CS87

$3.00

CS139 LECTURE NOTES
PART I
SECTIONS 1 THRU 21

PRELIMINARY VERSION

BY

J. EHRMAN

1. INTRODUCTION

These notes are meant to provide an introduction to System/360 which will help the reader to understand and to make effective use of the capabilities of both the machinery and some of its associated service programs. They are largely self-contained, and in general the reader should need to make only occasional reference to the "System/360 Principles of Operation" manual (IBM File No. S360-01, Form A22-6821), and to the "Operating System/360 Assembler Language" manual (IBM File No. S360-21, Fom C28-6514).

A digital computer can be considered from a variety of viewpoints; for convenience we will mention five possible ones, each of which treats the inner workings of the computer in successively less detail. To an engineer concerned with the design of its logical circuits, a computer might be considered basically a collection of devices for controlling and ordering the flow of electrical impulses. At another level a person concerned with methods to be used to make these logical circuits perform certain operation8 such as division might treat a computer as a collection of registers, switches, and control mechanisms which, when provided with the appropriate data, are to perform a series of steps leading eventually to the computation of a quotient. At the next level one might consider the basic operations of a computer to be those operations which perform a single arithmetic operation, a simple data movement, or a test of a single piece of data. Another viewpoint (typical of "higher-level languages" such as FORTRAN, ALGOL, and PL/1) is to consider that the basic operations of interest are the movement of blocks of data, the evaluation and assignment of mathematical expressions, and the control of counting and testing operations. At yet another level, as in certain applications such as traffic or production simulation, data reduction, and network analysis, the computer is considered as a device which accepts information in a form which closely approximates that of the problem under consideration, and produces output directly applicable to that problem.

Each of these ways of viewing a computer is of course not especially distinct from its neighbors. In this treatment we will be primarily concerned with the middle level, namely that of considering the basic operations, or instructions, that we want the computer to perform to be single arithmetic or logical operation*, simple data transmission operations,. etc. We will from time to time have occasion to consider the canputer from "neighboring" viewpoints: in some circumstances it will be useful to know some details of the internal sequencing of operations such as multiplication and division; at other times it will be convenient to consider instructions to the machine which will perform operations in a larger context than that ordinarily considered.

This level of programming which will be our primary concern is usually known as "machine language" programming; however, since the process of actually getting the desired instructions into the computer requires the aid of a number of other programs, the first of which is called an assembler, the terms "assembler language" programming or "assembler coding" are also used. Thus the service program of most concern will be the Operating System/360 Assembler; other programs of interest will be the Linkage Editor and the Resident Supervisor, each of which will be considered in the appropriate context.

2. BINARY AND HEXADECIMAL NUMBERS

System/360, like most other digital computers, makes heavy use of binary numbers for internal arithmetic. Because digits in a base two representation can take on only the values 0 and 1, it is relatively simple to build a mechanical or electrical device which represents the digit. For example, a 1 digit may be represented by the presence or absence of a current through a given circuit component or by the presence of a positive or negative voltage at some point. Because facility with the use of binary numbers is fundamental to an understanding of-the basic operation of System/360, it is useful to summarize the properties of the binary number representation. For the time being, all numbers will be assumed to be integers.

In base ten, when we write a number such as 1735 we mean the quantity

$$1 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0.$$

That is, each digit position as we move to the left is weighted by another power of the base, ten. Similarly, when in binary arithmetic we write the number 11010 we mean

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0,$$

which of course is not the same as what is meant by the <u>decimal</u> number 11010, where powers of ten are understood. In fact, the <u>binary</u> number 11010 is the representation (in the number system with base two) of the decimal number 26, which is obtained simply by performing the sum in the above example.

To clarify which base is intended when we write numbers, it will be convenient to attach a "subscript" at the right end of the number to indicate the base being used:

$$26_{10} = 11010_2, \qquad 1_{10} = 1_2,$$
$$10_{10} = 1010_2, \qquad 1000_2 = 8_{10}.$$

As the decimal numbers being represented become larger, the number of binary digits required becomes larger also.

Thus,

$$999_{10} = 1111100111_2.$$

It is therefore convenient to find a more compact notation for binary numbers. If we consider groups of four binary digits at a time, the possible decimal values that can be represented run from zero to fifteen, If we then choose to represent each of these groups by the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, we can establish the following table of correspondences:

Binary Digits	Decimal Value	Base 16 Digit
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

TABLE I.

Hexadecimal, Decimal, and Binary Digits

We will call the base sixteen digits in the third column <u>hexadecimal</u> digits, and will generally use them in situations when we have occasion to refer to binary numbers. As before, a "subscript'" of 16 will be used to indicate that the given set of digits is to be understood to have base 16:

$26_{10} = 11010_2 = 1A_{16}$, $26_{16} = 100110_2 = 38_{10}$, $1_{10} = 1_2 = 1_{16}$,

$10_{10} = 1010_2 = A_{16}$, $100_2 = 8_{10} = 8_{16}$, $100_{10} = 64_{16} = 1100100_2$.

Converting numbers between binary and hexadecimal representations can be seen to be quite simple: to convert a hexadecimal number to binary, simply substitute for each hexadecimal digit the four binary digits it represents; to convert a binary number to hexadecimal, group the binary digits four at a time starting from the right, and substitute the corresponding hexadecimal digit. For example:

$D5B_{16} = 1101\ 0101\ 1011_2$, (hexadecimal to binary)

$11\ 1110\ 100_2 = 3E8_{16}$. (binary to hexadecimal)

In the second of these examples it was assumed that two extra binary zero digits could be added at the left end of the number without affecting its value; thus we can write

$11_{16} = 10001_2$ rather than $0001\ 0001_2$.

Conversion between decimal and hexadecimal representations is somewhat more cumbersome, but if a conversion table such as the one in the Appendix is not available, the following method is usually sufficient for hand calculation.

In the positional notation we are accustomed to using, a string of digits $d_n d_{n\ 1} \ldots d_2 d_1 d_0$ is the representation in some base D of the number X:

$$X = \sum_{k=0}^{n} d_k D^k = d_0 D^0 + d_1 D^1 + d_2 D^2 + \ldots + d_n D^n .$$

Suppose we want to convert from this representation in bare D to the representation in a new base B:

$$X = \sum_{k=0}^{m} b_k B^k = b_o B^0 + b_1 B^1 + b_2 B^2 + \ldots + b_m B^m .$$

The known quantities are the old and new bases D and B, and the digits d_k of the old representation; then to find the digits bk in the new representation, the following scheme is used.

Divide X by B; save the quotient, end the remainder is b_o. That this is so can be seen from the definition of the quotient and remainder:

$$X = \text{Remainder} + B \times \text{Quotient} = b_o + B \times [b_1 + b_2 B + b_3 B^2 + \ldots + b_m B^{m-1}].$$

Divide the saved quotient by B; save the new quotient, and the new remainder is b_1. Continue this process until a zero quotient is obtained, and the successive remainders are the digits $b_o, b_1, \ldots b_m$; note that they were obtained in order of <u>increasing</u> significance.

<u>Examples</u>

1. Convert 19_{10} to base 2.

$$\begin{array}{r} 9 \\ 2\overline{)19} \\ \underline{18} \\ b_o = 1 \end{array} \qquad \begin{array}{r} 4 \\ 2\overline{)9} \\ \underline{8} \\ b_1 = 1 \end{array} \qquad \begin{array}{r} 2 \\ 2\overline{)4} \\ \underline{4} \\ b_2 = 0 \end{array} \qquad \begin{array}{r} 1 \\ 2\overline{)2} \\ \underline{2} \\ b_3 = 0 \end{array} \qquad \begin{array}{r} 0 \\ 2\overline{)1} \\ \underline{0} \\ b_4 = 1 \end{array}$$

Hence, $19_{10} = 10011_2$.

2. Convert 1000_{10} to base 16. (Note that the conversion arithmetic is done in base 10.)

$$\begin{array}{r} 62 \\ 16\overline{)1000} \\ \underline{992} \\ b_o = 8 \end{array} \qquad \begin{array}{r} 3 \\ 16\overline{)62} \\ \underline{48} \\ b_1 = 14 \text{ or } E_{16} \end{array} \qquad \begin{array}{r} 0 \\ 16\overline{)3} \\ \underline{0} \\ b_2 = 3 \end{array}$$

Hence $1000_{10} = 3E8_{16}$.

3. Convert 627_{10} to base 9.

$$\begin{array}{r} 69 \\ 9\overline{)627} \\ 621 \\ \hline b_0 = 6 \end{array} \qquad \begin{array}{r} 7 \\ 9\overline{)69} \\ 63 \\ \hline b_1 = 6 \end{array} \qquad \begin{array}{r} 0 \\ 9\overline{)7} \\ 0 \\ \hline b_2 = 7 \end{array}$$

So that $627_{10} = 766_9$.

4. Convert 766_9 to base 7. (This is simple once you've memorized the multiplication table in base 9, which is the base used for the conversion arithmetic.)

$$\begin{array}{r} 108 \\ 7\overline{)766} \\ 762 \\ \hline b_0 = 4 \end{array} \qquad \begin{array}{r} 13 \\ 7\overline{)108} \\ 103 \\ \hline b_1 = 5 \end{array} \qquad \begin{array}{r} 1 \\ 7\overline{)13} \\ \\ b_2\ \underline{7} = 5 \end{array} \qquad \begin{array}{r} 0 \\ 7\overline{)1} \\ 0 \\ \hline b_3 = 1 \end{array}$$

Thus $766_9 = 1554$,.

This can be done in more roundabout (but comprehensible) fashion by converting to base ten first and then doing the arithmetic in decimal,:

$$766_9 = 7 \times 81 + 6 \times y + 6 = 567 + 54 + 6 = 627_{10}$$

$$\begin{array}{r} 89 \\ 7\overline{)627} \\ 623 \\ \hline b_0 = 4 \end{array} \qquad \begin{array}{r} 12 \\ 7\overline{)89} \\ 84 \\ \hline b_1 = 5 \end{array} \qquad \begin{array}{r} 1 \\ 7\overline{)12} \\ \\ b_2\ \underline{7}5 = \end{array} \qquad \begin{array}{r} 0 \\ 7\overline{)1} \\ 0 \\ b_3 = \underline{\ \ }\ 1 \end{array}$$

So that $766_9 = 1554_7$ again.

5. Convert 1413_5 to base 10. This is most simply done by expanding the positional notation:

$$1413_5 = 1 \times 125 + 4 \times 25 + 1 \times 5 + 3 = 233_{10}.$$

Alternatively, using the fact that $10_{10} = 20_5$ in base 5 arithmetic,

$$\begin{array}{r} 43 \\ 20\overline{)1413} \\ 130 \\ \hline 113 \\ 110 \\ \hline b_0 = 3 \end{array} \qquad \begin{array}{r} 2 \\ 20\overline{)43} \\ 40 \\ \hline b_1 = 3 \end{array} \qquad \begin{array}{r} 0 \\ 20\overline{)2} \\ 0 \\ \hline b_2 = 2 \end{array}$$

giving $1413_5 = 233_{10}$.

6. Convert $3E8_{16}$ to base 10. In this case it is usually simplest to use the positional notation used earlier:

$$3E8_{16} = 3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0,$$

and then this sum can be evaluated in decimal. Thus we find

$$3E8_{16} = 3 \times 256 + 14 \times 16 + 8 = 768 + 224 + 8 = 1000_{10}.$$

This type of conversion is considerably simpllfied by the use of the table of multiples of powers of 16 in Table II or (for small numbers) by the use of the conversion table.

Discussion of binary arithmetic -- addition, subtraction, multiplication, and division -- will be deferred until later.

We will use several abbreviations regularly: a bit will mean a binary digit, and we will use hex as short for hexadecimal.

Hex Digit	×1	$\times 16^1$			$\times 16^2$ $\times 16^3$	$\times 16^4$	$\times 16^5$ $\times 16^6$	$\times 16^7$
1	1	16	256	4,096	65,536	1,048,576	16,777,216	268,435,456
2	2	32	512	8,192	131,072	2,097,152	33,554,432	536,870,912
3	3	48	768	12,288	196,608	3,145,728	50,331,648	805,306,368
4	4	64	1024	16,384	262,144	4,194,304	67,108,864	1,073,741,824
5	5	80	1280	20,480	327,680	5,242,880	83,886,080	1,342,177,280
6	6	96	1536	24,576	393,216	6,291,456	100,663,296	1,610,612,736
7	7	112	1792	28,672	458,752	7,340,032	117,440,512	1,879,048,192
8	8	128	2048	32,768	524,288	8,388,608	134,217,728	2,147,483,648
9	9	144	2304	36,864	589,824	9,437,184	150,994,944	2,415,919,104
A	10	160	2560	40,960	655,360	10,485,760	167,772,160	2,684,354,560
B	11	176	2816	45,056	720,896	11,534,336	184,549,376	2,952,790,016
C	12	192	3072	49,152	786,432	12,582,912	201,326,592	3,221,225,472
D	13	208	3328	53,248	851,968	13,631,488	218,103,808	3,489,660,928
E	14	224	3584	57,344	917,504	14,680,064	234,881,024	3,758,096,384
F	15	240	3840	61,440	983,040	15,728,640	251,658,240	4,026,531,840

TABLE II.

Multiples of Powers of 16

3. STRUCTURE OF SYSTEM/360

It is usual to describe the structure of most digital computers in terms of four major components: memory, arithmetic, control, and input-output units. It should be understood that an actual machine may not have components which can be separately identified in this way, but that for conceptual purposes it is possible to think of them as distinct units.

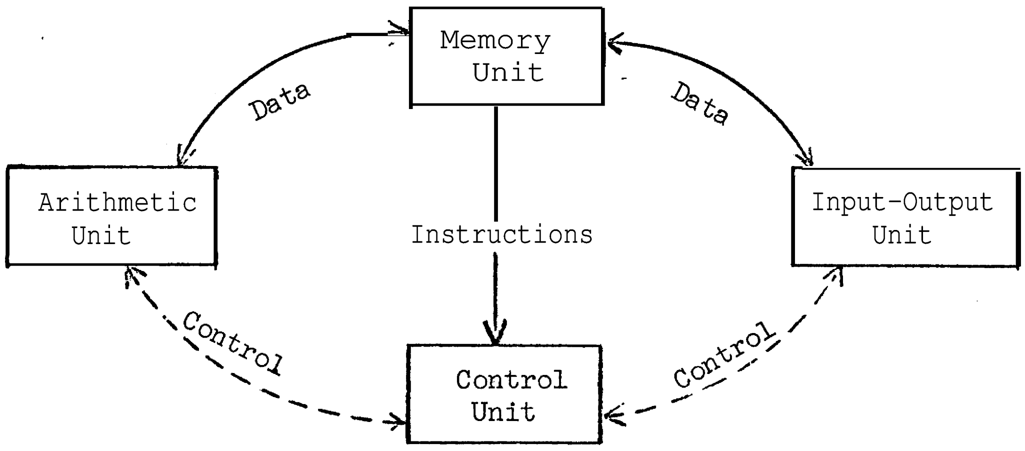


Figure 3.1 Structure of a Typical Computer

The solid arrows in the figure represent schematically the possible paths of data flow among the various units, and the dashed arrows indicate the flow of control signals. As indicated, the instructions for the control unit are contained in the same memory as the data used by the arithmetic and input-output units; this property is what gives modern digital computers their flexibility and power -- the computer can, on the basis of certain computed results, modify the instruction sequences which control the way it will treat other data.

In-the System/360 computers many of the functions performed by the control and arithmetic units use the same internal components, so that it is easier to make no special distinction between the two and simply call the combination the Central Processing Unit, or CPU.

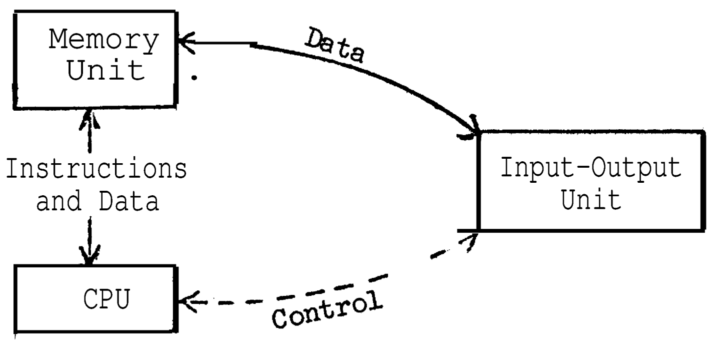


Figure 3.2 Structure of System/360

These units will be described in varying detail: the memory and arithmetic unit are of major concern to the machine language programmer; certain features of the control unit will be examined closely while others will be ignored for the time being; the input-output unit, which is simply a term which collectively denotes devices such as card readers, printers, magnetic tape units, etc., will be described only as necessary to make use of the computer in certain elementary ways.

The terminology introduced here is by no means fixed in the literature and everyday usage of the computing profession. For example, it is common to refer to magnetic drums as memory devices even though they are accessed through what we have called the Input-Output Unit. What we will call "memory" can be more accurately described by calling it the High-Speed Random Access Magnetic Core Memory, but the economy of a single term is apparent.

Memory

The basic unit of data in System/360 is a group of eight bits called a byte. The bits in a byte are by custom numbered from 0 to 7, beginning on the left with the numerically most significant digit. The definition of the "left" side of a byte will become clear shortly.

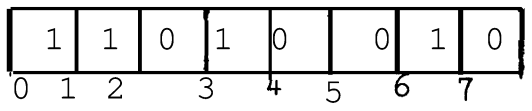


Figure 3.3 A byte containing the 8 binary digits 11010010

The memory unit is arranged so that it will hold a certain number of bytes in such a way that each byte may be accessed as rapidly as any other. The bytes may be considered to be individually numbered in order, beginning at zero; the number associated with each byte is its <u>address</u> or <u>location</u> in the memory unit. The memory may be thought of as a linear string of bytes arranged in order of increasing addresses.

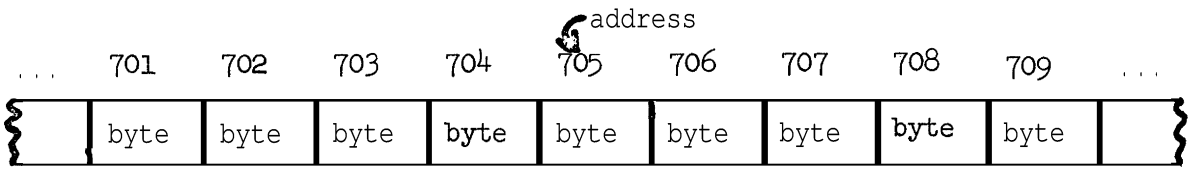


Figure 3.4 A portion of memory

Many of the machine instructions which refer to bytes 'in memory" (which is an abbreviation for "in the memory unit") actually refer to a group of consecutive bytes. In such a situation the group, or "operand", is always addressed by referring to its <u>leftmost</u> member, namely the byte with the lowest address in the group. Furthermore, certain instructions require that the address of a group of bytes (which, as stated, is the address of the leftmost byte) also be a multiple of the length of the group: the possible values for these instructions are 2, 4, or 8, and in such cases it is usual to refer to the groups of bytes whose addresses and lengths satisfy this condition as half'word, fullword, and doubleword data, respectively.

