operator wishes to type a message into the computer, the INPUT button must first be pressed. If EXECUTIVE is able to accept an input message, ie., if the last one has been dealt with, and if the typewriter is not required for an output message - then the INPUT button lights up and the keyboard is unlocked. The operator may now type the message, which must be one of the standard messages described below and must be typed according to the format rules. When the message is complete, the operator must check it for accuracy. If the message is at fault in any way, the CANCEL button must be pressed. EXECUTIVE will turn off the INPUT light, and type 'NO' beneath the cancelled message, to indicate that the message was not acted upon. The operator may now start the whole process again by pressing the INPUT button, as above. When the operator is satisfied that the message has been typed correctly, the ACCEPT button is pressed. This causes the INPUT light to go out, and EXECUTIVE now examines the message.

If EXECUTIVE is satisfied that the message is a standard one with correct format, 'OK' will be typed below the message, and the instructions contained therein will be carried out as described below for each individual message. If EXECUTIVE is not satisfied with the format, again 'NO' will be typed beneath the message, and the operator must examine the original request, and if required try again.

Any inappropriate operation of the console buttons will cause a 'NO' to be printed, and the Input light to be switched off if it is on. For example, pressing ACCEPT before INPUT or pressing INPUT twice will be treated as operator-errors. If, while a message is being typed, the typewriter is required by EXECUTIVE for an output message then EXECUTIVE takes priority; the INPUT light will go off, the keyboard will be locked and the message being typed in will be ignored.

6.3 MESSAGES TO EXECUTIVE

The individual messages which the operator may use are listed in the table below and are described in more detail in following subsections.

Certain rules apply to all messages, these are:-

(a) Every message must begin with a "newline" or "line-feed" character.

(b) The first word of the message must be spelled correctly, and it is essential that the first letter follows immediately after the "newline" character.

(c) The message must end with the conventional punctuation symbol . to signify end-of-message.

(d) Program names must be introduced by # and the 4 characters of the name must follow immediately, thus; #JACK.

(e) Spaces are optional between words, names and numbers.

(f) Messages requiring more than one number must give the numbers in the correct sequence. The format of the messages should be followed exactly, though spaces and additional text may be inserted.

The format of each message is exactly as written in the following paragraph headings.

TYPEWRITER INPUT MESSAGE SUMMARY

LOAD #JACK ON X.	Read a new program, named JACK, from unit X.
LOAD #JACK ON X, Y.	Read a new program, named JACK, from unit X but assign Y words of core store, ignoring the store request of the program.
GO #JACK.	Start obeying program JACK at the instruction number contained in its instruction number register.
GO #JACK AT 3.	Start obeying program JACK at instruction number 3.
CONTINUE #JACK ON 15.	Read more program for program JACK from unit 15.
ALTER 1#JACK AT 200, 000 3 2 1024.	Alter instruction 200 of subprogram 1 of JACK to the instruction specified.
SUSPEND #JACK.	Suspend JACK (master and relevant subprograms) awaiting a message from the operator.
DELETE #JACK.	Delete - or abolish - program JACK from EXECUTIVE lists and from the core store. Release peripherals.
MONITOR #JACK ON 4.	Read monitor steering information for program JACK from unit 4.
PRIORITY PRINT.	Print out current contents of the Priority List.
REVISE PRIORITY 0#JACK 80.	Change JACK's priority to 80.

6.3.1 LOAD #JACK ON 15.

Every program which is to be run on FP6000 must have been prepared originally by a program scheme such as ASSEMBLER. Chapter 7 describes how ASSEMBLER produces an assembled program tape in the form required by EXECUTIVE. In the following paragraphs any reference to a program means the assembled program produced by ASSEMBLER.

When a new program is to be loaded, EXECUTIVE must be informed of the fact. The operator will load a free tape unit with the program tape, then input the message

LOAD #JACK ON 15.

where 15 is the number of the unit containing the program. This will usually be the lowest-numbered unit of the appropriate type which is free at the time.

On receiving this message, EXECUTIVE first checks that unit 15 is free and that there is not already a program of the same name in the system. Provided these conditions are satisfied, EXECUTIVE reads the preliminary section of the program, which provides information about the core area and the types and numbers of peripherals required. EXECUTIVE checks that all these requirements can be accommodated, and if so, a list of the peripherals assigned to the program will be typed. For example, if the program requires 2 tape readers and 1 punch, the message will read

PROG	JACK
TR0	4
TR1	6
TP0	9

showing that units 4, 6 and 9 have been assigned to the program. The program will now be read in and stored. This operation takes place within the time sharing system. When all the program has been read, it is suspended and a message is typed

INSTRUCT #JACK

indicating that the program is ready to be entered. If any of the program's requirements cannot be met, the LOAD message will be answered by 'NO' and the program will have to wait until it can be accommodated.

6.3.2 LOAD #JACK ON 15, 4500.

This alternative form of the LOAD message enables the number of core store words specified by the program tape to be replaced by the number 4500. This is a useful facility for such programs as matrix schemes, where the storage requirement varies considerably, depending on the data to be used on a particular run.

6.3.3 GO #JACK.

When a program has been loaded and the message INSTRUCT #JACK typed out, the program may be initiated by the GO message. This will cause the program to start being obeyed at the entry point specified by the programmer.

An INSTRUCT request may also be received during the running of a program, since the program may contain a 157 instruction, which would request operator action of some sort.

The GO message may also be used to restart the program under these circumstances, provided that it is required to continue from the point of suspension. The GO message may similarly be used to start following a 160 instruction, which types the program's own message out instead of the INSTRUCT message.

6.3.4 GO #JACK AT 10.

10 specifies the entry point at which the program is to be initiated. This version of the GO message enables a program to be restarted from a chosen restart point after, say, a peripheral failure. It also allows the operator to override the program entry points originally specified by the programmer.

6.3.5 CONTINUE #JACK ON 15.

The CONTINUE message instructs EXECUTIVE to read more program from unit 15. Thus a program may be partitioned into two, or more, sections which do not need to be in the store at the same time. The overall store requirement of the program can thereby be reduced. The CONTINUE message should only be used with reference to a program which has been suspended. It is very likely to be used after a 157 instruction in the program has caused an INSTRUCT to be typed out. EXECUTIVE, of course, first checks that unit 15 is free and belongs to program JACK.

6.3.5 continued

Another use of the CONTINUE message is to restart the reading of program after EXECUTIVE has indicated a failure of the section just read; either a parity or check-sum failure. Since each section of a program carries its own transfer address, reading may be resumed by means of a CONTINUE message.

6.3.6 ALTER 1#JACK AT 200, 000 3 0 1024.

The use of the ALTER facility should be strictly limited to changing a few instructions during a development run. The contents of JACK's location 200 are replaced by the instruction specified. All the digits of the instruction must be typed and the F, X, M and N fields separated by a space on amendment tape by ASSEMBLER.

Note that #JACK is preceded by the appropriate subprogram number. For the Master program, the correct format is

ALTER 0#JACK etc.

6.3.7 SUSPEND #JACK.

In the event of an emergency, such as the program apparently having got out of hand, the operator may wish to halt the operation of the program in order to ascertain what has taken place. The SUSPEND instruction does not destroy the program, so that is is possible to restart if this is found to be desirable.

6.3.8 DELETE #JACK.

This message has the effect of confirming the 'Abolish' instruction written into a program. All details of program JACK are removed from EXECUTIVE's records. The peripherals which had been assigned to JACK are released, and the core store area is made available for another program to be read.

6.3.9 MONITOR #JACK ON 4.

The MONITOR message is fully described in Chapter 8 on the Monitor program. Again EXECUTIVE will check that unit 4 is free or belongs to JACK or the Monitor program. The instruction causes the Monitor program to read information regarding the Monitor points to be set up in a program.