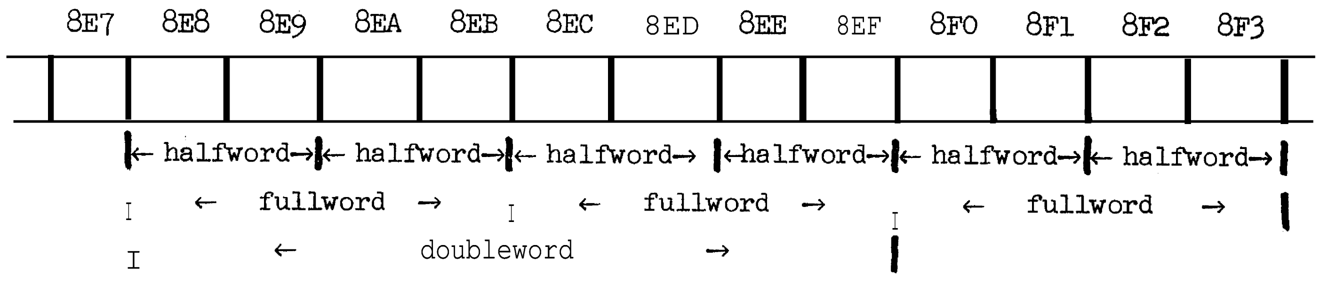


Figure 3.5 A portion of memory

Note that if (for example) a halfword operand (that is, a group of two bytes whose address is divisible by 2) were specified for some operation, and the address of that 16-bit operand were $8EA_{16}$, then bit 0 of the byte at $8EB_{16}$ would be considered to follow immediately after bit 7 of the byte at $8EA_{16}$. It is in this sense that bit 0 is taken to be the 'leftmost' bit of a byte: it follows (for certain operations) immediately after bit 7 of the byte at the next lower memory address.

The data contained in bytes or groups of bytes in memory can be manipulated in many different ways, depending on the intentions of the programmer. These will be discussed later.

Central Processing Unit

There are three things in the CPU of interest to the programmer: the general purpose registers, the floating-point registers, and the Program Status Word. There are sixteen general purpose (or simply general) registers, numbered from zero to fifteen, each one of them being 32 bits (or 8 hex digits or 4 bytes or 1 fullword) in length. They are represented schematically in the figure below.

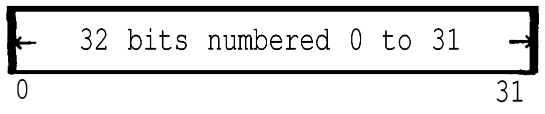


Figure 3.6 A Single General Purpose Register

RO	R1
R2	R 3
R4	R5
R6	R7
R8	R9
R10	R11
R12	R13
R14	R15

Figure 3.7 General Purpose Registers

Figure 3.7 is arranged with the registers in pairs, the left being an even-numbered register and the right being the next higher odd-numbered register. This is because certain of the machine operations (such as shifting, multiplication, and division) require the use of a pair of registers, and in such cases it is always such an even-odd numbered pair. We will have many occasions to refer to the general registers, so that it is convenient to introduce a short notation: we will write Rn to refer to general register n, so that R0 means register 0, R14 means register 14, and so on.

The presence of floating-point registers in the CPU is an option for certain models, but we will assume that the user of the machine we are discussing writes his programs for a computer that includes the floating-point feature. There are four floating-point registers, each 64 bits (or 16 hex digits or 8 bytes or 1 doubleword) in length. They are numbered 0, 2, 4, and 6.

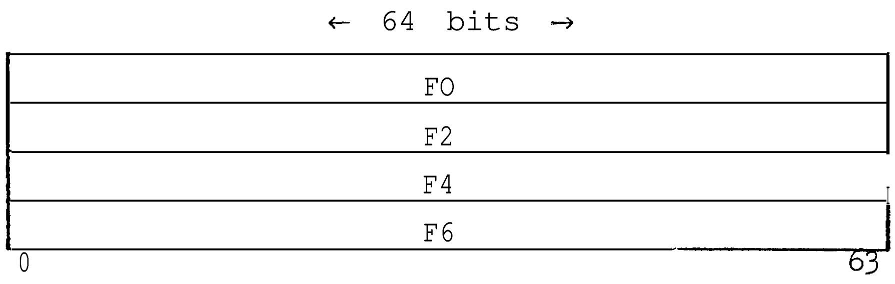


Figure 3.8 Floating-Point Registers

In certain circumstances the floating-point registers are used to contain operands 32 bits long, in which-case they use only the left half of the register, and the rightmost 32 bits of the registers are ignored; this will be discussed in the chapter on floating-point arithmetic. As in the figure above we will use the abbreviations F0, F2, F4, and F6 to refer to the four floating-point registers.

In many cases it will be easier to use the term "register" for either a general purpose register or a floating-point register; which is meant will be clear from the context of the discussion.

The Program Status Word (or PSW for short) is not of direct concern in most programming applications, so that we need not be concerned at present with examining it in detail. The PSW is a double-word (and hence it is actually a Program Status Doubleword, but nobody really cares about the difference) which indicates in a compact form certain important details of the operation of a program in the System/360 CPU.

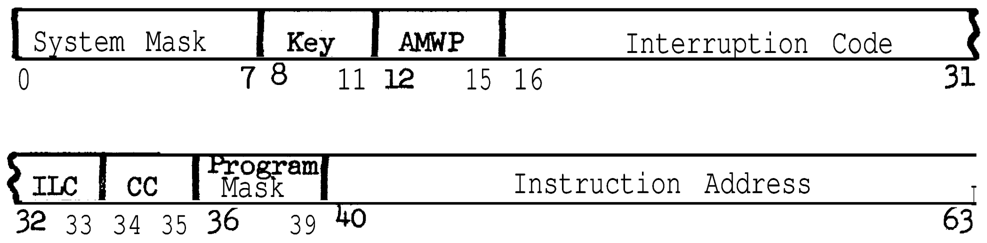


Figure 3.9 Program Status Word

The various pieces of the PSW (which resides in the CPU, not in memory, and is therefore pretty much inaccessible) will be explained in various contexts later. For the present, however, the items of interest lie in the rightmost 32 bits: the portions denoted "ILC" (Instruction Length Code), "CC" (Condition Code), and "Instruction Address" (which we will abbreviate "IA") are the parts of the PSW which will be treated in most detail. The Condition Code indicates the result of certain operations (e.g., that a sum is negative) and the two bits of the CC can be tested by certain instructions. This right-hand portion of the PSW will be of more interest than the first 32 bits for most of the following discussion; the ILC and IA will be discussed in the next section. The reader is cautioned that there will be omissions in the discussion of the PSW until the treatment of interruptions, where the subject will be covered in greater detail.

Input-Output

The process of data transmission between the memory and external devices such as card readers, printers, card punches, magnetic tapes, magnetic drums, disc files, etc., is handled in System/360 by channels. These are capable of transmitting bytes of data in such a way that the CPU can continue with the execution of a processing program at the same time that the channel is moving information to or from a different area of memory. The problems involved in synchronizing the transmission of such data with its use by the processing program in the CPU are quite complex and will be avoided for the time being, but will be touched upon later during the discussion of interruptions.

4. INSTRUCTIONS (I)

As was indicated in the diagrams of the "structure" of a computer in the previous section (Figs. 3.1 and 3.2), the instructions obeyed by the computer are held in memory along with the data to be processed. Instructions in System/360 can be 2, 4, or 6 bytes long, depending on what the placement of the data to be operated on happens to be, and on what the instruction causes to be done with the data. Instructions are always aligned so that the leftmost byte is on a halfword boundary:- that is, an instruction address must always be divisible by two. Otherwise, it doesn't matter, for instance, that a 4-byte instruction begins halfway between two fullword boundaries.

The actual process of performing the instructions in a program may be visualized as in the following figure.

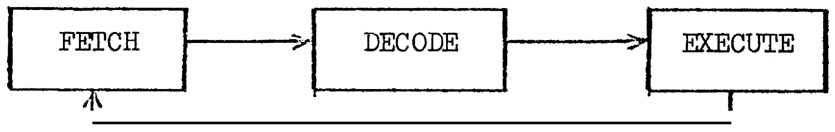


Figure 4.1 Instruction Cycle

In the "Fetch" portion of the cycle, the CPU causes the instruction in memory which begins at the byte whose address is contained in the rightmost 24 bits of-the PSW (the Instruction Address or IA) to be brought into the CPU and placed in an internal holding register where it may be examined. Though this internal register is not accessible to the programmer, we will from time to time make reference to it, so we will simply call it the Instruction Register, or IR for short. There is a simple way for the CPU circuits to know the length of an instruction and therefore how many bytes to bring from memory; this will be explained at the end of this section.

To complete the Fetch portion of the cycle, the CPU adds the length in bytes of the instruction now in the instruction register to the IA in the PSW, so that it will contain the address of the next instruction to be fetched when the current instruction has completed its execution. This means of course that instructions are packed tightly in memory; there are no leftover bytes between instructions.

To decode the instruction, the CPU examines the bit pattern of the bytes in the IR to see what action is intended. Since (1) the bytes were brought from memory and (2) the memory contains both data and instructions, it is quite possible that the bytes brought to the IR were intended by the programmer to represent data and not instructions. The CPU, however, has no way of knowing this in advance; it simply goes to the memory address given in the IA portion of the PSW and puts those bytes into the IR to be interpreted as an instruction. If this is what was intended, well and good (remember that in the beginning of Section 3 it was noted that the ability to treat instructions as data is what gives a computer its power); otherwise strange things can occur. Because not all of-the possible bit patterns in the IR represent "legal" instructions (i.e., actions the CPU can actually perform), the decoding mechanism can occasionally detect a confused situation before too much damage has been done, and cause the appropriate remedial actions to be Initiated.

Assuming that the bytes in the IR do indeed contain a valid instruction, some further actions may be necessary before the decoding is completed, such as the calculation of addresses of data to be operated on during the "Execute" portion of the cycle.

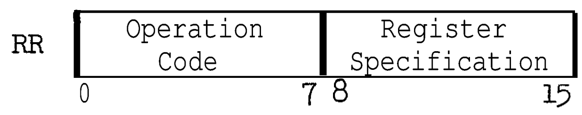
It is during this final execution phase that the actual operation is performed. The operation may be a simple one which could, for example, cause the contents of one general register to replace the contents of another, or it may involve many intermediate steps of complicated logic or arithmetic. If no errors are detected during the execution phase (such as attempting to divide something by zero), the CPU then begins the cycle again by returning to the "fetch" portion of the cycle. It should be noted that the time required for all this is very small even for a relatively slow computer: the entire cycle takes only millionths of a second, so that with this tremendous rapidity it is possible to perform calculations far too laborious to be done by hand.

The instructions which can be executed by the System/360 CPU can be grouped into five general classes:

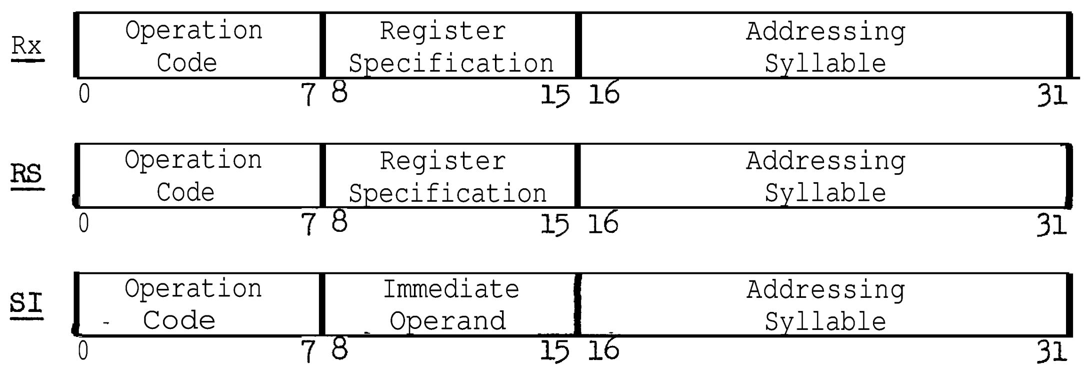
1) Register-to-Register (RR),
2) Register to Indexed Storage (RX),
3) Register-to-Storage (RS),
4) Storage-Immediate (SI),
5) Storage-to-Storage (SS).

The letters RR, RX, RS, SI, and SS are abbreviations which will be used regularly to indicate the class of instructions being discussed; the specific instructions belonging to each class will be treated in later chapters.

RR instructions are always two bytes long.



RX, RS, and SI instructions are always four bytes long.



The RX and RS instruction formats differ only in the interpretation given by the CPU to the bits in the "Register Specification" byte.

SS instructions are always six bytes long.

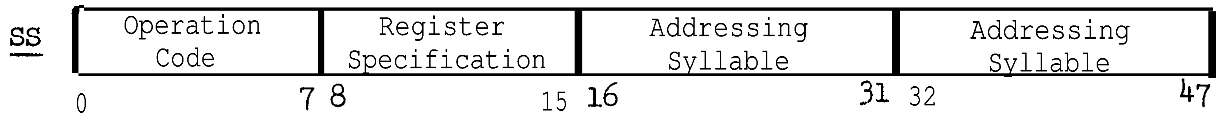


Figure 4.2 Instruction Formats

It can be seen that the operation code, which specifies what action is to be performed, occupies the first byte of the instruction. The second byte contains information necessary to the details of the execution of the instruction; its interpretation differs for instructions in the various classes. For all instructions except RR instructions an addressing syllable is used by the CPU to compute the address of an operand in memory; this process will be discussed in the next section.

The first two bits of the operation code contain the information which tells the CPU how many bytes are needed from memory to obtain the complete instruction. Since a minimum of two bytes per instruction must always be fetched, the CPU can check these two leading bits to tell how many more bytes are required. The bit patterns are as shown in the figure below; the xxxxxx is meant to indicate the remaining six bits of the eight-bit operation code.

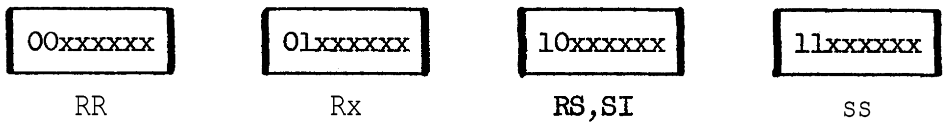


Figure 4.3 Bit Patterns for Each Instruction Type

Thus if the first two bits are 00 the instruction is two bytes long; if the bits, are 01 or 10 the instruction is four bytes long; and if the bits are 11 the instruction is six bytes long. Before proceeding with the decoding phase of the instruction cycle, the CPU places the number of <u>pairs</u> of bytes in the instruction in bits 32 and 33 of the PSW (namely in the position labeled "Instruction Length Code"). If an error is detected during the decoding or execution of the instruction, and if the PSW at the time of the error is saved somewhere, then the programmer can determine (by examining the IA and ILC) what instruction caused the error. (This is of course precisely what <u>is</u> done; we will note for now that if the ILC were not saved, it would not be possible to determine the exact location of the offending instruction, since the location of the <u>next</u> instruction to be executed is what appears in the PSW and the length of the bad instruction is variable. This is a subject with many ramifications, to be covered later.)

5. ADDRESSING

To refer to items in memory such as data or instructions, the programmer must usually make use of one of the general purpose registers. This is due to the way the CPU uses the information in an "addressing syllable", which always occupies a halfword in memory.

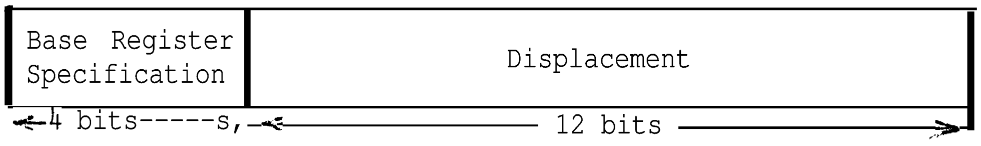


Figure 5.1 Structure of an Addressing Syllable

The 4-bit field at the left of the addressing syllable contains a single hex digit which can take values from 0 to 15, and which specifies a general purpose register. The 12-bit field in the rest of the addressing syllable contains a number called the *displacement* which can take values from 0 to 4095. To generate the address or' an operand, the CPU does the following:

Step 1) The 12-bit displacement is put at the right-hand end of a 24-bit internal register called the Memory Address Register (abbreviated MAR), and the leftmost 12 bits of the MAR are cleared to zeros;

Step 2a) If the base register specification digit is *not* zero, then the rightmost 24 bits of the general purpose register specified are added to the contents of the Memory Address Register, and carries out the left end of the MAR are ignored (the register used is called the *base* register);

Step 2b) If the base register specification digit *is* zero, nothing is added to the MAR (so that R0 cannot be used as a base register).

At this point the quantity in the MAR may be used as the address of an operand in memory. However, if the instruction is of type RX, a further step called an indexing cycle is needed. The second byte of an RX-type instruction (labeled "Register Specification" in Fig. 4.2) contains two 4-bit fields, the second of which is called the index register specification:

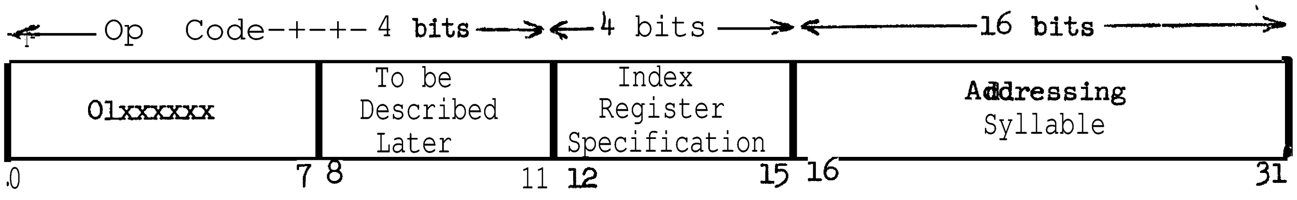


Figure 5.2 RX Instruction Showing Index Register Specification

Step 3) If the instruction is of type RX, and the 4-bit index register specification digit is not zero, then the right-most 24 bits of the general purpose register specified by the index register specification digit are added (again ignoring carries out the left end) to the contents of the MAR.

The resulting quantity in. the MAR is called the effective address.

(Binary arithmetic will be discussed in detail in Section 7. For the following examples, it should be sufficient to note that 0 + 0 = 0; 0 + 1 = 1 + 0 = 1; 1 + 1 = 0 and carry 1. These examples go into considerably more detail than is necessary for a working understanding of addressing, and the arithmetic is included just for the sake of completeness. Since addressing will reappear in several later places, don't worry about absorbing all the fine points immediately.)