6.3.10 PRIORITY PRINT.

This message will be answered, not by 'OK' but by a listing of the programs and subprograms currently in the system. They will be typed in priority sequence, and their priority rating will be printed immediately following the name. Master programs are typed as 0#JACK, etc. The list will appear as follows:-

2JACK54
0FRED48
1JACK27
0JACK20

6.3.11 REVISE PRIORITY 1#JACK 65.

The operator may adjust the sequence of programs in the priority list, perhaps to accelerate a job which has become urgent, or in an attempt to improve the priority allocation. 65 is the number to which the priority of JACK's subprogram 1 is to be raised. Only one priority list entry may be changed per REVISE PRIORITY message. If EXECUTIVE accepts the message, it will type the list out in reply (instead of 'OK').

There are no hard-and-fast rules for assigning priorities to programs, though the general plan is to give high priority to programs using slow peripherals and low priority to programs using a lot of computing time and little peripheral equipment. Individual FP6000 operators will quickly find which combinations of regular programs time-share best, and by means of the REVISE PRIORITY message some experiments can be made until the most satisfactory allocations have been found.

CHAPTER 7

ASSEMBLER

CHAPTER 7

TABLE OF CONTENTS

INTRODUCTION

ASSEMBLER can operate equally well with either punched paper tape or punched cards; the description in this section uses the terminology of the paper tape version.

The function of ASSEMBLER is to convert the source program, which is written in a form convenient for the programmer, into the assembled program. The source program is written as a succession of program lines, each terminated by the character "newline"; these lines are read one at a time by ASSEMBLER and translated into a sequence of FP6000 words. Certain types of program lines, called stored words, are converted directly into FP6000 words; however, this must not be taken to imply that the assembled program can differentiate between instructions and constants, it is still the programmer's responsibility to ensure that no attempt is made to obey stored words which are not instructions.

ASSEMBLER provides the ability to refer to the addresses of instructions or constants by codeword names.

7.1 THE SOURCE PROGRAM

7.1.1 Codewords

Codewords are written in a program in places where it would be difficult to specify an exact value at that stage; they can also make the program more meaningful to read. The most common application is that of symbolic addresses for storage locations (see Section 2.1.4) when absolute addresses need only be allocated after the whole program has been written.

A codeword consists of up to four alphanumeric characters of which the first must be alphabetic, eg:

FILE
R732
Q

Certain codewords have special preassigned values and should not be defined otherwise; their uses are described later in this section. These codewords are as follows:-

	T	C	CP
		L	CR
		A	LP
		S	MT
Codeword			TP
List:			TR
			TY

The number of additional codewords is fixed at 150.

7.1.2 Octal and Single Length Decimal Integers

A Single Length Decimal Integer must lie in the range -8,388,608 integer +8,388,608. It must be preceded by + or -. Spaces or a tab are permitted in the format before the sign or the asterisk, but no spaces are allowed elsewhere. Non-significant zeros will be ignored.

7.1.2 continued

Examples of permissible decimal integers are:

```
-09342
-27
+168
```

An Octal Integer directly defines a binary pattern and is always positive. It consists of up to eight octal characters which are preceded by * . An example of the format, which may contain spaces or a tab preceding the * but not elsewhere is:

```
*77342
```

This is equivalent to the pattern:-

000 000 000 111 111 011 100 010

7.1.3 Compound Expressions

A compound expression consists of any combination of:

codewords
single length integers
octal integers

These are connected together algebraically with the symbols , or + for addition and - for subtraction.

```
769-*77
FILE, 10 -Q +ITEM
```

Spaces may occur immediately before or after a separating symbol and will be ignored. The intermediate results of the summation of the elements can always occupy 24 bits, although overflow must not occur at any time, but the final value of the codeword expressions has a limit which depends on the context.