Examples

1. Suppose the addressing syllable of an SI-type instruction is $\boxed{1011}\boxed{001011010101}$ in binary (or $\boxed{B}\boxed{2D5}$ in hex) and suppose that the contents of general purpose register 11_{10} is
1100 0111 0011 1110 1001 0000 1010 1111 in binary (or C73E90AF in hex).
Then the effective address of the instruction is (giving both binary and hex):

$$\begin{array}{rrl} & 0000\ 0000\ 0000\ 0010\ 1101\ 0101 & \quad 0002D5 \quad \text{displacement} \\ + & \underline{0011\ 1110\ 1001\ 0000\ 1010\ 1111} & + \underline{3E90AF} \quad \text{base (from R11)} \\ & 0011\ 1110\ 1001\ 0011\ 1000\ 0100_2 & \quad 3E9384_{16} \end{array}$$

2. Suppose the addressing syllable of the same instruction is $\boxed{0}\boxed{468}$. Then the effective address is 000468_{16}, since R0 cannot be used for a base.

3. Suppose an RX-type instruction is $\boxed{43}\boxed{0A}\boxed{7}\boxed{468}$, and that the contents of R7 is 12345678_{16} and the contents of R10 is $FEDCBA98_{16}$. (Note that the base register specification digit, namely 7_{16}, means that R7 will be used. The instruction chosen for this and the next two examples would, if execu'ced by the CPU, cause the contents of the byte at the memory location given by the effective address to replace the rightmost byte of R0.) Then the effective address is

```
  0000 0000 0000 0100 0110 1000          000468  displacement
+ 0011 0100 0101 0110 0111 1000        + 345678  base (from R7)
  0011 0100 0101 1010 1110 0000          345AE0
  1101 1100 1011 1010 1001 1000        + DCBA98  index (from R10)
+ 0001 0001 0001 0101 0111 0111₂         11157578₁₆ effective address
```

(The carry out the left end is ignored.)

4. Suppose an RX-type instruction is $\boxed{43}\boxed{00}\boxed{7}\boxed{468}$ and t&t the contents of register 7 is as in example 3. Th 3 the effective address is

```
  0000 0000 0000 0100 0110 1000          000468  displacement
+ 0011 0100 0101 0110 0111 1000        + 345678  base
  0011 0100 0101 1010 1110 00002         345AE0₁₆ effective address
```

5. Suppose an RX-type instruction is $\boxed{43}\boxed{07}\boxed{0}\boxed{468}$ and that the contents of register 7 is as in example 3. Then the effective address is

```
  0000 0000 0000 0100 0110 1000          000468  displacement
+ 0000 0000 0000 0000 0000 0000        i- 000000  base
  0000 0000 0000 0100 0110 1000          000468
+ 0011 0100 0101 0110 0111 1000        + 345678  index
 -0011 0100 0101 1010 1110 00002         345AE0₁₆ effective address
```

In this example the values of the base and index register specification digits were interchanged from those in example 4, so that the indexing cycle was required in example 5 to compute the same effective address. Or-the smaller models (30, 40, and 50) of the System/360 series, extra time is required to perform this additional arithmetic, so that in some cases it may be worth trying to avoid unnecessary indexing cycles.

In a situation where only one register is used in the calculation of the effective address (as above, where the base register specification digit was 0 and the index register specification digit was 7) it is customary to speak of that register as the base register, even though it may be the index register in an RX-type instruction. This allows us to refer to this addressing scheme as a base-displacement addressing technique.

The effective address in the MAR can have a number of uses, the primary one being to address operands in memory; it is also used for shifting and branching (which will be discussed later). However, three further observations may be made about effective addresses which will be used to refer to data in memory.

First, the presence of 24 bits in the MAR means that a System/360 computer has the capability of addressing 2^{24} or 16,777,216 bytes. Now it will almost always be the case that the model being used will have a smaller memory, since memory is one of the more expensive parts of the computer. Thus, suppose (for example) we are programming for a machine with $2^{16} = 10000_{16}$ = 6553610 bytes of memory, and use an instruction which generates an effective memory address-which is larger than 10000_{16}. Since this effective address cannot refer to anything accessible to the CPU, some sort of error-recovery procedure must be initiated; this error condition is known as an addressing exception, and causes a program interruption to begin the error-handling sequence.

Second, it was noted in the earlier discussion of the memory that certain instructions which operate on groups of bytes such as fullwords require that the address of the leftmost byte be divisible by the length (in bytes) of the operand. If this condition is not satisfied, another error condition known as a specification exception is recognized. For example, the RX-type instruction $\boxed{58}\boxed{40}\boxed{0}\boxed{123}$ specifies that a fullword operand is to be transmitted from memory and placed in R4. Since the effective address for this case is 000123_{16}, the proper (i.e., leftmost) byte of the fullword is not being addressed, so that a specification exception is recognized during the execute portion of the instruction cycle, and a program interruption will initiate the error-recovery sequence.

Third, because the only part of the memory which can be referred to without the use of a base register is the area with addresses 0 to $4095_{10}=FFF_{16}$, the programmer will almost invariably be required to refer to operands in memory with the help of a base register. (One might think that he need only fit his program into those first 4096 bytes and then not have to worry about all this base-register trouble, but that area of memory and more will usually be occupied by the routines which provide error handling, input-output operations, and the like; it's called "The System". So we just have to live with it.) This means that if we are to address a byte in memory at address Q, there must be a base register available (that is, one of registers 1 to 15) which contains a number between Q and Q-4095, since we could then generate an effective address of Q by using a displacement between 0 and 4095. If there is no such number in a register, then the byte at Q is not addressable. Thus, if all the general registers contain zero, only the first 4096 bytes of memory are addressable! Usually what must be done is to place some constant in a register which then allows us to address the desired region of memory; that is, that register then provides addressability for that region. However, if the constant itself is in another portion of memory which is not currently addressable, we are back to where we started, needing another constant to address the first constant. In fact, it is possible for the CPU to be executing instructions in a portion of memory, and the instructions cannot address themselves! (Remember that the IA is in the PSW, not in a register.) Fortunately, there are simple solutions to the problems of addressing, and these will be the subject of several later discussions.

6. TWO'S COMPLEMENT REPRESENTATION

Up to now we have discussed the binary representation only for positive numbers, in which it was implicit that any positive integer may be preceded by an arbitrarily long string of zero digits, which are then ignored. The representation of negative numbers requires further consideration. To use a practical case, we will illustrate the discussion by using whole numbers of length 32 bits, corresponding to the length of a **fullword** in memory and of a general register.

To begin with, suppose all of the binary digits of the number being examined are taken to be the rightmost 32 bits of any positive integer. Then

0	is represented by 00000000_{16},
1	is represented by 0000000116,
130	is represented by 00000082_{16},
2^{31}	is represented by 80000000_{16},
$2^{32}-1$	is represented by $FFFFFFFF_{16}$,
$2^{32}+1$	is represented by 00000001_{16}, and so on.

Thus, if the number is less than 2^{32} its value can be correctly held in the 32 bits we have made available, and if it is greater than or equal to 2^{32}, some significant bits are lost off the left end. (That is, the value of the number is represented modulo 2^{32}.) There are machine instructions which allow the CPU to perform addition and subtraction with operands of this form; -such arithmetic (modulo 2^{32}) is called logical arithmetic. Hence we call this the logical representation of binary numbers, where all the bits of the operand are interpreted as having "positive weight". (A "negative weight" for a digit will appear later in discussing negative numbers.)- That is, if the 32 bits are (from right to left; note that this temporary scheme is the reverse of the numbering convention introduced earlier) $b_0, b_1, \ldots b_{30}, b_{31}$, then the value X represented by the digits bi is

$$X = \sum_{i=0}^{31} b_i 2^i. \qquad \text{(logical representation)}$$

This representation is the most c-on way to interpret a string of bits. There are several representations used for numbers which can assume both positive and negative values, the most common of which are the sign-magnitude, one's complement, and two's complement representations. Since the last of these representations is used for most integer arithmetic in System/360, we will investigate its properties in detail. Actual arithmetic using binary numbers will be covered in subsequent sections.

The two's complement representation (the name will be explained shortly) of a positive integer x is (if x satisfies $0 \leq x \leq 2^{31}-1$) simply the usual binary representation with the least significant digit at the right-hand end; and is the same as the logical representation. The upper limit of $2^{31}-1$ is chosen because it is the largest-integer which can be represented using 31 binary digits; the remaining 32nd digit at the left-hand end is zero, and will be used for the sign digit. The two's complement representation of a negative integer x which satisfies $-2^{31} \leq x \leq -1$ is the following: the leftmost bit is now set to 1 to indicate that the number is negative, and the remaining 31 bits are aet to the binary representation of the positive integer $2^{31} + x$, which satisfies $0 \leq 2^{31} + x \leq 2^{31}-1$. In effect we have done the following: if x is positive, the sum $\Sigma b_i 2^i$ gives the value of x, because the leftmost bit, being zero, does not contribute to the sum. If x is negative, the sum of the rightmost 31 bits is $2^{31} + x$ and the leftmost bit is always a one, so that we can combine these to obtain

$$x = -2^{31}b_{31} + \sum_{i=0}^{30} b_i 2^i . \quad \text{(2's complement)}$$

This formula is almost the same as that used for the logical representation except that the leftmost bit (b_{31}) contributes negatively to the sum -- that is, has "negative weight". We will occasionally call the two's complement representation, where positive and negative numbers are allowed, the arithmetic representation.

The relationship between the logical and two's complement representation is quite simple, which may be seen by rewriting the above sum for X:

$$X = +2^{31}b_{31} + \sum_{i=0}^{30} b_i 2^i .$$

If b_{31} is zero, the logical and two's complement representations give the same value, and $x = X$. If b_{31} is one, then $X = x + 2 \times 2^{31} = x + 2^{32}$. But because we can only represent numbers less than 2^{32} in the logical representation, $x + 2^{32}$ for positive x is the same as X, with the extra bit being lost. Thus, for $0 \leq X \leq 2^{32}-1$ and $-2^{31} \leq x \leq 2^{31}-1$, we have

$$X = 2^{32} + x \text{ (modulo } 2^{32}).$$

(The above equation is the original source of the term 'two's complement". In the earliest computers it was customary to treat such fixed-point numbers as fractions -- the representation was the same as the one just deseribed, except that the "binary point" (the binary equivalent of the decimal point) was assumed to lie just to the right of the sign bit rather than at the right-hand end of the number. The equation giving the relationship between logical and arithmetic representations was then written $X = 2 + x$, so that the representation of a negative number was obtained by finding its complement with respect to two.)

The actual calculation of the binary two's complement representation of a negative number can be somewhat cumbersome. If the previous rule is followed, we must calculate the binary representation of the positive quantity $2^{31} + x$ for some negative x, and the conversion can be tedious. It turns out, however, that getting $2^{31} + x$ by calculating $(2^{31} - 1 + x) + 1$ is relatively simple, because the representation of $2^{31}-1$ is 31 one-bits. Since x is negative, $2^{31}-1 + x = 2^{31}-1 - |x|$. Thus the magnitude of x is subtracted from a string of 31 ones. But wherever $|x|$ has a one bit, the resulting difference bit will be 0, and vice versa. Thus the subtraction need not be done: simply change each bit into its opposite (namely the result of subtracting it from 1), and we have $2^{31}-1 - |x|$. (The result is called the one's complement of $|x|$.) Then add 1 in the rightmost position to get $2^{31} + x$, set the leftmost bit to 1, and there it is. And since $|x|$ when treated as a 32-bit number always has a leading zero digit, we can include the treatment of the sign bit in the following two-step prescription.

Given Y: find the **two's** canplement **representation** of **-Y.**

1) Take the one's complement of Y (change all 0 **digits** to 1 and all 1 digits to 0).

2) Add a 1 digit in the low-order (rightmost) position, and ignore carries out of the leftmost position.

To illustrate this process, consider the following two examples in which the arithmetic is done with eight binary digits for the sake of simplicity.

1. Find the two's complement representation of -2.

   1) Representation of +2: $0000\ 0010_2$

   2) **One's Complement:** $1111\ 1101$

   3) Add one: $\underline{+1}$

   $1111\ 1110_2$

2. Find the two's **complement** of +75.

   1) Representation of +75: $0010\ 1011_2$

   2) One's Complement: $1101\ 0100$

   3) Add one: $\underline{+1}$

   $1101\ 0101_2$

The above prescription also works in the opposite direction, which can be seen **from** the following example.

Find the 8-bit two's complement of $1111\ 1110_2$.

   1) One's Complement: $0000\ 0001$

   2) Add one: $\underline{+1}$

   $0000\ 0010_2$

which is the binary representation of +2. Thus the two's **complement** of the two's complement of a number is the original number.

There are two unusual cases which arise in the two's complement representation: the complement of zero and of the largest negative number.

1 . Find the 8-bit two's complement of $0000\ 0000_2$.

   1) One's Canplement: $1111\ 1111$

   2) Add one: $\underline{+1}$

   (carry one) $0000\ 0000$

To the 8-bit accuracy chosen, the result is zero, and the carry of a 1 bit out the left-hand end is lost. Thus the negative of zero is still zero, which is a mathematically satisfying result; there is no such quantity as a negative zero, which can be the case in some other representations.

2. Find the 8-bit two's complement of 1000 00002.

```
    1)  One's  Complement:  0111 1111
    2)  Add one:                   +1
                           ----------
                            1000 00002
```

It can be seen in this case also that the complement of the number is the same as the original number.

Thus we see that the two unusual cases which arise during complementation are those for which all the bits except the sign bit are zero, and it is found that the complemented result is the same as the original operand. For a zero operand this is desirable, but for the negative case we have a situation in which there is no corresponding positive value available for a representable negative value. Such a situation is described by saying that we have generated an <u>overflow</u> condition -- that is, the result is too large to fit into the number of bits allotted for it. Overflow will be treated in more detail in the following section on two's complement arithmetic. We will note in passing that the number of quantities with negative representation is the same'as the number of quantities with positive representation, since the non-sign bits of the number may be chosen arbitrarily. It is sometimes said that the set of negative values in the two's complement representation has one more member than the set of positive values; what is meant is simply that the largest negative magnitude is larger by one than the largest positive magnitude.

Decimal Value	32-bit Two's Complement Representation
0	0000 000016
1	0000 0001_{16}
256	0000 010016
5000	0000 1388_{16}
2147483647(2^{31}-1)	7FFF FFFF_{16}
-2147483648(-2^{31})	8000 000016
-2147483647(-2^{31}+1)	8000 0001_{16}
-5000	FFFF $\mathrm{EC78}_{16}$
-256	FFFF $\mathrm{FF00}_{16}$
-2	FFFF FFFE_{16}
-1	FFFF FFFF_{16}

Figure 6.1 Examples of Two's Complement Representation

As was mentioned earlier, it is implicit in the representation of positive numbers that an arbitrary number of zero bits may be added onto the left end of a number without affecting its value. For example, the 8-bit and 16-bit representations of the decimal value +9 are 0000 10012 and 0000 0000 0000 10012, respectively. Similarly, the 8-bit and 16-bit two's complement representations of -9 are 1111 0111_2 and 1111 1111 1111 0111_2, respectively. Thus, for numbers which can be correctly represented in a given number of bits, the correct representation using a larger number of bits is found by simply duplicating the sign bit toward the left as many places as desired. This process is called sign extension.

Length of Representation	Representation of +1	Representation of -1
8 bits	01_{16}	FF_{16}
16 bits	0001_{16}	FFFF_{16}
32 bits	0000 0001_{16}	FFFF FFFF_{16}
64 bits	0000 0000 0000 0001_{16}	FFFF FFFF FFFF FFFF_{16}

Figure 6.2 Examples of Sign Extension

Sign extension will appear later in the discussion. of instructions which perform shifting, and which do arithmetic with halfword operands.

7. TWO'S COMPLEMENT ARITHMETIC

Arithmetic operations on numbers in a binary representation are a basic capability of almost all computers. Though the details of the number representation may vary slightly from one machine to another, the methods for performing additions, subtractions, multiplications, and divisions remain nearly the same for all machines. Thus the discussion which follows will be slightly more general than would be necessary if only one particular model of the System/360 series were being discussed.

We have already used some examples of binary addition in the treatment of addressing, in which the addition was straightforward. The rules for the addition of binary digits are summarized in the following short table.

+	0	1
0	0	1
1	1	0, carry 1

The addition of numbers in the logical representation is the most straightforward, since the bits are all numeric digits and do not represent signs. Thus the only unusual condition to observe in such an addition is whether or not a carry occurs out of the leftmost position, which would indicate whether the resulting sum is or is not representable by the number of bits available. In the two's complement arithmetic representation, the addition is performed in the same way, but the result is interpreted somewhat differently. (1) All bits of each operand are added, including sign bits, and carries out the left end of the sum are lost. (This is the same as for logical addition.) (2) If the result cannot be correctly represented using the number of digits available, an overflow condition is said to have occurred. Note that overflow is possible only when adding operands of like sign: adding numbers with opposite sign always produces a representable result (or, as is often said, the result is in range). When an overflow occurs, the sign of the result is always the opposite of the sign of the two participating operands. The actual method used on most machines to detect overflow is somewhat simpler, since the sign-change detection would require remembering the signs of both operands for comparison against the sign of the sum. In practice, the adding circuits need only note that the carries into and out of the sign bit position disagree, to be able to detect overflow: that is, if the carries out of the two leftmost bit positions differ, an overflow has occurred.