7.1.4 Codeword Definitions

When a program has been completely written, and is about to be assembled, all the codewords used must have their numerical values defined before they are reached in the program; with this provision the definitions can be located at will. Whenever ASSEMBLER encounters a codeword, this is replaced by its defined value. The value of a codeword is defined by a program line containing codeword and value, separated by the symbol =. The value, which may be positive or negative, is written as a compound expression.

Thus a codeword does not have to be defined absolutely and acceptable definitions are, for example:-

```
FILE = 769
ITEM = -10
   Q = ITEM, FILE +*77
```

Since FILE and ITEM are defined by the first two program lines, the last line will define Q as -10 + 769 + 63 = 822. The value of a compound expression in a codeword definition is allowed to occupy 24 bits.

7.1.5 Transfer Address Codeword, T.

T, the Transfer address is the address of the location in which the current stored word will be placed. The value of T depends on the context; if used in an instruction it is the address of the store which contains this instruction, but used in a codeword definition it is the address of the next word to be assembled as part of the program. T is the most commonly used codeword and is unique since its value is continually changing; it is automatically incremented by unity after the assembly of each word of program and may be redefined by the programmer at a new value at any point in the program.

There are four main applications of T:-

(a) To simplify branch instructions. T is used as part of a compound expression in the N-address in order that the branch address can be defined relative to the actual branch instruction. Thus the following instruction will cause the program to branch to an instruction 3 words ahead and hence omit two words.

```
074    0    0        T + 3
```

(b) To dictate where in the program's core store area the various program words are to be stored. Thus the codeword definition of T (shown below) will cause the following program words to be stored from location 256 onwards. Note that if T is not otherwise defined at the start of a program, storage commences at location 20.

```
T = 256
```

(c) To leave working space internally in a program by advancing the transfer address. Thus the following definition will leave 20 words clear.

```
T = T + 20
```

7.1.5 continued

(d) To label points in the program with codewords. For example, for programming convenience, when a program is written in sections, access to these sections is facilitated by defining the name of the section as the starting transfer address.

SEC2 = T

Thus, the definition of the name, as above, enables subsequent branches to be made to the tenth instruction of SEC2 by using the following instruction:-

074 0 0 SEC2 + 9

7.1.6 Shift Codewords, C, L, A, S.

These are used to facilitate the writing of the N-address of shift instructions and correspond to the N_t bits for the four types of shift, viz:

Type of Shift	N_t Value	Codeword	Codeword Value
Cyclic	0	C	0
Logical	1	L	1024 (*2000)
Arithmetic	2	A	2048 (*4000)
Special	3	S	3072 (*6000)

For example, X6 will be right shifted logically 10 places by the instruction:-

112 6 0 L,10

7.1.7 Peripheral Unit Codewords, CP, CR, LP, MT, TP, TR, TY.

These are used to facilitate the writing of the N-address of those group 15 functions which contain the phrase "Unit X of type N". They are defined as:

Peripheral Type	Codeword	Value
Tape Reader	TR	0
Tape Punch	TP	1
Line Printer	LP	2
Card Reader	CR	3
Card Punch	CP	4
Magnetic Tape Unit	MT	5
Peripheral Typewriter	TY	7

For example, the instruction "Disengage my tape reader 3" could appear as

152 3 0 TR

7.1.8 Stored Words

Program lines taking one of the six forms, (a) to (f) each generate a corresponding word which is assembled and stored as part of the assembled program.

These forms are:-

(a) A Single Length Decimal Integer.

(b) An Octal Integer.

(c) A Single Length Decimal Fraction.

(d) An Instruction.

(e) A Counter and Index.

(f) A Compound Expression Constant.

(a) and (b): These have already been defined in section 7.1.2.

(c) A Single Length Decimal Fraction must contain a decimal point, which must be preceded by + or -. Spaces are permitted preceding the first character but nowhere else in the format. A maximum of 13 decimal digits may follow the decimal point but ASSEMBLER rounds the value off to just less than 7 digits so that the result is single length.