Subtraction is performed in the machine by adding the two's complement of the number to be subtracted. That is, A-B is calculated using A + (-B), where (-B) is the two's complement of B. A few examples using 8-bit arithmetic will illustrate the methods of addition and subtraction.

```
1.  5-3:            0000 0101     becomes        0000 0101
                   -0000 0011                   +1111 1101
                                 (carry lost)    0000 0010   = $2_{10}$

2.  3-5:            00000011      becomes        0000 0011
                   -0000 0101                   +1111 1011
                                 (no carry)      1111 1110   = $-2_{10}$

3.  25-(-17):       0001 1001     becomes        0001 1001
                   -1110 1111                   +0001 0001
                                 (no carry)      0010 1010 . = $42_{10}$

4.  (-17)-25:       1110 1111     becomes        1110 1111
                   -00011001                    +1110 0111
                                 (carry lost)    1101 0110   = $-42_{10}$

5.  -17-(-25):      1110 1111     becomes        1110 1111
                   -1110 0111                   +0001 1001
                                 (carry lost)    0000 1000   = $8_{10}$

6.  67-(-93):       0100 0011     becomes        0100 0011
                   -1010 0011                   +0101 1101
                                 (no carry)      1010 0000   = $-96_{10}$ (overflow)

7.  (-93)-67:       1010 0011     becomes        1010 0011
                   -0100 0011                   +1011 1101
                                 (carry lost)    0110 0000   = $96_{10}$ (overflow)

8.  -128-(-93):     1000 0000     becomes        1000 0000
                   -1010 0011                   +0101 1101
                                 (no carry)      1101 1101   = $-35_{10}$
```

```
9.  3-3:          0000 0011      becomes      0000 0011
                 -0000 00112                 +1111 1101
                            (carry lost)      0000 0000    =  0
```

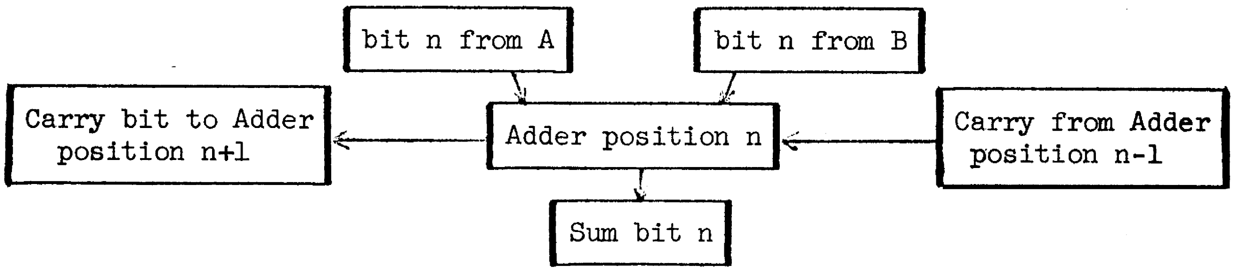
The above examples illustrate addition and subtraction and give the expected results. However, there is one case in which the method as given above fails to detect correctly the presence or absence of **overflow,** and this occurs when the maximum negative number is being subtracted from something.

```
10.  1-(-128):    0000 0001      becomes      0000 0001
                 -1000 0000                  +1000 0000
                             (no carry)       1000 0001   (no overflow found)

11.  -1-(-128):   1111 1111      becomes      1111 1111
                 -1000 0000                  +1000 0000
                            (carry lost)      0111 1111   (overflow indicated)
```

In each of these two last cases the overflow indication is incorrect. This is because the process of taking the two's complement of the maximum negative number has already generated an overflow condition. To see how the computer can still use the overflow detection scheme described above, it is worth examining in slightly more detail the actual addition process in the machine. (The next paragraph may be omitted by those uninterested in such details.)

Remember that the two's complement of a number is found by inverting each bit of the number and then adding a one in the low-order position. It is very easy to build circuits which invert bits; similarly, the addition of a 1 bit to the low-order position is also easy, for the following reason. Each digit position of the adder circuits must add the corresponding bits of the two input operands and the carry-bit from the next lower-order bit position.


bit n from A
bit n from B
Carry bit to Adder position n+1
Adder position n
Carry from Adder position n-1
Sum bit n

In the lowest-order position of the adder there of course can be no carry from a lower-order bit position; if an identical adder circuit is used, however, the carry input is still there, and can be used to insert the 1 to be to be added to the low-order position. Thus subtraction is simply a matter of passing the second operand B through a bit inverter which **forms** the one's complement, and then activating the low-order carry input to the adder to add the 1.

Thus we arrive at the following rule:

> Subtraction is performed by adding the one's complement of the second operand and a low-order one to the first operand.

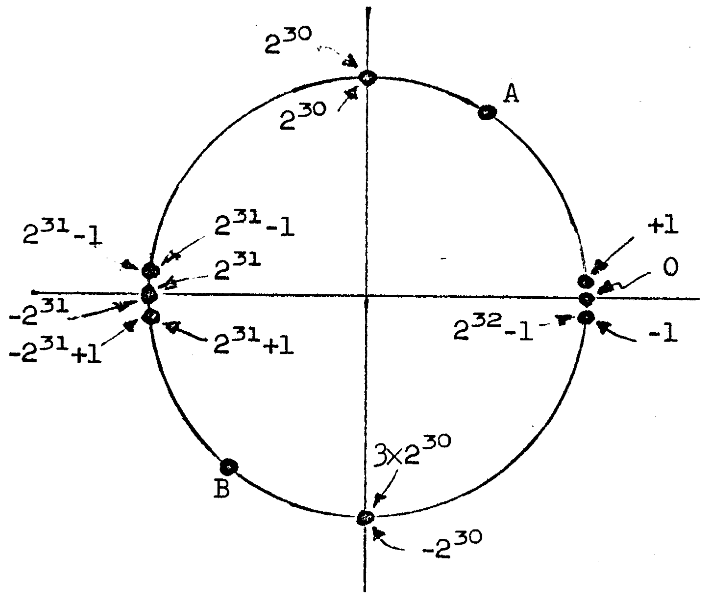
It is easy to demonstrate that the correct algebraic result is obtained by simply adding all the bits of the operands in the two's complement representation as though they were logical operands. Since the logical representation X corresponding to an integer x satisfies (assuming 32-bit operands) $X = 2^{32} + x$ (modulo 2^{32}), then the sum of two operands X and Y is

$$(x + Y) = 2^{32} + 2^{32} + (x + y) \pmod{2^{32}} = 2^{32} + (x + y) \pmod{2^{32}}.$$

Thus the arithmetic and logical sums give the same binary result; the bits are just interpreted differently for each representation.

One further observation may be made concerning the addition and subtraction of numbers in the logical representation. From the examples given above it can be seen that if the second operand is logically smaller than or equal to the first (see examples 1, 4, 5, 7, 9, and 11) then there will be a carry out of the leftmost bit position. It may be seen in examples 2, 3, 6, 8, and 10 that if the first logical operand is logically smaller than the second operand subtracted from it, there is <u>no</u> carry out of the left end. In these latter cases we have in some sense generated a "negative" logical answer, since the result is not correctly represented to the given number of bits. A number of examples illustrating these cases will be given later, when the instructions for logical arithmetic are discussed.

There is a simple pictorial representation of the two's complement representation which is helpful in seeing what happens when two such numbers are added or subtracted. The circle is visualized as having 2^{32} points on its circumference, arranged as indicated. Arithmetic values are on the outside of the circle, logical values on the inside.



If we begin at 0 and add 1 to a number, we will move around the circle in a counter-clockwise direction until $2^{31}-1$ is reached. When 1 is added again, we reach -2^{31} and an overflow condition exists. Continuing to add 1 then brings us back to 0. It can be seen that adding a positive number to or subtracting a negative number from an existing number (say, A, as on the circle) causes us to move in a counter-clockwise direction. If in moving in this direction we go past the point labeled -2^{31}, an overflow occurs. Similarly, adding a negative number to or subtracting a positive number from an existing number (say, B, on the circle) causes us to move in a clockwise direction; and if the motion carries us past the point labeled- -2^{31}, we again have an overflow condition.

8. BINARY MULTIPLICATION AND DIVISION

Before we discuss the actual machine instructions which perform multiplication and division using integer arguments, it will be useful to examine a few simple illustrations of the basic method used by typical computers to form products and quotients of binary numbers. A detailed understanding of the methods is of course not necessary to be able to use the corresponding instructions, but will help in remembering a number of conventions that these instructions require;

Multiplication

To illustrate the method used in multiplication, let us first work an example in decimal arithmetic. Suppose we have a "machine" with registers which will hold j-digit decimal numbers, which we will assume are positive. Let the numbers to be multiplied by 126 and 213. First of all, since we are multiplying two 3-digit numbers, the product will be either 5 or 6 digits long. Thus if we are to be able to correctly represent it, the product register must be at least 6 digits long. Since we assumed the number registers were 3 digits long, it appears that we need a double-length register (or a pair of registers connected in some way) to hold the product. So we will assume there is a 6-digit register somewhere, the right and left halves of which will hold an ordinary 3-digit number. Now let us examine the way in which we normally form such a product, as when working with pencil and paper. By taking the product of the multiplier and each of the multiplicand digits in succession, we generate a series of partial products which must be properly aligned and then added. (Note that we are using the terms "multiplier" and "multiplicand" in the reverse of their normal meaning; this is done so as to be consistent with the terminology used in other descriptions of System/360.) This manual process can be

```
multiplier        126
multiplicand    x 213
                  378
   partial       126
   products     252
   product      26838
```

broken down even more, by writing the sequence of operations in a different way.

initial register contents	000	213
add multiplier to upper end	+126	
that's 1 time	126	212
add multiplier	+126	
that's 2 times	252	211
add multiplier	+126	
that's 3 times	378	210
shift right 1 place	037	821
add multiplier	+126	
that's 1 time	163	820
shift right 1 place	016	382
add multiplier	+126	
that's 1 time	142	381
add multiplier	+126	
that's 2 times	268	380
shift right 1 place	026	838

We place the multiplicand in the right half of the double-length register and clear the left half to zero. Then by examining the rightmost digit of the multiplicand we know how many times to add the multiplier to the left half of the double-length register. When the rightmost digit has been counted down to zero, the partial product of that digit and the multiplier has been added to the accumulating result. Then the entire double-length register is shifted to the right one digit position, at which time the zero digit at the right-hand end is lost and a zero digit is inserted in the vacated position at the left. The process of adding the multiplier and counting down on the multiplicand digit then continues until the proper partial product has been added to the accumulated result. This process is repeated for as many steps as there are multiplicand digits. When completed, the result in the double-length register is the product, and all the multiplicand digits have been shifted off the right-hand end. The main points to observe are that (1) the multiplicand is placed in the right half of the double-length register, (2) the left half is initially cleared to zero, (3) the multiplier is added to the left end depending on the multiplicand digit at the far right, and (4) the decimal point of the result (that is, the position of the least significant digit) is at the right-hand erd of the double-length register, because the number of right shifts was the same as the number of digit positions in a single-length register.

The above example omits one rather important detail which is not actually necessary to an understanding of the basic process. (These two paragraphs concern technicalities, and may be skipped with little loss of continuity.) When the multiplier is being added to the left half of the double-length register, it is possible that an overflow can occur. If the multiplicand had been 219 rather than 213, the first partial product (126 x 9 = 1134) would have been too large to hold in the three digits provided. Thus provision must actually be made for an extra digit at the leftmost end of the register. This extra digit can be thought of as hidden from the user of the registers, since when the right shift is performed at the conclusion of each cycle, the contents of this "overflow digit" position move into the leftmost digit of the double-length product register. Since the example was carefully contrived to avoid the necessity of worrying about this detail, the presence of a zero digit at the left end after the right shift is seen simply to be an-indication that there was no overflow in the formation of the partial product. The assumed presence of this extra digit position will be useful in the discussion of division.

This small but annoying difficulty can also be handled by having the extra "digit position" attached after the rightmost digit of the double-length register. Then instead of adding and then shifting, we could first shift and then add. Thus the extra digit position will hold the number of times the multiplier is to be added. However, the additions of the multiplier must then be realigned so as to add to the second, third, and fourth digits of the double-length register rather than the leftmost three. Either way, the whole business is a necessary nuisance. (These comments will of course apply to the binary multiplication example which follows.)

The above scheme, when used for multiplying <u>binary</u> numbers, is conceptually very easy to implement since a test of the rightmost bit determines in simple yes-no form whether or not the multiplier is to be added -- no counting of additions is required. To illustrate this, suppose we have y-digit binary numbers and registers and wish to multiply 00110_2 by 010012 to obtain a 10-bit product in a double-length register. Then the sequence of steps shown below indicates the method.

	00110	multiplier (in separate register)
Initialize	00000 $0100\underline{1}$	multiplicand in right half of double-length register
Step 1: rightmost bit = 1, add multiplier	00110 01001	
Shift right 1	00011 $0010\underline{0}$	(1 bit lost)
Step 2: rightmost bit = 0, no add. Shift right 1	00001 $1001\underline{0}$	
Step 3: rightmost bit = 0, no add. Shift right 1	00000 $1100\underline{1}$	
Step 4: rightmost bit = 1, add multiplier	00110 11001	
Shift right 1	00011 $0110\underline{0}$	(1 bit lost)
Step 5: rightmost bit = 0, no add. Shift right 1	00001 10110	final product = 1101102 = 54_{10}

It is most important to observe that the product is really a double-length number, and not simply two single-length numbers stuck end to end. If we were to consider the contents of the left and right halves of the double-length register as ordinary single-length two's canplement operands, we would find the result in the right, or low-order half, to be negative! Since the product (which was computed from two positive numbers) must be positive, it can be seen that the need for a double-length register means that no special significance can be attached to the low-order result, unless it is known in advance that the product is correctly representable in a single register. The leftmost bit of the right-hand register is therefore not a sign bit -- it has positive weight in the double-length result.

In the example above, the two operands were purposely chosen to be positive so as not to introduce any problems with signs. Since the operands actually used may be positive or negative two's complement integers, there are other steps which must be taken to find the correctly signed product. For all practical purposes, however, we may assume that the CPU performs the multiplication by using the magnitudes of the operands, and then complements the double-length result if a sign-bit analysis of the original operands indicates that the result is negative.

It is also common in modern computers to gain speed by considering not the rightmost single bit of the multiplicand (as on the IBM 7090), but to consider the rightmost two bits (IBM 7094), three bits (Burroughs 5500), or even four bits (larger models of System/360). This of course brings us back to a situation similar to that in the decimal example, where the proper multiple of the multiplier must beaded to the left end of the developing product. In these cases, where the arithmetic can be considered to be of base 4, 8, or 16, the "proper multiple" is of course not found by counting down by ones on the multiplicand digit, but by having the internal circuits generate the proper factor in a very much smaller number of steps. This serves to increase the speed of multiplication considerably, since then a separate addition is not required for each 1 bit detected in the multiplicand.

Division

Division works the same as -multiplication, only backwards. Instead of adding onto the high-order half of the accumulating product, we subtract; instead of counting down in the rightmost digit position, we count up; instead of shifting right, we shift left. As before, an example using 'decimal arithmetic will illustrate the process.

Since we start with a dividend and divisor and wish to find a quotient and remainder which satisfy the equation

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder},$$

it is apparent that the dividend must be a double-length number. Again supposing that the basic register length is three decimal digits, another requirement becomes apparent: since (a) the quotient, to fit in a register, can be at most three digits long (that is, not exceeding 999) and (b) the remainder must be less than the divisor, we must not have a dividend larger than

$$999 \times \text{divisor} + (\text{divisor} - 1) = 10^3 \times \text{divisor} - 1.$$

(The factor of 10^3 is the base raised to the number of available digits.) Since multiplication by 10^3 in this example is equivalent to shifting left three places, the above relation means that if the division is to produce a valid'quotient, the high-order half of the dividend must be <u>less</u> than the divisor. (If for instance the divisor were 456, then any dividend not smaller than $456000 = 10^3 \times 456$ would require a 4-digit quotient; if the dividend is not greater than 455999 = 10' X456 - 1, the the quotient can be held in the three digits allotted. Note that the three high-order digits, 455, are now less than the divisor.)

Suppose we want to divide 162843 by 762. In ordinary long division, we would do the following sequence of steps. At each step we determine how many multiples of the divisor can be subtracted from the leftmost part of the dividend, and enter that number as the quotient digit. When the subtraction process has been completed, the remainder, from which no further subtractions can be made, is 537, and the quotient is 213. Just as a check, we find that

```
       213
762)162843
    1524
     1044
      762
      2823
      2286
       537
```

762 x 213 + 537 = 162843. On a machine, the process is almost identical. Using the above scheme of decimal registers, the division works as follows:

162 843	High-order part of dividend smaller than divisor,
762	division may proceed.
1 628 430	Shift dividend left once; save leftmost digit in an
-762	"overflow digit" position. Since dividend ≥ divisor,
0 866 431	subtract, and count up at right end.
-762	dividend ≥ divisor; subtract again
0 104 432	dividend < divisor; no subtraction
1 044 320	shift dividend left again
-762	dividend ≥ divisor; subtract and count up
0 282 321	dividend < divisor; no subtraction
2 823 210	shift left for last time-
-762	dividend ≥ divisor; subtract
2 061 211	subtract and count up by 1
-762	dividend ≥ divisor; subtract
1 299 212	subtract and count
-762	dividend ≥ divisor; subtract
537 213	dividend now < divisor; stop

As the successive digits of the quotient were developed, they appeared at the right hand end of the double-length register, and were shifted left as the division progressed. Thus at the completion of the division, the quotient is to be found in the right half of the register pair, and the remainder, from which no further subtractions could be made, is in the left half.

As was the case for multiplication, binary division is simplified by the fact that at most one subtraction need be made for each quotient digit generated. To illustrate, consider this example using a five-bit divisor and a ten-bit dividend. Let the dividend be 0000111011~ = 59_{10}, and let the divisor be 00110_2. Note that the two halves of the double-length dividend are not two five-bit numbers stuck end to end: the leftmost bit of the right half of the dividend is not a sign bit (with negative weight) but an arithmetic digit (with positive weight). The quotient and remainder, however, are ordinary (i.e., signed two's complement) five-bit numbers, so that when the division is complete the proper results are found in each register . This leads to the following scheme.

1. Shift the dividend left once. If the high-order (left) part of the dividend is not smaller than the divisor, an illegal division is being attempted.
2. Shift left one bit position. If the high-order part of the dividend is greater than or equal to the divisor, subtract the divisor from the dividend and insert a 1 bit in the rightmost digit position. Otherwise do nothing.
3. Return to step 2 until a total of 5 shifts has been done including the shift of step 1. (For 32-bit operands this cycle repeats 31times.)

00011 10110	shift left once
(00110)	dividend < divisor, OK to continue
00111 01100	shift left once (second time)
00001 01101	subtract divisor, insert 1
00010 11010	shift 'left once (third time)
	dividend < divisor; no subtraction
00101 10100	shift left once (fourth time)
	dividend < divisor; no subtraction
01011 01000	shift left once (fifth and last time)
00101 01001	subtract divisor, insert 1.

Thus the remainder $00101_2 = 5_{10}$ in the left half, and the quotient $01001_2 = 9_{10}$ in the right half are as expected.

The example given assumed a positive dividend and divisor; if either is negative some further steps are necessary. The division can be thought of as proceeding with the magnitudes of divisor and dividend, and afterwards the quotient is made negative if the signs of the divisor and dividend differed, and the remainder is made negative if the dividend was negative.

As in the case of multiplication, there are techniques used for speeding up the division process which are used on some models of System/360. These details are of concern only to the machine designer, so that the programmer can think of division as proceeding through the simple steps shown above.

9. ASSEMBLER LANGUAGE

As was indicated in the introduction, the service program which will be of most use in setting up instruction sequences for execution by the machine is the Assembler. The collection of conventions and rules established for use of the Assembler is known simply as Assembler Language, even though there is no resemblance to what we usually mean by the term "language".

Before describing some of the basic conventions used in communicating with the Assembler, it may help to consider first the overall process of running a machine-language program on the computer. This process may be broken down into five major parts, as follows: (1) job initiation, (2) assembly, (3) linkage editing, (4) execution, (5) job termination.

1. Job initiation will usually involve the checking of the job information provided by the programmer, such as charge number, time and page estimates, and so forth, as required by the particular computer installation. If these details are acceptable, then preparations are made for the execution of a series of job steps, which in this case will include assembly, linkage editing, and execution.

2. The assembly step is represented schematically in Fig. 9.1. The Assembler is a processing program (a previously prepared set of machine instructions) which is placed in the memory of System/360 and is allowed to begin execution.