Examples of permissible decimal fractions are:

+.375
-00.1697
-.0076984321

(d) An Instruction consists of the sequence F X M /N.

F is punched as three octal digits
X is punched as one octal digit
M is punched as one of the digits 0, 1, 2, 3
/N is punched as a compound expression and is always preceded by an oblique stroke.

Spaces are allowed between F and X, X and M, and M and / but N must follow immediately after the oblique stroke. F, X and N must always be punched but M can be omitted if it is zero; in this instance / acts as an identifying marker for N. The final magnitude of the compound expression in the N-address of an instruction must not exceed 4095 for all functions except those of groups 05, 06 and 07, where the maximum is 32,767; in these three groups no odd-numbered functions are permitted and M must be omitted or punched as zero.

7.1.8 continued

(e) A Counter and Index consists of a compound expression as counter, separated from a compound expression as index by /.

The counter/index pair must be enclosed within brackets. As explained previously in Section 2.10, the most significant 9 bits of a counter/index word are available for the counter portion thus limiting the magnitude of the counter to 511. The index portion in this context occupies 15 bits and limits the final value of the compound expression defining the index to 32,767. No spaces are allowed before the oblique stroke.

An example is: (276/FILE + 20).

To enable character counters to be set, the final element of the compound expression may be terminated by the fraction .0, .1, .2 or .3 and the number after the point is then stored in bits B0 to B1 of the counter and index; this restricts the counter not to exceed 127 and allows the index portion to occupy 17 bits in all.

Examples are: (CT/LINE .1)
(88/FILE + 0.3)

The fraction must always be part of the final element of the compound expression (it cannot appear without an integral part) and the fractional part is always added irrespective of the sign of the last element as a whole. Hence, the compound expression LINE - 7.3 is regarded as LINE - 7 + 0.3.

(f) A Compound Expression Constant enables a constant to be defined by a compound expression which occupies the full 24 bits of a word; when punched the Compound Expression Constant is preceded by an oblique stroke.

7.1.9 Special Brackets < >

If a program has a Master program, a Subprogram 1 and Subprogram 2 then the special brackets < > are used as an aid in program organization when data is common to all three divisions. The programmer is advised to assign the common codewords with values measured from the Master's datum point. Such codewords used in the N field of instruction are enclosed in special brackets < >. Then the codeword value in the N field is automatically adjusted, at the time it is read, to refer to the datum of the division in which the instruction occurs.

7.2 PROGRAM DIRECTIVES

The program tape can contain the following directives:-

#NEED

This directive must be followed by the list of requirements of the program including the following:

NAME=XXXX
CORE=n
PROG=0
SUB1=n_1
SUB2=n_2
READER=m_1
PUNCH=m_2
END

The first three items above and 'END' are required for the successful assembly of any program. The other items are optional. The quantities n_1 and n_2 represent the datum points of the subprograms 1 and 2 relative to the (zero) datum of the master program. XXXX is the four-character name by means of which the program will be controlled from the typewriter. The numbers m_1 and m_2 are the actual number of readers and punches required by the program. The reader used for input of the program is not assigned to the program, unless it is called for in the 'NEED' sequence.

#STOP = n

This directive will cause the assembler to stop reading tape and suspend the program, ready to enter it at location n on receipt of a GO message.

#TEXT

This directive implies that the line following is a message to be stored in the program. The line is read and stored in as many words as necessary, starting at a location one word ahead of the current transfer address T. A control word is formed which contains the number of characters and the starting address (T + 1) of the message, this control word is placed at location T. The transfer address is then adjusted to the next available word and the assembly continues to read more program.

#LIST

This directive outputs on the punch the codeword table.

7.2.1 Running a Program with the Assembler

The operator loads that version of EXECUTIVE which contains a copy of the Assembler.