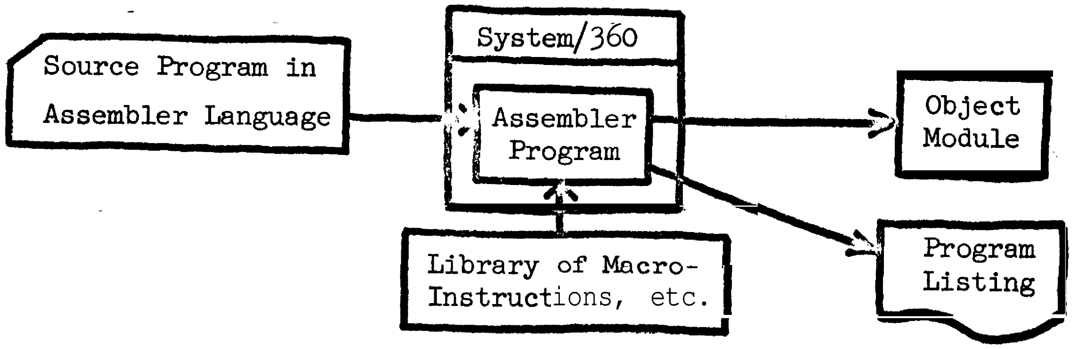

Source Program in Assembler Language
System/360
Assembler Program
Object Module
Library of Macro-Instructions, etc.
Program Listing

Figure 9.1 Simplified Schematic of Assembler Processing

The Assembler reads the statements (to be described shortly) of the programmer's Assembler Language program, processes them -- possibly with the help of some pre-stored data in the library of macro-instructions (also to be described later) -- and eventually produces as its output an object module, which will usually be written onto some storage device such as a magnetic drum or disk. (The object module may also be punched on cards, so that a programmer could then have his program in both its original form and in its assembled form.) Usually the programmer will want a program listing, which is printed output giving the source program and pertinent details of the Assembler's processing, along with indications of any errors detected by the Assembler.

3. The linkage editing step is shown schematically in Fig. 9.2. The Linkage Editor, like the Assembler, is a processing program which is placed in memory and allowed to begin execution.

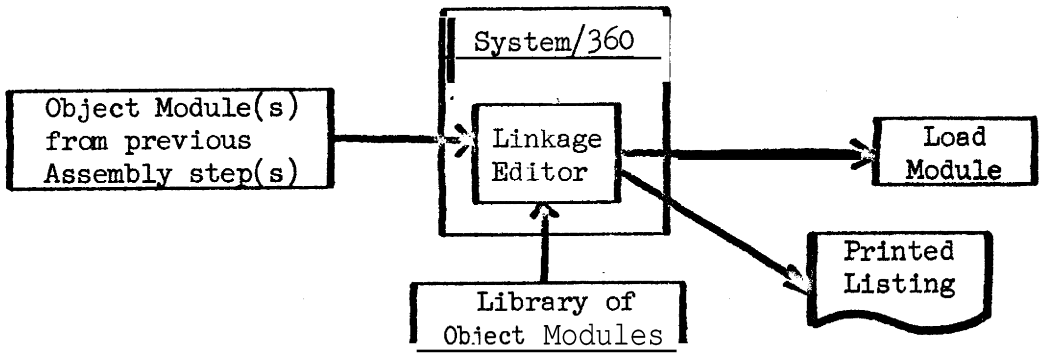

System/360
Object Module(s) from previous Assembly step(s)
Linkage Editor
Load Module
Printed Listing
Library of Object Modules

Figure 9.2 Simplified Schematic of Linkage-Editor Processing

The Linkage Editor reads the object module (or modules; cases in which several may appear will be described later) and combines it with other object modules that may be necessary for proper program execution. The output produced is the completed program and is called the load module, which is written onto a storage device for later use. A printed listing of information pertinent to the link-edit step may also be produced.

4. The execution step requires that the load mpdule produced by the Linkage Editor be placed in (or "loaded" into) memory, in such a way that it will execute correctly (assuming, of course, that the programmer has made no blunders!). An essential feature of this process is relocation, details of which will be treated in several later sections.

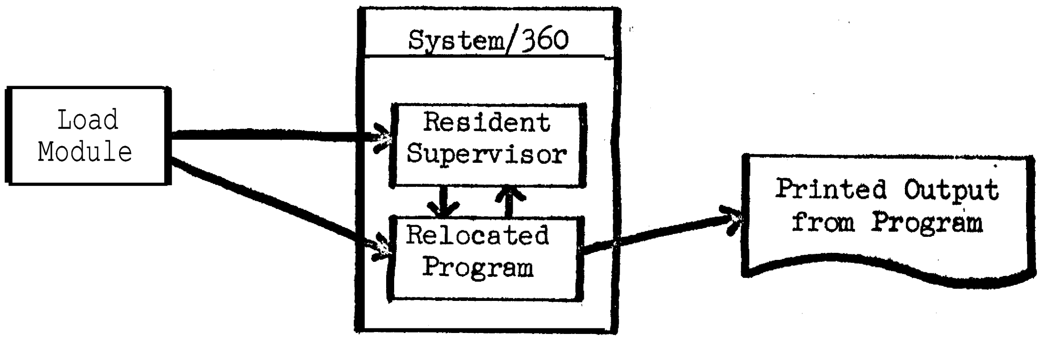


Figure 9.3 Simplified Schematic of Program Loading and Execution

When the program has been loaded and relocated, the Resident Supervisor transfers control to the program (that is, sets the Instruction Address to the address of whatever instruction was specified as the one with which execution is to begin). The program then performs whatever processing was specified by the programmer, and when it is finished returns control to the Supervisor (that is, sets the IA to an agreed-upon value so that the Supervisor may continue processing the next job).

5. When the Supervisor program has regained control it performs any necessary "cleaning-up" operations such as noting the amount of time used by the job, -the number of pages printed, and so on. If more jobs are to be done, the Supervisor reverts to step 1 (Job Initiation) and the entire cycle repeats.

The brief description of job processing given above will help in understanding some of the constructs necessary to the writing of a correct Assembler Language program, since certain of them apply during each of the assembly, link-edit, and execution steps and must be used with the different steps in mind.

A program is prepared for the Assembler in the form of statements punched on cards. Statements are of four general types: comment statements, machine instruction statements, assembler instruction statements, and macro-instruction statements. Comment statements are used by the programmer to insert explanatory material in the program so that it will be easier to read and understand the program listing. Machine instruction statements contain instructions which the computer may execute during the execution step of the job. Assembler instruction statements contain information of use to the assembler during the assembly step; these can be as simple as a statement specifying that four blank lines are to be left in the program listing, or can be more complicated such as a statement which informs the Assembler that it may assume certain registers may be used as base registers. (This latter case will be treated in detail in Section 12.) Finally, macro-instructions provide a convenient means for specifying sequences of statements (all four types are allowed) in which various parts of the specified sequence can be changed to suit the needs or desires of the programmer. We will see later that the ability to process macro-instructions is a very powerful and useful feature of the Assembler Language.

The Assembler provides a number of other capabilities which considerably simplify the programmer's task. For example, we saw in Section 5 that a' typical machine instruction might consist of 8 hexadecimal digits. Rather than having to remember that the operation code 43_{16} causes a byte to be transferred from memory to the right-hand end of a general register, a mnemonic operation code is provided which gives an easily-remembered abbreviated description of what the operation code does. In the above case, the mnemonic is IC, which stands for "Insert Character", character in this case being synonymous with byte. Another useful feature is that the Assembler allows us to specify information in a variety of forms: as decimal, hexadecimal, and binary numbers, as strings of characters, as arithmetic expressions, and so on. Thus we will find that if we want to designate register 15 for some use, we can use the decimal number 15 instead of having to use the hexadecimal digit F, which is what may eventually appear in the instruction itself. A third and most important feature of the Assembler is the provision for symbols which may be used by the programmer to name places in memory. Thus, if a program needs to make reference to a fullword area in memory which contains a particular piece of data, the Assembler will permit the programmer to name the fullword and then to make references to the data by using the name. A discussion of symbols and certain aspects of their use will be given in the next section. In the remainder of this section we will give some examples of statements, and define or illustrate terms which will be used in describing statements.

In general, statements occupy columns 1 through 72 of a card, with column 72 having a special meaning: if column 72 is not blank, it means that the next card is to be considered as a continuation of the card with the non-blank character in column 72, in such a way that column 16 of the second card is considered to follow immediately after column 71 of the first. (These numbers are actually under the control of the programmer, who may specify with an assembler instruction statement that other card columns are to be used for the start and end of a statement. The numbers given are simply the usual ones which the Assembler will assume are to be used if it is not told otherwise.) (It is a common error for beginning programmers to punch characters in column 72 unintentionally, so that the next statement is processed in an unexpected way.) Columns 73 through 80 are ignored by the Assembler when it processes the statement, and may be used for identification or sequencing information.

A comment statement is identified by the presence of an asterisk (*) in column 1. Any information desired may appear in columns 2 through 71. An example of a comment statement appears below, as it would be punched on a card.

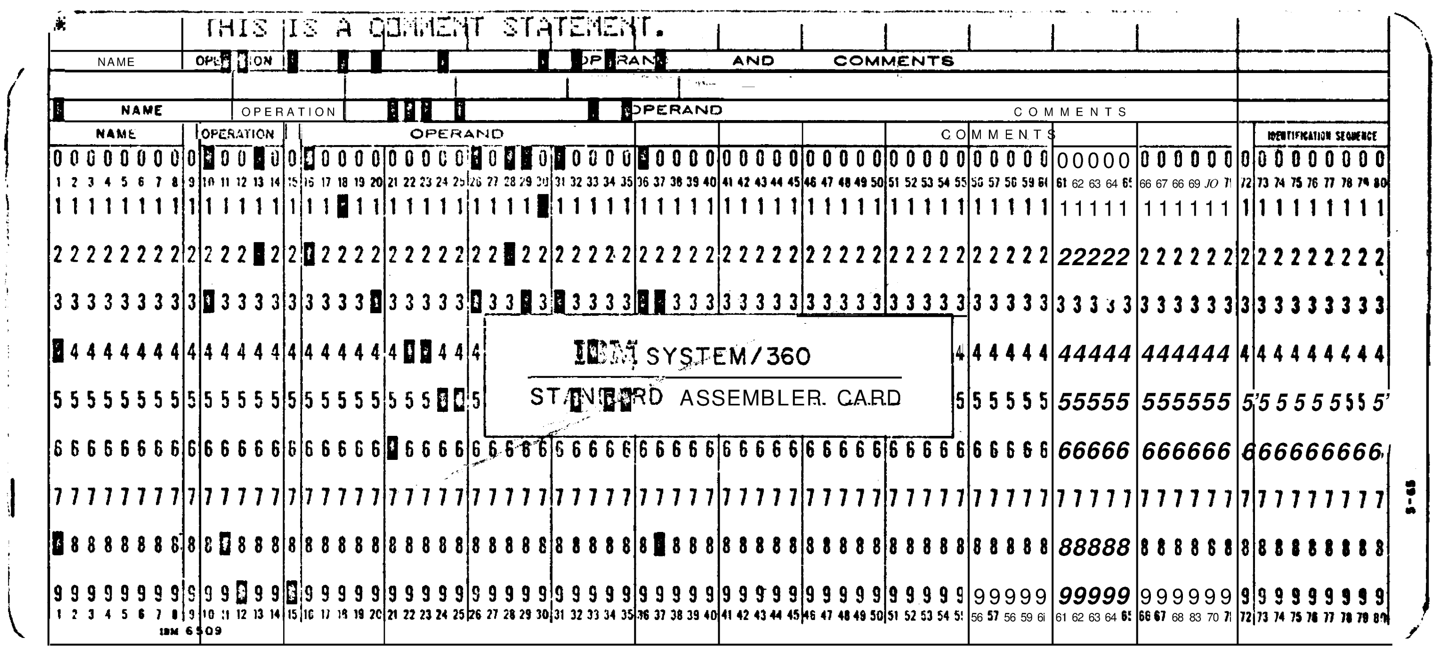


Figure 9.4 A Comment Statement

The machine instruction statement, assembler instruction statement, and macro-instruction statements each have four parts called fields. They are respectively the name, operation, operand, and comment fields; of these, an entry in the operation field must always be present, and for certain . types of statements entries in some of the other fields may or must be omitted. If there is a name field entry in the statement, it must begin with a non-blank character in column 1; it is terminated by the first blank column after column 1. If no name field entry is desired, column 1 must be left blank. After the name field, and separated from it by one or more blank columns, comes the operation field entry; it ends with the first blank-column after the start of the -operation field. After the operation field entry and separated from it by one or more blank columns comes the operand field entry which, like the name and operation field entries, terminates (except for one unusual case to be described later) with the first blank column detected after the start of the operand field. The rest of the card is treated as comments (that is, it is ignored) by the Assembler, and does not influence the processing of the statement (unless, of course, the comment field extends into column 72 indicating a continuation on the next card). Note that with the exception of the name field, no requirement is made regarding the columns in which the other three fields must start; they simply end with a blank column. This allows what are called free-field statements, in which the programmer may arrange the information on the cards of his program as he desires, with the only restriction being that the fields appear in the proper order.

The figure below illustrates a machine instruction statement in which entries in all four fields appear, and which if executed in a program would cause the contents of general register 7 to be replaced by the contents' of general register 3. (The particular form of the operand field entry will be discussed later.)

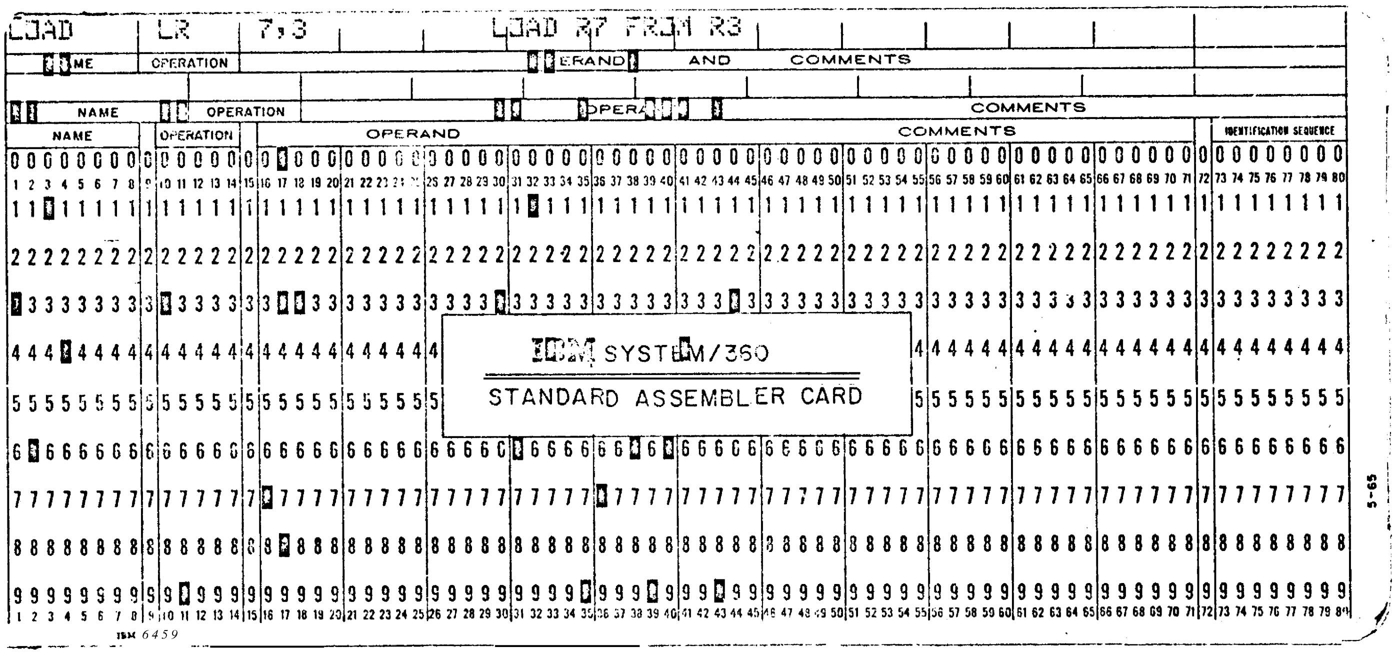

LOAD LR 7,3 LOAD R7 FROM R3
NAME
OPERATION
OPERAND
AND
COMMENTS
IDENTIFICATION SEQUENCE
IBM SYSTEM/360
STANDARD ASSEMBLER CARD
IBM 6459
5-65

Figure 9.5 A Machine Instruction Statement

An assembler instruction statement (in which the name and comment field entries are omitted) which would cause the Assembler to leave four blank lines in the program listing is given in the following figure.

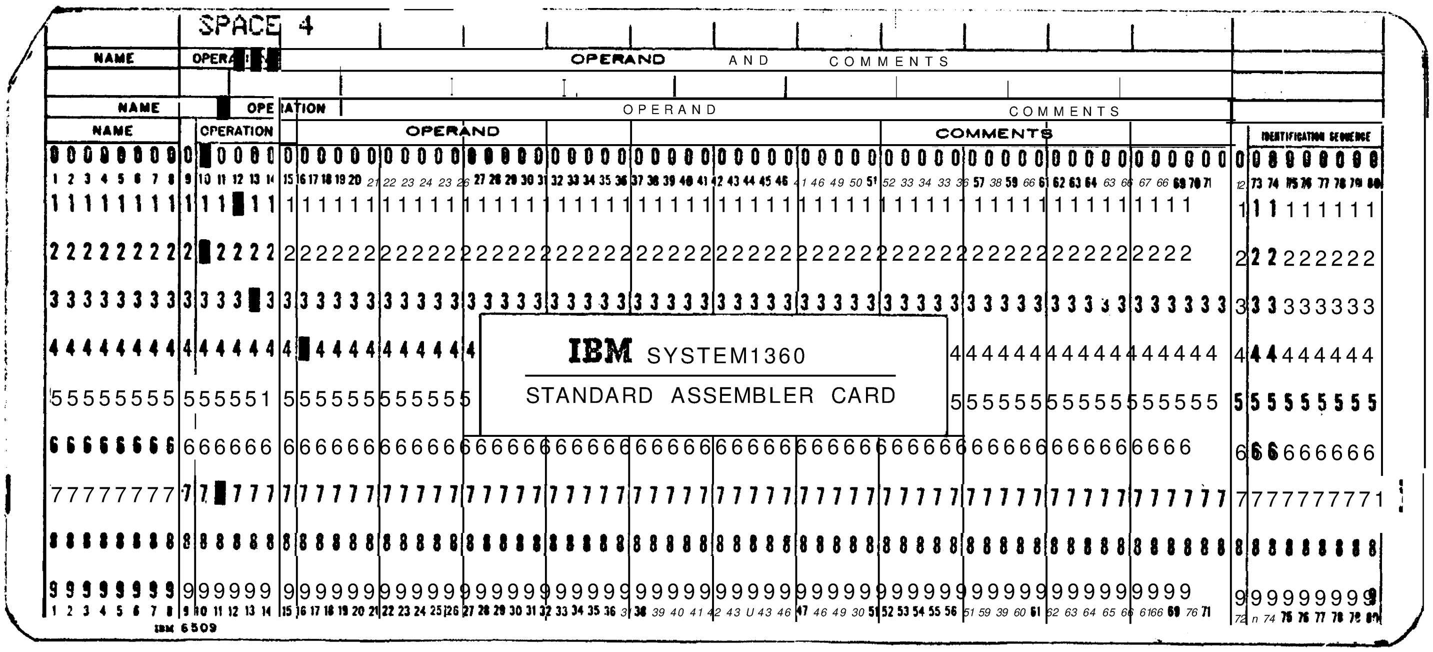


Figure 9.6 An Assembler Instruction Statement

Finally, an example of a macrd-instruction statement in which only the operation field entry appears is given below.