The operator places the program tape in a tape reader and types:-

ASSEMBLE #JACK ON n

where n is the number of the reader. If the assembler is not present a message will be typed saying:-

NO

The program tape is read up to the directive #STOP. If the codeword list is called for it is punched with all values in decimal. If the name assigned to the program is CAMP, then the operator enters the program by typing:-

	GO0 #CAMP AT Y	where Y is a decimal address
or	GO0 #CAMP AT *Y	where Y is an octal address

If more program is to be read the operator types:-

CONTINUE #CAMP ON n.

7.3 EXAMPLES OF SOURCE PROGRAMS

The following trivial program will sum all the positive integers in a table of integers. The program is illustrating typing format only.

```
#NEED
CORE = 150
NAME = PSUM
PROG = 0
END
#PROG
ANS = 130
TABL = 35
LENG = 4
SUM = TABL + LENG

T = TABL
+1
-26
+32
+16
T = SUM
124 2  /LENG
100 5  /0
000 4  2/TABL
056 4  /T + 2
001 5  2/TABL
060 2  /T - 3
010 5  /ANS
156 0  /0
#STOP = SUM
```

The operator message sequence for the program would be:-

```
ASSEMBLE #PSUM ON 4.

GO 0 #PSUM
```

7.4 CORRECTIONS AND ALTERATIONS

Once the source tape has been prepared it is very likely that corrections or alterations to it will be required.

7.4.1 The Error Punch

In general the source tape is considered to be correctly punched if the printed version reads correctly; however, each type of program line has a defined format. If the assembler cannot interpret a program line it is punched out and the ASSEMBLER continues to read.

This read and punch action continues until the program directive #STOP is encountered.

7.4.2 Replacement of Stored Words

It would obviously be inconvenient to have to repunch the complete source tape and reassemble it every time a correction is required to the program, although it may be advisable to do this to prepare a final production version of the assembled tape. To correct mispunched stored words that have been detected or to alter individual stored words, it is only necessary for the new lines to be processed by Assembler. In the case of an error the information provided by the codeword table and error punch enables the programmer to ascertain which is the faulty program line.

An additional source tape can now be prepared which consists of a transfer address setting followed by a single program line, thus:

```
T = 57
000 1 4 /99
T = 176
046 0   /6
#STOP
```

This source tape can be assembled as normal program.

7.4.3 Correction of Mispunched Directives or Codeword Definitions

When an error is a directive or codeword definition the source tape must be repunched and reassembled.

7.4.4 Insertion or Deletion of Program

Deletion of a section of program can easily be effected by altering the first instruction to be omitted to be an unconditional branch to the instruction immediately following the omitted section.

Insertion of a section of program is more difficult and usually necessitates reassembly of all subsequent program. However, reassembly can be avoided on replacing the last instruction before the insertion, or the first one after the insertion, with an unconditional branch. This branch leads to a set of instructions, including the replaced one, which is added at the end of the program. These added instructions end with an unconditional branch back to the original program.

CHAPTER 8

THE DEBUG ROUTINE

TABLE OF CONTENTS

CHAPTER 8

THE DEBUG ROUTINE

8.1 Description

A program may be tested by being run in conjunction with the Debug, or Monitor, program. There are three basic modes of monitoring, and all three modes may be used at different stages of the same test run. The modes are:

(1) Single shot - ie. print out every instruction just before it is obeyed.

(2) Branch - ie. print out every branch instruction which causes branching to occur.

(3) Monitor Print - ie. print the contents of specified locations in specified styles.

Thus Single shot and Branch modes enable a program to be traced as it is obeyed, while the monitor print mode facilitates the printing of intermediate results.

The initial letters of these three modes - S, B and M - are used as "warning characters" to inform the Debug program of the modes of monitoring required at various points. A subsidiary set of warning characters is used to describe the styles of monitor printing required.