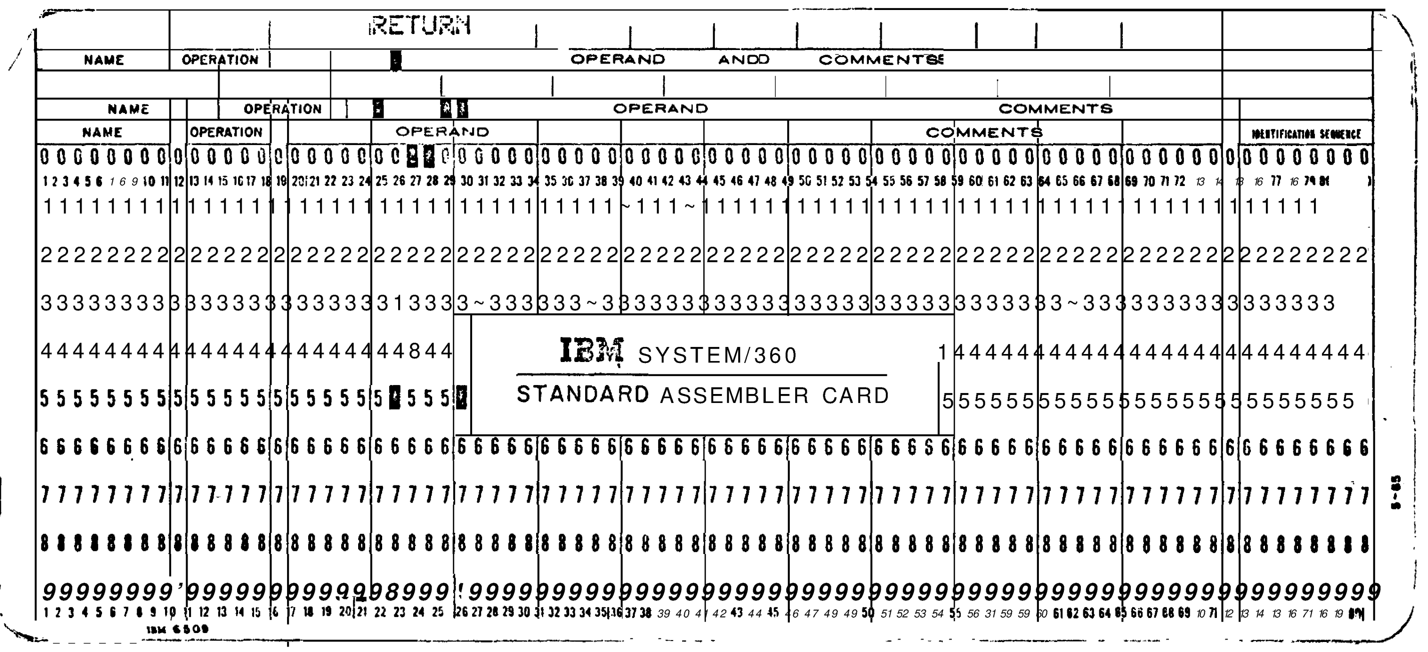


Figure 9.7 A Macro-Instructfon Statement

10. SELF-DEFINING TERMS AND SYMBOLS

In using the Assembler Language, two constructs of importance are self-defining terms and symbols. Each has a value; in self-defining terms the value is inherent in the term, whereas values are assigned to symbols by the Assembler (under control of the programmer, of course).

There are four types of self-defining terms: decimal, hexadecimal, binary, and character; the value of each is always taken to be positive.

A decimal self-defining term is simply an unsigned string of decimal digits. 12345, 98, and 007 are examples of decimal self-defining terms. The size of a decimal self-defining term is limited by the fact that 24 bits are allotted by the Assembler to hold its value; hence a decimal self-defining term must (a) contain 8 or fewer digits and (b) be less than or equal to $2^{24}-1$ = 16777215.

A hexadecimal self-defining term is written as the letter X, an apostrophe, a string of up to 6 hexadecimal digits, and a second apostrophe. X'123456', X'FACED', and X'001B7' are examples of hexadecimal self-defining terms. As above, the value of a hexadecimal self-defining term must be at most $2^{24}-1$ = X'FFFFFF'.

A binary self-defining term is written as the letter B, an apostrophe, a string of up to 24 binary digits, and a second apostrophe. B'110010', B'0001', and B'1111111100001100' are examples of binary self-defining terms, Because 24 bits are allotted for the value of self-defining terms, at most 24 digits may be specified between the apostrophes. Note also that the value of the term is assumed positive even though the leftmost position contains a one bit.

A character self-defining term is written as the letter C, an apostrophe, a string of up to three characters (except for two cases to be described momentarily), and a second apostrophe. Thus, C'A', C'...', and C'A B' are valid character self-defining terms. The third example, in which a blank appears, is the exception to the rule mentioned in Section 9 that the operand field is terminated by the first blank column after it starts: if the blank is part of a character string as in a character self-defining term, it doesn't count. The two unusual cases which arise in character strings concern the apostrophe and the ampersand. It is clear that if apostrophes *are* to be used to delimit the character string, some means must be found to get an apostrophe into the character string. (The ampersand has a special use in macro-instructions which will be treated later.) The technique used in the System/360 Assembler Language is to represent an apostrophe (or ampersand) in a character string by a pair of apostrophes (or ampersands) -- a character self-defining term containing a single apostrophe (or ampersand) would therefore be written C'''' (or C'&&'). This can lead to cryptic constructs such as C'''''''' and C'&&&&&&', but they are valid character self-defining terms.. The problem now arises as to how a value is associated with character self-defining terms; it is clear that this will depend on the internal representation assumed for characters. In System/360 the conventional representation is called the Extended Binary Coded Decimal Interchange Code, or EBCDXC, or even EBCD, for short. Each character is represented internally by a single byte -- two hexadecimal digits -- as indicated in Table III. Note that the characters $, #, and @ are considered to be letters in the Assembler Language. This will have bearing on the definition of symbols, which will be discussed shortly.

Character	Representation	Character	Representation	Character	Representation
blank	40	C	C3	T	E3
.	4B	D	C4	U	E4
(	4D	E	C5	V	E5
+	4E	F	C6	W	E6
&	50	G	C7	X	E7
$	5B	H	C8	Y	E8
*	5C	I	C9	Z	E9
)	5D	J	D1	0 (digit)	F0
-	60	K	D2	1	F1
/	61	L	D3	2	F2
,	6B	M	D4	3	F3
#	7B	N	D5	4	F4
@	7C	∅ (letter)	D6	5	F5
'	7D	P	D7	6	F6
=	7E	Q	D8	7	F7
A	C1	R	D9	8	F8
B	C2	S	E2	9	F9

Table III. EBCDIC Character Representation

Thus the value associated with the character self-defining term C' ' is the same as that of the hexadecimal self-defining term X'40', the binary self-defining term B'1000000', and the decimal self-defining term 64. Which type of term is chosen by the programmer is largely a matter of context; certain types will be more natural than others in some places. In practice, we will find that decimal self-defining terms are used so extensively that it is easy to forget that any other type of self-defining, term of the same value could be used as well.

In the previous section, Fig. 9.5 is an example of an instruction in which the operand field entry contains the decimal self-defining terms 7 and 3.

Symbols are a somewhat more intricate matter, even though their use will be seen later to be as. simple and natural as the use of self-defining terms. A symbol is a string of from one to eight letters or digits, the first of which must be a letter. (Remember that $,@, and # are "letters" to the Assembler.) No special characters are allowed (namely "(", ")", "+", "-", "*", "/", "=", "."♦","," ' ", "&", and " " (blank)). The following are all valid symbols.

A	AGENT007	A1B2C3D4
#235	ØO@H	APØPLEXY
JAMES	KSFØ	PRURIENT
$746295	WØNKA	ZYZYGY99

The following are not valid symbols, for the reasons given.

$7462.95	(decimal point not allowed)
BØND/007	(no division sign allowed)
SET GØ	(no blanks allowed)
235#	(does not start with a letter)
CHARACTER	(too many characters)
TEN*FIVE	(contains the special character *)
C'WØNKA'	(no apostrophes allowed)

Symbols have the following six attributes: value, relocatability, length, type, scaling, and integer. Of these, the first three will be our main concern, and the last three will be discussed later.

A-symbol acquires a value by virtue of its appearance as the name field entry in a statement of an appropriate type. The relocatability attribute depends on several factors, one of which will be mentioned shortly; we usually say simply that a symbol is <u>relocatable</u> or <u>absolute</u> (not relocatable). The length attribute of a symbol depends on the type of statement in whose name field the symbol appears. We will give a number of examples of the use of symbols in statements which are typical of actual programs. The reader should bear in mind that these are simply examples and that the instructions described here will be covered in detail later.

Symbols are mainly used as names of places in memory. In Fig. 9.5 the symbol LØAD is the name of the location at which the instruction (whose mnemonic is LR) begins. In the machine instruction statement

```
GETCØNST        L               0,4(2,7)
```

the symbol GETCØNST is the name of another machine instruction which loads a fullword from memory into general register 0. In the assembler instruction statement

```
TEN             DC              F'10'
```

TEN is a name for a fullword area in memory into which the assembler will place'the integer constant 10. In the macro-instruction statement

```
EXIT            RETURN          (14,12),T
```

the symbol EXIT is the name of the beginning of the macro-instruction. It is clear that no symbol can be given a value in a comment statement.

Two further questions will be discussed in this section: how do symbols get their values, and of what real use are they anyway? A partial answer to the second question is that their use greatly simplifies the programming task, and we will be in a position to appreciate this soon. To answer the first question, it is useful to examine briefly the pertinent part of the assembly process.

When a program is ready to be assembled, one of the first steps the Assembler must perform is the assignment of a <u>relative origin</u> (or starting location). In the discussion of job processing it was mentioned that at' the beginning of the execution step the user's program (in load module form) had to be loaded into memory. Now it will almost invariably be the case that the programmer has no a priori knowledge of where the Supervisor program will begin loading <u>his</u> program, and in fact the place where it begins may change each time the program is run. Thus, during the assembly 'step, the best that the programmer (and therefore the Assembler) can do is assign a relative origin for the program which will act as an <u>assumed</u> location for the beginning of the program. (The program must of course be written so that it will work correctly even if the assumed relative origin differs from the actual origin assigned by the Supervisor.)

Using this assumed origin as the initial value of the Location Counter (which we will abbreviate LC), the Assembler begins scanning the statements of the source program. As each statement is read, the assembler determines (a) whether a symbol appears in the name field, and (b) the length of the area in memory which will be occupied by the instruction. If there is a symbol, the value assigned to it will (except for one unusual case) be the value of the LC at that time. The LC is then incremented by the length just computed. For example, suppose the value of the LC was $7B6_{16}$ when the statement given in the first example above was scanned. Then the value of the symbol GETCØNST would be $7B6_{16}$, and because the instruction whose mnemonic is L is an RX-type instruction of length 4 bytes, the LC is incremented by four and will be $7BA_{16}$ when the scan of the following statement is begun. In this way the Assembler scans all the statements of the program and assigns values to all symbols appearing as name field entries. It should be noted that there are other methods for assigning values to symbols, but the method described is what will most often be used, and that there are also assembler instruction statements which allow the programmer to change the value of the Location Counter. This usual method of symbol definition provides the simplest definition of a relocatable symbol: suppose the relative origin is changed by some fixed amount; if the value of the symbol changes by the same amount, then that symbol is relocatable. We will see later that it is also possible to define symbols whose values either do not change or which change in different ways. (The reader should also note that there is a definite difference between the LC, which is maintained by the Assembler program in the course of processing the statements of the source program, and the Instruction Address in the PSW, which gives the location in memory of the next instruction to be executed during the execution step of the program. They are not at all the same.)

After this brief discussion of how symbols get their values, we turn to the question of their utility. Suppose we want to write an instruction which will load the integer constant ten into RO (remember that this is an abbreviation for general register 0). Suppose also that we also know that some other general register will contain an address which will provide addressability for the fullword area of memory containing the constant. Then we could calculate what the exact displacement would have to be and write the instruction with the base and displacement given explicitly. If, for example, these were 6 and $4EC_{16}$ respectively, we could write (the details of writing the operand field will be discussed in the next section)

```
          L           0,X'4EC'(0,6)
```

If, however, the fullword area containing the constant were given the name TEN (as in the example earlier), we could write instead

```
          L           0,TEN
```

and let the Assembler figure out what base and displacement to use. To do this the Assembler needs only to be informed of the address it should assume will be in register 6 (the method will be discussed in Section 12), and the calculation of the displacement will be done for us. It may seem that this is a relatively small return for so much effort; it can be seen, however, that if the program 'is modified slightly so that the constant no longer lies in exactly the same position relative to the assumed given base address, then all instructions which refer to the constant must have their displacements recalculated. (It is of course implicit in this discussion that (a) no program works just the way we want it to on the first try, and (b) even if it did we'd think of some changes to make before we got done with it. If this were not so we could dispense with assemblers and be content with producing programs consisting of strings of hexadecimal digits -- but even those who programmed the earliest machines that way are agreed that assembly languages are an improvement.) Thus the main function of the Assembler will be to provide a convenientmeans for writing and modifying a given program and getting it to execute correctly, by performing many of the details of the programming process for us.

11. INSTRUCTIONS (II), MNEMONICS AND OPERANDS

In this section we will consider some of the problems of writing actual machine instructions, using a number of instruction formats and giving some simple example6 of actual code sequences. The use and detail6 of the functioning of the individual instructions-will be the subject of many later discussions, so no effort should be made to memorize the mnemonics, operation codes, or descriptions of any of the instructions at this point.

Mnemonics provide a short abbreviation for a descriptive word or phrase which designates the action of each operation code. They may range from something as simple as "A" meaning "Add", to "BXLE" meaning "Branch *on* Index Low or Equal". To simplify the presentation, we will discuss each class of instructions separately, and sometimes give examples of how they are written. A number of abbreviations such as r_1, s_2, I, etc. will be explained as we go along.

RR Instructions

Instructions of RR format are given in Table IV; several things should be noted about the instructions listed there. First, not all of the available digit combinations between 00_{16} and $3F_{16}$ (in the column6 labeled "Opcode") are used as actual operation-codes. Second, all of the instruction6 in the second column refer to the floating-point registers, the uses of which will be described in detail later. (The floating-point instructions operate on data in a format which is interpreted differently from the integer representations discussed in Section 6.) Third, two of the instructions (namely SSK and ISK) are not normally available to the programmer and their descriptions will therefore be deferred (they are called privileged operations).

Opcode (hex)	Mnemonic	Instruction	Opcode (hex)	Mnemonic	Instruction
04	SPM	Set Program Mask	20	LPDR	Load Positive
05	BALR	Branch and Link	21	LNDR	Load Negative
06	BCTR	Branch on Count	22	LTDR	Load and Test
07	BCR	Branch on Condition	23	LCDR	Load Complement
08	SSK	Set Storage Key	24	HDR	Halve
09	ISK	Insert Storage Key	28	LDR	Load
0A	SVC	Supervisor Call	29	CDR	Compare
			2A	ADR	Add Normalized
10	LPR	Load Positive	2B	SDR	Subtract Normalized
11	LNR	Load Negative	2c	MDR	Multiply
12	LTR	Load and Test	2D	DDR	Divide
13	LCR	Load Complement	2E	AWR	Add Unnormalized
14	NR	Logical AND	2F	SWR	Subtract Unnormalized
15	CLR	Compare Logical	30	LPER	Load Positive
16	ØR	Logical ØR	31	LNER	Load Negative
17	XR	Exclusive ØR	32	LTER	Load and Test
18	LR	Load	33	LCER	Load Complement
19	CR	Compare	34	HER	Halve
1A	AR	Add	38	LER	Load
1B	SR	Subtract	39	CER	Compare
1C	MR	Multiply	3A	AER	Add Normalized
1D	DR	Divide	3B	SER	Subtract Normalized
1E	ALR	Add Logical	3C	MER	Multiply
1F	SLR	Subtract Logical	3D	DER	Divide
			3E	AUR	Add Unnormalized
			3F	SUR	Subtract Unnormalized

TABLE IV.

RR Instructions

For all but two of the RR instructions, the two operands of the operand field entry in a machine instruction statement must be written in the form

$$r_1,r_2$$

where the operands r_1 and r_2 will be described shortly. The exceptions, which have only a single operand in the operand field entry, are SPM (in which case the operand is written in the form r_1) and SVC (in which case it is written in the form I).

To explain the meaning of the notation "r_1,r_2", it is perhaps useful to refer to the example of a machine instruction statement in Fig. 9.5, in which the operation and operand fields were "LR 7,3". (It was noted in the description of the figure that execution of this instruction would cause the contents of R7 to be replaced by the contents of R3.) In this case, "r_1" is "7" and "r_2" is "3". In fact, the quantities r_1 and r_2 must simply be absolute (i.e., non-relocatable) <u>expressions</u> of value less than 16; a more formal definition of the term "expression" will be given shortly. Thus, we could just as well have written LR X'7',B'11' in this example. For RR instructions, the values of the expressions in the operand field are placed by the Assembler into two adjacent hexadecimal digits, called operand register specification digits, in the second byte of the instruction (which was labeled "Register Specification" in the first diagram of Fig. 4.2), as in the following figure.

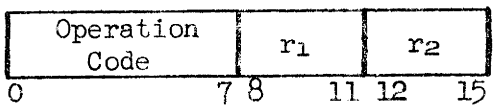


Figure 11.1 RR Instruction Showing Register Specification Digits

The subscripts on the quantities "r_1" and "r_2" are simply a way to distinguish which operand is being referred to; in general we will find that using the terms "first operand", '*second operand", etc. in a consistent manner will help in remembering what actions are being performed by each instruction. We would therefore say for most of the RR instructions that the operand r_1 specifies the register containing the "first operand". It will become apparent that the word "operand" is used here in two different senses: as part of the operand field entry of some instruction statement, an operand is an expression which will eventually be-translated by the Assembler into some part of an instruction; we also call an operand one of the quantities in a register or in memory which at execution time participates in the given operation. The difference is not terribly important but can be confusing, and which is meant will normally be clear from context. Thus the operands (first meaning) in the operand field entry of the instruction LR 7,3 are 7 and 3, whereas at execution time the operands (second meaning) of the

LR instruction will be found in general registers 7 and 3. Using Table IV to find that the operation code corresponding to the mnemonic LR is 18_{16}, the two-byte instructionwhich would be assembled from the statement as given would be $\boxed{18}\boxed{73}$ in hexadecimal.

For the case of the SPM instruction the digit labeled r_2 in Fig. 11.1 is ignored when the instruction is decoded; and for the SVC instruction, the entire second byte of the instruction is occupied by an 8-bit number which is specified by the absolute expression "I", as indicated above, Thus SPM 14 and svc 255 are acceptable forms of each instruction, in which decimal self-defining terms are used for the operand field entries.

Before discussing RX format instructions, we will discuss in more detail the complexities of what is meant by an "expression". Since most of the material of the next several pages will be illustrated in fairly simple examples to be given later, it is not important that some of these conventions of Assembler Language remain unclear for now.

An expression is an arithmetic combination of <u>terms</u> (and we will also give a definition of the term "term") which can be evaluated by the Assembler to produce a meaningful value for the operand. Mathematical operators allowed include +, -, *, and /, indicating addition, subtraction, multiplication, and division respectively; the rules used in performing these operations are described below. The quantities used as the basic elements of an expression are terms, which can be one of the five following items:

a self-defining term (absolute);
a symbol (absolute or relocatable);
a Location Counter Reference (relocatable);
a literal (relocatable);
a Symbol Length Attribute Reference (absolute).

Each of the latter three will be described later. An expression using a symbol and a self-defining term is GETCØNST+X'4A' and an expression using only self-defining terms is X'12'+C'.'-B'1010001'+7 which the reader -can verify to have the value 19_{10}.

To illustrate the definition of an absolute symbol (up to now we have illustrated only the use of relocatable symbols), we will make brief mention of the EQU assembler instruction: the assembler instruction statement "symbol EQU expression" gives to the symbol in the name field the attributes (including value and relocatability) of the expressionin the operand field. Thus the statement

```
ABS425    EQU    425
```

serves to define an absolute symbol with value 425_{10}. (This is the unusual case mentioned in Section 10 where the value of the symbol is not the value of the LC when the symbol was encountered.)

Parentheses in an expression may be used, as in ordinary mathematical use (and as in algebraic procedural languages such as FORTRAN, ALGOL, and PL/1) to indicate groupings. As one might suspect, an expression may not contain two operators in succession; a less familiar restriction is that an expression may not begin with an operator, sc that -5+ABS425 is invalid, whereas 0-5+ABS425 is correct. (The maximum number of terms allowed and the maximum level of nesting of parentheses in an expression both depend on the size and sophistication of the Assembler; we will simply mention an upper limit of 16 and 5 respectively, corresponding to the OS/360 Assembler.)

Expressions

With these notational matters more or less in hand, we can now state the rules for evaluation of expressions.

1. Each term is evaluated to fullword accuracy, namely 32 bits. The relocatability attribute of each term is noted.

2. Parenthesized subexpressions are evaluated first, and the resulting value used in computing the value of the rest of the expression. Thus in the expression (X'100'+2*(ABS425-420))+1 (where ABS425 is assumed to have been defined as above), the value of (ABS425-420) would be evaluated first.

3. As is the case in procedurallanguages, multiplications and divisions are done before additions and subtractions. Thus the value of the expression just given would be evaluated as (X'100'+(2*(5)))+1 and not ((X'100'+2)*(5))+1. Note that relocatable terms or subexpressions may not occur in multiply or divide operations.

4. Operations are performed in left-to-right order. Thus 5*2/4 means (5*2)/4, not 5*(2/4).

5. Multiplications yield a 32-bit result which is the low-order half of the double-length product; thus significant bits can be lost if the product is too large.

6. Division always yields an integer result; remainders are discarded. Thus 5*2/4 has the value 2, and 5*(2/4) has the value 0. Division by zero is permitted, with the result simply being set to zero.

7. Negative quantities are carried in standard two's complement representation.

8. When the expression has been completely evaluated, it is truncated to the value contained in its rightmost 24 bits, which is then considered (as was noted for self-defining terms) to have a positive value, even though the bits dropped off may have all been ones.

9. The relocatability attribute of the result is found as follows: if there is an even number of relocatable terms appearing in the expression in such a way that they are <u>paired</u> (that is, they appear with opposite signs) so that a change in the relative origin assigned to the program has no effect on the value of the expression, then the expression is absolute. If there is one remaining unpaired term not directly preceded by a minus sign, then the expression is relocatable and has the relocatability attribute of the unpaired term. (Numerous examples will be given later, so don't worry if this seems obscure at present.)

After this somewhat lengthy digression, we return to the problems of writing actual machine instructions by noting that the machine instruction example at the beginning of the chapter could have been written

```
LØAD      LR     C'45'-(7*X'2A36')+ABS425*B'11111'-235,18/(Q-Q)+3
```

though the gain in clarity is not obvious. A somewhat more reasonable usage might be as illustrated in the following sequence of statements.

```
R7     EQU   7
R3     EQU   3
LØAD   LR    R7, R3
```

Note that there is a difference between (1) the notational convenience "R7" (meaning general register 7) introduced in Section 3, (2) the definition of an absolute symbol R7 to have the value 7, and (3) the use of the symbol as an operand in the operand field entry of a machine instruction where the use of register 7 is indicated. The above example is entirely equivalent to the two below.

```
ZØRCH  EQU   3                      R7    EQU   3
ZILCH  EQU   7                      R3    EQU   7
L#AD   LR    ZILCH,ZØRCH            LØAD  LR    R3,R7
```

Just to show that programming with RR instructions is in fact quite simple, suppose that at some point in a program we wish to add the contents of R2 to R14, subtract the contents of R9 from the sum, and leave the result in R0; the following three statements (whose properties will be discussed later) would suffice:

```
LR  0,2   MØVE CØNTENTS ØF R2 TØ R0
AR  0,14  ADD CØNTENTS ØF R14
SR  0,9   SUBTRACT CØNTENTS ØF R9
```

RX Instructions

RX instructions are given in Table V. As was the case in Table IV, not all of the available digit combinations are used as actual operation codes; and all of the instructions in the right-hand column again refer to operations on the floating-point registers and will be discussed later. None of the RX instructions is privileged, and the format of the operand field entry is the same for each. It should be kept in mind that RX instructions always refer to memory in some way. Referring to Fig. 11.2, we see that four quantities are to be specified -- the operand register specification digit r_1, the index register specification digit x_2, the base register specification digit b_2, and the displacement d_2. (We are again entering on a fairly technical discussion, the details of which need not be assimilated at this point, since many later examples will be given in illustration of the various possibilities.)

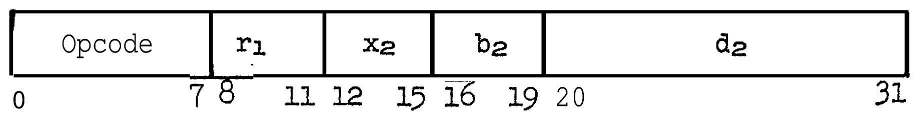


Figure 11.2 RX Instruction Showing Register Specification Digits

Opcode (hex)	Mnemonic	Instruction	Opcode (hex)	Mnemonic	Instruction
40	STH	Store	60	STD	Store
41	LA	Load Address	68	LD	Load
42	STC	Store Character	69	CD	Compare
43	IC	Insert Character	6A	AD	Add
44	EX	Execute	6B	SD	Subtract
45	BAL	Branch and Link	6C	MD	Multiply
46	BCT	Branch on Count	6D	DD	Divide
47	BC	Branch on Condition	6E	AW	Add Unnormalized
48	LH	Load	6F	SW	Subtract Unnormalized
49	CH	Compare			
4A	AH	Add	70	STE	Store
4B	SK	Subtract	78	LE	Load
4C	MH	Multiply	79	CE	Compare
4E	CVD	Convert to Decimal	7A	AE	Add
4F	CVB	Convert to Binary	7B	SE	Subtract
50	ST	Store	7C	ME	Multiply
54	N	Logical AND	7D	DE	Divide
55	CL	Compare Logical	7E	AU	Add Unnormalized
56	ØR	Logical ØR	7F	su	Subtract Unnormalized
57	X	Exclusive ØR			
58	L	Load			
59	C	Compare			
5A	A	Add			
5B	S	Subtract			
5C	M	Multiply			
5D	D	Divide			
5E	AL	Add Logical			
5F	SL	Subtract Logical			

TABLE V.

RX Instructions

There is quite a variety of ways in which the operand field entry of an RX-type machine instruction statement may be written, but they all eventually must yield values for the four needed quantities. Rather than give all the forms for the operand field entry immediately, we note first that it is of the general form

r_1,<address specification>

where <address specification> will be discussed shortly. The operand register specification digit r_1 is formed according to the same rules given above for the r_1 and r_2 digits of RR instructions: it must be an absolute expression of value less than 16.

Suppose first that we wish to specify explicitly the values assigned to x_2, b_2, and d_2: this is done by writing the second operand (namely <address specification>) as

$d_2(x_2,b_2)$

For example, the instructions in examples 3, 4, and 5 of Section 5 (page 5-3) could be written (giving both the assembled form and the operation and operand field entries of the machine instruction.statement) as in Fig. 11.3.

$\boxed{43}\boxed{0A}\boxed{7}\boxed{468}$	IC	0,X'468'(10,7)
$\boxed{43}\boxed{00}\boxed{7}\boxed{468}$	IC	0,1128(0,7)
$\boxed{43}\boxed{07}\boxed{0}\boxed{468}$	IC	0,1128(7,0)

Figure 11.3 RX Instruction with Explicit Operands

In the last of these three examples, we could have written the second operand as 1128(7) and the Assembler will give the omitted item (the base register specification digit b_2) the value zero.

As was mentioned in the discussion-of addressing in Section 5, the use of the index register specification digit x_2 when the base register specification digit b_2 was intended can lead to programs which function more slowly, though correctly. By specifying only the base digit when no indexing is intended, the program is both more efficient and more easily understood -- the second of the above examples, where we could have written 1128(,7) also, is therefore preferable to the third.

The utility of the Assembler **becomes more apparent** when we consider all the forma in which the **second** operand of an RX instruction may be written; these **are given** in **Fig.** 11.4 below.

Explicit Address	Implied Address
$d_2(x_2,b_2)$	$s_2(x_2)$
$d_2(x_2)$	
$d_2(,b_2)$	s_2

Figure **11.4 Address** Specification in **RX-Type Instructions**

In the three **cases where** an explicit address **is** desired, **each of** the **quantities** d_2, x_2, and b_2 (where **specified**) must be an absolute **expression;** x_2 and b_2, like r_1, **must** have value less than 16, and d_2 must have **value** lees than or equal to $4095_{10} = FFF_{16}$. Rote that the second and third forms **of** explicit **address** implicitly specify $b_2 = 0$ and $x_2 = 0$, respective&y, **as indicated previously.**

In the two cases where an implied address is desired, the quantity s_2 may be either an absolute or a relocatable **expression** of value **less** than 2^{24}. **This** means that we may write **instructions** such **as** L **0,ANSWER** and leave it to the Assembler to **compute** the proper base and dieplacement; **how this** is **done** will be **discussed** in the next **section.** For the **moment suppose that** the **Assembler** has **sufficient** information **so that** the instruction **IC 0,BYTE** **is** translated into $\boxed{43}\boxed{00}\boxed{7}\boxed{468}$ as in Fig. **11.3. Then** if the index **register** to be used is **R10,** the **instruction** IC **0,BYTE(10)** would be translated into $\boxed{43}\boxed{0A}\boxed{7}\boxed{468}$.

This is the- **same** instruction used in example 3 in section 5; **the** example given there **was simply** meant to illustrate an address **calculation** at execution time rather than (as above) the method **used** by **the Assembler** to specify the base and index digits. We will find that the moet **common** 'means **of** address specification in simple **programs** is through the **use** of **implied addresses,** where the Assembler computes the proper displacement for **us.**

To give a simple example of a sequence of **statements** which increment by one the **fullword** integer stored in memory in an addressable **area** named by the symbol N, we could use the following:

```
L   O,N    LØAD FRØM N INTØ RO
A   O,ØNE  ADD INTEGER CØNSTANT 1
ST  O,N    STØRE RESULT BACK AT N
```

where it is assumed that an addressable fullword area named ØNE which contains the integer constant +1 has been defined in the program. We will see later that there are several ways to define such constants.

RS and SI Instructions

The RS-type and SI-type instruction6 listed in Table VI are somewhat varied both in application and in the ways in which the operand fields are specified. Note that there are nine privileged instructions: SSM, LPSW, WRD, RDD, SIØ, TIØ, HIØ, TCH, and "Diagnose", for which there is no mnemonic.

Opcode (hex)	Mnemonic	Instruction	Cpcode (hex)	Mnemonic	Instruction
80	SSM	Set System Mask	90	STM	Store Multiple
82	LPSW	Load PSW	91	TM	Test Under Mask
83		Diagnose	92	MVI	Mwe
84	WRD	Write Direct	93	TS	Test and Set
85	RDD	Read Direct	94	NI	Logical AND
86	BXH	Branch on Index High	95	CLI	Compare Logical
87	BXLE	Branch on Index Low or Equal	96	ØI	Logical ØR
			97	XI	Exclusive ØR
88	SRL	Shift Right SL	98	LM	Load Multiple
89	SLL	Shift Left SL			
8A	SRA	Shift Right S	9C	SIØ	start I/Ø
8B	SLA	Shift Left S	9D	TIØ	Test I/Ø
8c	SRDL	Shift Right DL	9E	HIØ	Halt I/Ø
8D	SLDL	Shift Left DL	9F	TCH	Test Channel
8E	SRDA	Shift Right D			
8F	SLDA	Shift Left D			

TABLE VI.

RS and SI Instructions
(For Shift Instructions, S = Single, L = Logical, D = Double)

Since the operand fields of RS and SI instructions cannot be described in as uniform a way as was possible for RX instructions, the details will be left to the discussion of the individual instructions. A simple example of an SI instruction is MVI FLAG,0 which would cause the byte named FLAG (which is assumed to be addressable) to be set to zero.

SS Instruction6

The instructions of SS type are given in Table VII. There are no privileged SS instructions. As was the case for the RS and SI instructions, discussion of the operand field formats will be deferred. The last six instructions in the right-hand column are decimal instructions, which operate

Opcode (hex)	Mnemonic	Instruction	Opcode (hex)	Mnemonic	Instruction
D1	MVN	Move Numeric	F1	MVØ	Move with Offset
D2	MVC	Move	F2	PACK	Pack
D3	MVZ	Move Zone	F3	UNPK	Unpack
D4	NC	Logical AND			
D5	CLC	Compare Logical	F8	ZAP	Zero and Add
D6	ØC	Logical ØR	F9	CP	Compare
D7	XC	Exclusive ØR	FA	AP	Add
DC	TR	Translate	FB	SP	Subtract
DD	TRT	Translate and Test	FC	MP	Multiply
DE	ED	Edit	FD	DP	Divide
DF	EDMK	Edit and Mark			

TABLE VII,
SS Instructions

on data which is stored in a different format (called packed decimal) from that described earlier for fixed-point integers in two's complement representation; decimal instructions will be treated later. An example of an SS instruction which would cause five bytes to be moved from a memory area named AREA to an area whose first byte is named FIELD is
MVC FIELD(5),AREA .

To conclude this short presentation of the instruction repertoire of System/360, a summary is given in the figure below of some of the overall characteristics of the instructions as they depend on the first four bits of the operation code. As was illustrated in Section 4, the first two bits determine the type and length of the instruction. The second pair of bits determines (depending on the instruction type) the operand length or the general functions performed by the instructions.

First Bit Pair	Second Bit Pair 00	01	10	11
00 (RR)	Branching and Status Switching	Fullword Fixed-Point and Logical	Floating-Point Long	Floating-Point Short
01 (RX)	Halfword Fixed-Point and Branching	Fullword Fixed-Point and Logical	Floating-Point Long	Floating-Point Short
10 (RS, SI)	Branching, Status Switching, and Shifting	Fixed-Point, Logical, and Input/Output		
11 (SS)		Logical		Decimal

Figure 11.5 General Instruction Classification

A closer examination of a complete table of operation codes reveals a great deal of symmetry in the specification of the codes used for similar functions. For example, the four instructions which perform the Logical AND operation (namely, NR, N, NI, and NC) all have operation codes in which the second hex digit is 4 and the first hex digits differ by multiples of 4 (namely, 14, 54, 94, and D4). Since we will make reference to instructions almost entirely by use of mnemonics, these details are only of passing interest for our purposes. The reader who is interested in a broader discussion of these topics -- collectively known as system architecture -- should consult the IBM Systems Journal, Vol. 3, Nos. 2 and 3, and the IBM Journal of Research and Development, vol. 8, No. 2.

12. ESTABLISHING AND MAINTAINING ADDRESSABILITY

In this section we will give an exposition of some simple methods for providing addressability for a program, and how the Assembler makes use of some programmer-provided information to calculate displacements. Rather than give a set of rules and show how they work, we will start with what we want and work backwards to some techniques which can be used to get it.

One particular instruction is central to the discussion, namely BALR. For the time being we will be interested -only in the situation where we write BALR r_1,0 (so that the second operand register specification digit r_2 is zero). The effect of this instruction when executed is to replace the contents of general register r_1 by the rightmost 32 bits of the PSW: the ILC, CC, and Program Mask occupy the leftmost byte of the register, and the rightmost 24 bits contain the value of the IA (which will be the address of the instruction following the BALR, because the IA is incremented by the instruction length (2 for BALR) during the Fetch portion of the instruction cycle). This is one solution to the problem posed at the end of Section 5, where addressability was first discussed; the BALR instruction gives us a way to find out where in memory a program is located.

Suppose that the following short sequence of statements is part of a program which is in memory and ready to be executed, and assume for the moment the Supervisor has relocated the program so that the first instruction (the BALR) happens to be at memory location 5000_{16}.

```
Location      Name      Operation     Operand

5000                    BALR          6,0
5002          BEGIN     L             2,N    LØAD CØNTENTS ØF N INTØ R2
5006                    A             2,ØNE ADD CØNTENTS ØF ØNE
500A                    ST            2,N    STØRE CØNTENTS ØF R2 INTØ N
 --- twenty-two additional bytes of instructions, data, etc. ---
5024          N         DC            F'8'   FULLWØRD INTEGER 8
5028          ØNE       DC            F'S    FULLWØRD INTEGER1
```

Figure 12.1 A Simple Program Segment

Some explanation of the items in the example may be helpful. The instructions L, A, and ST respectively (1) put the contents of a fullword from memory into a general register (i.e., Load the register), (2) Add the contents of a fullword area in memory to the contents of a register, and (3) replace the contents of a fullword area in memory with the contents of a general register (i.e., STore the register). The DC statements, which are treated in the next section, are meant simply to provide two fullword areas of memory with names "N" and "ØNE" which contain the fullword integer values desired; we have arbitrarily set the contents of the fullword at N to the integer 8 even though in an actual program any value might be possible. All of these instructions will be covered in detail later.