The Debug program is informed of the modes of monitoring required at various points by means of a "steering tape" (or deck). The steering tape consists of a number of statements, each one specifying a warning character, an address and, with the exception of M, a count and, optionally, a printing style for S and B only. Thus the statement

S 56 10

means trace in single shot mode from when the instruction in location 56 is obeyed, for at most 10 instructions. Since no style is specified it is taken to be zero. The format of a line is very flexible. Any number of spaces and Newline characters may precede the warning character, and any non-numeric characters (in ∝ shift for paper tape) may be typed between the warning character and the numeric address, between the numeric quantities, and at the end of the statement. Numbers may be specified in octal by immediately preceding them by *. Each line must end with a Newline character. Thus the four statements:

S AT 56,10

SINGLE SHOT AT 56 FOR 10 INSTRUCTIONS

S *70-10

S56/10

are each equivalent to the first example.

The full range of warning characters available is summarised in Table 1. If no style or style 0 is specified for S or B, the order number and order only will be printed at each step. In style 1 the contents of the accumulators, and the N-address where relevant, are also printed - as octal patterns.

Thus:

S82,15,1

will print each instruction obeyed after location 82, for at most 15 instructions, and will print the contents of the accumulators and the N-address (groups 00-07 inclusive, 13 and 17). Note that single shot monitoring occurs <u>before</u> the instruction is obeyed, so that the accumulators are printed as set before the instruction. The first instruction to be printed in single-shot is the one next to be obeyed after the one addressed in the S request.

8.1 continued

The maximum count permitted is 511, and of course care should be taken that all addresses specified lie within the program's range.

As an example of a steering tape, suppose it is required to run a program normally up to instruction 53, single shot for 25 instructions from that point, monitor a maximum of six successful branch instructions from location 100 in style 1, and whenever the instruction in location 82 is obeyed print the contents of accumulators 6 and 7 as fractions and locations 176-195 as integers. The program is to be entered at location 40. Then the steering tape would be punched as follows:

```
S 53,25

B 100,6,1

M 82

F 6,2

I 176,20

*

Z AT 40
```

"M 82" introduces the list of printing requirements when this instruction is obeyed. The symbol * terminates this list. A second (or more) M request at a different address could be included in order to monitor results at a later stage. The requests S, B and M above could appear in any sequence, provided that the lines M to * are regarded as a unit.

There are certain rules which must be obeyed.

(1) Two S, B and M request statements may not specify the same address.

(2) A list of requests subsidiary to M must terminate with *.

(3) Not more than 10 S, B and M requests in total may be made.

(4) Not more than 10 subsidiary printing requests in total may be made.

(5) The steering tape must end with a Z. The entry point is optional. If none is specified the entry point on the program tape will be used.

(6) A monitoring request may not be made at an instruction which has function 023, or at a "123 7" instruction. Such requests will cause the steering tape to be rejected. Since 123 7 instructions have meaning only when set by Debug, they should not be written into a normal program. Monitoring will cease and the program will be permanently suspended if a 123 7 instruction is encountered which has not been set by the Debug program.

(7) A monitoring request should not be made at any instruction which is overlaid or in some way altered during the execution of the program. Such requests either will be lost or will cause the debugging run to fail. Similarly a monitoring request should not be made at an instruction which is accessed as data by any other instruction.

WARNING CHARACTERS FOR STEERING TAPE

S	at N for Y instructions with Style 0 or 1 punching.	Single shot mode from address N, for Y instructions or until another mode setting is encountered. Type 0 - minimum punching (0 may be omitted) Type 1 - full punching
B	at N for Y entries with Style 0 or 1 punching.	Monitor on successful branch instructions, for Y such instructions, or until another mode setting is encountered.
M	at N	M defines the location N of the order at which monitor punching is required while the program is running. The nature of the punching is given by the subsequent series of P, I, F, O, D, L, V characters; without the M these cause punching at the actual instant they are read from the steering tape or deck.
P	at N for Y	Punch Program from N to (N + Y - 1)
I	at N for Y	Punch Integers from N to (N + Y - 1)
F	at N for Y	Punch Fractions from N to (N + Y - 1)
O	at N for Y	Punch Octal from N to (N + Y - 1)
D	at N for Y	Punch Y Double Length Integers from N onwards
L	at N for Y	Punch Y Double Length Fractions from N onwards
V	at N for Y	Punch Y Floating Point numbers from N onwards
*		Ends the list of requests for monitoring for a particular M point.
Z	at N	End of steering tape or deck. N is optional and may be used to specify the entry point to the program.

TABLE 1

8.2 Operation

The Debug program is read into the machine in the same way as any other program. It must be loaded before the program to be tested. The test program is then read in, and when it is loaded the operator types the MONITOR message, eg:

MONITOR 0#JACK ON UNIT 4.

where unit 4 is the unit from which Debug is to read its steering information, and the program name is preceded by the section (Master, subprogram 1 or 2) which is to be tested. At this time EXECUTIVE checks that the Debug program is in the computer, and extends its limit to allow it to access the test program's area. Next the steering tape (or deck) is read by the Debug program. Any test data which the program requires to read during testing should follow the steering information in the reader.

The Debug program reads the steering information one statement at a time, and changes the instructions in the locations specified to 123 7 instructions, maintaining within its own area a list of the instructions thus replaced.

Test programs should be designed to terminate by asking to be abolished. Provided that the test run does end in this way then this program may be abolished and a further test program read in, without the need to restore Debug. If the test run ends in any other way it is advisable to delete first the test program and then Debug, and then to reload Debug if it is required again.

Note: Occasionally a test run may terminate with a typewriter message:

ILLEGAL DEBUG XXXX

This indicates either that the program has spoiled its own register 8 (during a peripheral transfer or an 047 instruction), thus causing Debug to reach beyond the program's limit to access the "current instruction", or that a monitor point requested on the steering tape was beyond the program's limit.

8.3 Further Details

This section is included for interest, but is not a prerequisite to the successful use of the Debug program.

As has been stated, the 123 7 instruction has no meaning within (and should not be used by) normal programs. However, when the MONITOR message is typed, EXECUTIVE sets the Debug program into "Monitor Mode 2". It is a feature of the hardware that when a program is in monitor mode 2 a 123 7 instruction will cause an entry to EXECUTIVE - this action is called a Monitor Interrupt. Similarly, monitor mode 1 will cause a monitor interrupt to occur after every instruction, and monitor mode 3 will cause a monitor interrupt to occur after every successful branch instruction and at 123 7 instructions. Hence the three basic modes of monitoring - M, S and B.

Debug reads one statement at a time from the steering tape, and changes the instructions mentioned as monitor points into 123 7 instructions. Debug maintains a list of the instructions thus replaced. When the Z warning character is read, and if necessary the order number of the program adjusted, Debug returns control to EXECUTIVE by means of obeying a 123 7 instruction. EXECUTIVE now enters the test program in monitor mode 2, and the program will be obeyed as normal until the first 123 7 instruction is encountered, when a monitor interrupt to EXECUTIVE occurs. EXECUTIVE suspends the program and transfers control to the debug routine. The debug routine scans its list of break points to find the action required. First it carries out the replaced instruction, then considers the mode. If single shot is commencing, it does the required punching for the next instruction, alters the monitoring mode to 1 in a fixed location for EXECUTIVE, counts down one on the number of instructions to single shot and enters EXECUTIVE by means of a 123 7 instruction. EXECUTIVE suspends the debug routine, sets up monitor bits and restarts the program. In single shot mode, EXECUTIVE is entered after each instruction, control is transferred to the debug routine, punching is carried out and control returned to EXECUTIVE.

Unless a single shot sequence ends with, or encounters before the end, another mode-setting instruction, mode 2 will be set and no more debug punching will occur until the next point at which single shot or branch mode or monitor printing has been requested.