When the program has begun and after-the BALR has been executed, R6 will contain xx005002$_{16}$, where xx stands for two hex digits whose values are of no concern at the moment. To determine the proper displacement for the L instruction at 5002$_{16}$, we can use the known contents of R6 (since the xx digits are ignored in address computations) to compute a displacement of 5024$_{16}$ - 5002$_{16}$ = 022$_{16}$; then the assembled machine instruction (using the operation code 58 for the mnemonic L) should be $\boxed{58}\boxed{20}\boxed{6}\boxed{022}$. Then when the instruction is executed, the computation of the effective address yields 022 + 005002 = 005024, which is what we want. If we continued in this fashion for the rest of the instructions, we would find that the following "assembled" quantities in the indicated locations would give the desired results.

Location	Assembled Contents	Original Statement		
5000	0560		BALR	6,0
5002	58206022	BEGIN	L	2,N
5006	5A206026		A	2,ØNE
500A	50206022		ST	2,N
5024	00000008	N	DC	F'8'
5028	00000001	ØNE	DC	F'1'

Figure 12.2 Simple Program Segment with Assembled Contents

So far, so good: we have constructed a sequence of statements which will give a desired result if it is placed in memory at the right place. It is natural to ask at this point what would happen if the program had been put elsewhere by the Supervisor. So, assume that the same program segment begins at $84E8_{16}$, as in the figure below.

```
Location      Statement

84E8                BALR 6,0
84EA          BEGIN L     2,N
84EE                A     2,ØNE
84F2                ST    2,N
   --- the same 22 bytes of odds and ends ---
850C          N     DC    F'8'
8510          ØNE   DC    F'1'
```

Figure 12.3 Same Program Segment, Different Memory Location

Now, the contents of R6 after the BALR is executed would be $xx0084EA_{16}$. To access the contents of the fullword at N, using R6 as a base register, the necessary displacement is $850C - 84EA = 022_{16}$ (as before!) and the displacement necessary in the A instruction is $8510 - 84EA = 026_{16}$. Thus the assembled program would appear as in the figure below.

```
Location      Assembled Contents

84E8             0560
84EA             58206022
84EE             5A206026
84F2             50206022
   ----------------
850C             00000008
8510             00000001
```

Figure 12.4 Same Program Segment with Assembled Contents

The identical assembled program would be used in each case to perform the desired calculation. It therefore appears that so long as the same fixed relationship is maintained between the various parts of -the program segment (namely that there be 22 bytes between the ST instruction and the fullword named N, and that N and ØNE name areas that fall on fullword boundaries, the segment could be placed anywhere in memory and still execute correctly.

This is because the displacements of the three RX-type instructions were calculated on the assumption that at the time the program is executed there would be an address in R6 (namely the address of the L instruction named BEGIN) which could be used for a base address. Indeed, we could have assumed that the program began at momory location zero (even though an actual program would not be placed there) because the contents of R6 after the BALR would then be xx000002 and the displacements would be calculated exactly as before. In the first example, the actual origin of the program segment was 5000_{16}; we could by chance have assigned that value as a relative origin in the program and had the values of the Assembler's Location Counter correspond identically to the actual locations later assigned by the Supervisor to each instruction. In that case, we would need to inform the Assembler that the quantity to be used as a base is 5002_{16}, and that it would be found in R6 at execution time. Similarly, in the second example, the relative origin would be $84E8_{16}$, and the contents of R6 that the Assembler should assume in order to calculate the correct displacements would be $84EA_{16}$. If the value of the actual origin is assigned to the relative origin by the programmer, and if the Assembler knows that the contents of R6 at execution time will also be the value of the symbol BEGIN , then the correct displacements will be found.' However, in each of the above examples, the computation of the displacements actually depended not on a knowledge of the actual locations of the instructions at execution time, but only on their locations relative to one another and on the value assumed to be available for addressing purposes. Thus, the technique used is to assign a relative origin for the program, and then to give some value relative to that relative origin which may be used for computing displacements; although this seems complicated, we will find it quite simple in practice.

The assembler instruction which provides this information is the USING instruction. It is written

USING s,r_1

where s is a relocatable or absolute expression (usually just a symbol. will be used) whose magnitude is less than 2^{24}, and r_1 is an absolute expression of value less than 16 which specifies the register to be used as a base. (As usual, there is more to using USING than has been stated here, but we will use this simplified explanation for the time being.) Thus, the statement USING BEGIN,6 would inform the Assembler that register 6 may be assumed (for purposes of computing displacements) to be a base register which will contain the value of the symbol BEGIN . We could rewrite the sample program segment to include the USING statement as in the figure below.

```
                    BALR      6,0
                    USING     BEGIN,6
          BEGIN     L         2,N
                    A         2,ØNE
                    ST        2,N
             ----------------------
          N         DC        F'8'
          ØNE       DC        F'1'
```

Figure 12.5 Program Segment with USING Instruction

If the relative origin assigned by the programmer is zero, the value of the symbol BEGIN is 2, and the values of the symbols N and ØNE are 24_{16} and 28_{16} respectively. To complete the addressing syllable of the ST instruction, the Assembler need only note tha-t the difference between the value of the symbol N and the value that the USING instruction specifies will be present in R6, is $24 - 2 = 22_{16}$; this is the required displacement. It should be noted at this point that the value provided by the USING statement must allow the Assembler to compute a legal displacement. If the calculation yields a negative value or one greater than 4095, the location referred to by the-symbol in question is still not addressable, and further steps would have to be taken.

Two important features of the program segment in Figure 12.5 should be noted. First, the USING instruction does absolutely nothing about actually loading a value into a register; it merely tells the Assembler what to assume will be there when the program is executed. Second, if the BALR instruction had been omitted, there is no guarantee when the program is executed that the correct effective addresses will be computed. The example below will help to illustrate this.

Suppose an error had been made in punching the 'card with the L instruction, such that it appeared

```
BEGIN  L    6,N    LØAD CØNTENTS ØF N INTØ R2
```

(the first operand was incorrectly punched as 6 instead of 2). The assembled program would then appear as in Figure 12.6, assuming a relative origin of 0 had been assigned to the BALR instruction.

```
Location      Assembled  Contents      Statement
   0          0560                               BALR    6,0
                                                 USING   BEGIN,6
   2          58606022                 BEGIN     L       6,N
   6          5A206026                           A       2,ØNE
   A          50206022                           ST      2,N
---------------------------------------------------------
  24          00000008                 N         DC      F'8'
  28          00000001                 ØNE       DC      F'S
```

Figure 12.6 Sample Program Segment with Erroneous Statement

It is apparent that thfs program will assemble correctly, as did the one in Figure 12.5, since all quantities are properly specified. However, at <u>execution time</u>, things go rapidly awry. Suppose again that the actual location assigned by the Supervisor to the BALR is 5000_{16}, so that when the L instruction is executed, R6 contains $xx005002_{16}$. Now, the L instruction transmits a fullword from the memory location at the effective address given by the second operand into the register specified by the first operand, 'which in this case is R6. When the effective address of N is being calculated, R6 will contain the correct base address; but when the execution of the L instruction is complete, the contents of R6 will have become 00000008_{16}, and <u>not</u> xx005002. When the <u>next</u> instruction is executed, the effective address calculated is $26_{16} + 8_{16} = 00002E_{16}$ and not 5028_{16}, which is where the desired operand is to be found. In this case, the generated effective address is not divisible by 4, so that it refers to the incorrect byte of the required fullword operand; hence a specification exception occurs, and remedial action can be initiated immediately. This does not by any means imply that at any time we have the misfortune to destroy the contents of a base register that the CPU will be able to detect the error. Indeed, if the contents of the fullword at N had been the integer 2 instead of 8, then the effective address would have been computed to be 2 + 26 = 28_{16}, which is a perfectly acceptable address for a fullword. The subsequent instructions would thus have gone their way, adding the contents of the fullword at memory location 28_{16} to R2, and storing the result at location 24_{16}, which is obviously not what is intended. It is partly a matter of chance as to' how much further damage such a program error can cause when the program is executed; indeed, when the CPU finally (if it ever) detects an error, all evidence pointing to the offending instruction may have been lost (R6 may have been changed several times!), making error tracing difficult. Thus the programmer must take care to insure the integrity of the contents of registers being used for base registers, -since the Assembler makes no checks for instructions performing operations on registers.designated in USING instructions as base registers. This warning should not be taken lightly; the errors caused by mishandling base registers are among the most destructive of program continuity and the most difficult to find.

There is one further method in common use for establishing addressability, which is simply to require that when "control" reaches a certain point in the program (where a specified instruction is about to be executed), an agreed-upon address be in an agreed-upon register. Thus if the program segment used in the above examples were part of a larger program, we could then require that at any time that control reaches the statement named BEGIN, the actual address of that instruction must be in R6. Then the BALR could be omitted, and the USING instruction would specify that R6 may still be assumed to contain the correct value. The problem of how one part of a program knows where the others are, so that it can pre-load the correct address into the agreed-upon register, will be discussed later; the solutions to this problem are basic to the use of subroutines, which is an important programming topic.

In many of the following sections we will have occasion to examine short segments of coding which illustrate the use of various instructions. Rather than indicate explicitly the assignment of a base register and its contents, we will assume that each segment is part of a larger program in which addressability has been taken care of. We will also assume that all symbols used have been defined and are addressable, and that the base register is different from any registers used or changed in the example

13. CONSTANTS, STORAGE AREAS, AND LITERALS

In several places in the preceding sections we have made occasional use of the DC assembler instruction to indicate that a constant was to be constructed and placed in the program by the Assembler (DC is a mnemonic for "Define Constant"). In this section we will elaborate on the definition of constants and describe a technique which simplifies their use.

As indicated in some of the examples given previously, the DC instruction may have name, operation, operand, and comment field entries, of which the operation and operand field entries are mandatory. Since the comment field entry is optional, its use will be ignored in the following discussion. Rather than give all the rules for defining constants immediately, it is perhaps simpler to examine a few simple cases which illustrate the principles involved.

The statement DC F'8' defines (as stated in a number of earlier examples) a fullword integer constant of value 8_{10} placed on a fullword boundary. That is, four items have been specified:

(1) the value of the constant (in this case $+8_{10}$)

(2) the type of internal representation to be used for the given value (in this case two's complement integer);

(3) the length of the-constant (in this case four bytes); and

(4) the alignment in memory of the constant (in this case on a fullword boundary).

Because the Assembler does no placing of data in memory, it is probably difficult to see at present how a given sequence of four bytes can be placed, after processing by the Assembler, Linkage Editor, and Resident Supervisor, on proper boundaries. We will see that there are a few simple conventions which make this easy to accomplish. Some other types of conversion we will discuss here, and the letters which specify/the types/are Character (C), Binary (B), Hexadecimal (X), Halfword Integer (H), and Address Constant (A). The first three of these were encountered in the treatment of self-defining terms, and their use in the DC instruction is quite similar.

For the larger System/360 Assemblers, the operand field entry may consist of a number of operands which. are separated by commas; however, for most of the cases which will be of interest, a single operand will suffice. There are four parts to an operand: (1) a duplication factor, (2) a letter specifying the type of representation, (3) modifiers, and (4) the value of the constant or constants. Of these only the second (type) and fourth (value) are required, as in the example above where, F'8' was specified. The duplication factor is a relatively simple concept which will be treated shortly. There are three types of modifier, namely length, scale, and exponent, of which only length will be treated here. Because there is an important relationship between boundary alignment and the' use of a length modifier, we will discuss the techniques tied to obtain the proper alignment of constants and data.

When the relative origin is specified by the programmer at the start of his program, the Assembler checks whether the value given is exactly divisible by eight; if not, it is 'rounded up" to the, next larger multiple of eight, which is then used as the relative origin of his program. Thus the Assembler insures that the program begins with the most restrictive possible boundary alignment. Then if a constant is defined which must fall on some particular kind of word boundary, the Assembler need insure only that its Location Counter be divisible by the proper power of two (that Is, by 2, 4, or 8) at the locatlon of the leftmost byte of the constant. The Linkage Editor and Resident Supervisor 'must then respect this assumed alignment for the beginning of the program; this ensures that data and Instructions will fall on the proper boundaries when the program is finally loaded into memory for execution. We will of course assume that this is exactly what happens in the rest of our discussion; some of the implications of this method of handling programs will be treated in later discussions which give more details of the processes of linkage editing and loading.

We must now investigate what it is that the Assembler actually does to ensure that its Location Counter is indeed divisible by the desired quantity. Suppose in some program that after a sequence of instructions has 'been processed the value of the LC is $12E_{16}$, so that if another machine instruction were assembled at this point it would begin on a halfword boundary between two fullword boundaries (recall that instruction addresses need only be divisible by 2). Suppose also that the next statement is not a machine instruction statement but is DC F'8' instead. To assemble the four bytes representing the constant (namely 00000008_{16}) beginning at $12E_{16}$ would be incorrect, since an instruction which referred to the constant might require that its memory address be on a fullword boundary. To avoid such an erroneous situation, the Assembler will automatically skip enough bytes to obtain the desired boundary alignment. Thus in this simple example the LC would be increased to 130_{16} before the fullword constant is assembled into the program, and the LC would have a value of 134_{16} after the constant is processed rather than the value of 132_{16} which would be the case if no automatic alignment had been performed. An automatic alignment is not performed in the following circumstances:

1) it isn't needed (that is, the LC happens by chance to fall on the desired boundary); or
2) the type of constant specified doesn't call for it (which is the case for types C, B, and X); or
3) a length modifier is present.

A length modifier allows the programmer to specify the exact length of a constant, and is written immediately following the letter which specifies the data type, in the form .

Ln

where n is either an unsigned decimal self-defining term, or a positive absolute expression enclosed in parentheses. For example, the statements

```
DC   FL3'8'        and        DC   FL(2*4-5)'8'
```

would both cause the constant 000008_{16} to be assembled beginning at the value of the LC when the DC statement was encountered; no boundary alignment is performed. Because alignment is automatic only when the length is implied (that is, no length modifier is given), the two statements

```
     DC   F'8'          and          DC   FL4'8'
```

while defining the same constant may give different results since the former is automatically aligned and the latter is not. (As usual, there is occasionally a little more to the use of a length modifier than is stated here, but what has been omitted, namely, bit-length specifications, will be of no importance or interest until later.)

One further effect of automatic boundary alignment occurs when a symbol appears as the name field entry in a DC assembler instruction statement. Suppose as before that the value of the LC is, $12E_{16}$ when each of the following statements is encountered.

```
IMPLIED  DC   F'8'
EXPLICIT DC   FL4'8'
```

Figure 13.1 Implied and Explicit Length Specifications

Because no boundary alignment is performed in the latter case it is clear that the value of the symbol EXPLICIT will be $12E_{16}$. In the former case, however, two bytes must be skipped by the Assembler to achieve the required boundary alignment Implied by type F. Since we will want to be able to refer to the constant by using the symbol IMPLIED, it is also clear that it should have the value given to the location of the leftmost byte of the constant, namely 130_{16}. Thus if a symbol is to be defined, it is given its value after bytes are skipped to achieve boundary alignment. In fact, a general rule may be stated: the Assembler will never automatically assign the value of a symbol to the location of skipped bytes. (The programmer can find ways to do so if he is so inclined.) This includes the case where a byte must be skipped to ensure that an instruction begins on a halfword boundary. When bytes are skipped to achieve alignment of a following constant.or instruction, the Assembler will insert zeros into the bytes skipped.

We are also in a position now to describe the length attribute of a symbol, which was first mentioned in Section 10. If a symbol appears in the name field entry of a DC instruction, then the length attribute of the symbol is the length in bytes of the first constant assembled. (Cases where more than one constant may be assembled will be treated shortly.) Thus in the examples in Figure 13.1, both symbols have length attributes of 4; and in the machine instruction statement given in Figure 9.5 the length attribute of the symbol LØAD would be 2, since LR is an RR-type instruction of length two bytes.

A duplication factor (sometimes called a multiplicity, replication, or repetition factor) specifies the number of times the constant is to be duplicated, and is written immediately preceding the letter which specifies the constant type. It may be either an unsigned decimal self-defining term, or a positive absolute expression enclosed in parentheses. For example, the statements DC 3F'8' and DC (5/2+1)F'8' are equivalent to writing the statement DC F'8' three times in succession. And because more than one operand may (for the larger Assemblers) be written in the operand field entry of a DC instruction, we could also achieve the same result by writing DC F'8',F'8',F'8' . There is still one more way of defining multiple constants (again, for the larger of the System/360 Assemblers) which we will mention after discussing some of the other types of constants which will be of use in future examples.

The type H constant is quite similar to type F, in that two's complement integer conversion is specified. The only difference is in the default values assumed for length and alignment, which assign a halfword integer to two bytes aligned- on a halfword boundary. Thus the statement DC H'-10' would cause the constant $FFF6_{16}$ to be assembl and placed on the next available halfword boundary. If an explicit length is given, there is no difference between constants of types H and F, so that FL3'8' and HL3'8' are for all practical purposes identical operands.

The following discussion deals with numerous technical matters in a fairly loose way -- rather than give explicit rules at once we will continue to use examples to illustrate the problems involved. The rules will be summarized in a short table at the end of the section.

'The three useful constant types C, X, and B differ from F and H in that no default values are assumed for either length or alignment. For example, the five bytes required to store the constant generated by the statement
DC C'12345' will be placed by the Assembler at the next available address given by the current value of the LC. If a particular boundary alignment is desired, extra steps must be taken which will be described later in this section. The method of writing such constants is, as might be guessed, the same'as for writing character, hexadecimal, and binary self-defining terms, except that the limitations on length and value are different. In the case of self-defining terms, the value of the term was restricted to being less than 2^{24}, whereas much longer constants can be defined with the DC instruction. Thus one can define constants in statements such as in Figure 13.2 below.

```
TITLE  DC   C'THIS IS A LØNG CHARACTER CØNSTANT'
DIGITS DC   X'8462AFCB975310'
```

Figure 13.2 Examples of Character and Hexadecimal Constants

In the discussion of data converted according to types F and Hit was reasonable that the resulting binary numbers should be placed with the least significant digit at the right-hand end of the desired storage area, and that the sign bit should be extended to the left. In all the examples given, the constants were small enough to fit safely in the allotted space. The problem may arise as to what should be done if (1) the constant is too small to occupy fully the number of bite allocated for it by the length specification (whether an explicit length modifier or the default length is used), or if (2) the constant is too large to fit in the allotted space. Some examples of such cases are given in Figure 13.3, along with the constants actually stored by the Aasembler. The rules used to determine the final values of the constants are given below.

Constant too Large	Assembled Value	Constant too Small	Assembled Value
~‘65537’	000116	H'2'	0002_{16}
FL1'-300'	$D4_{16}$	FL1'-6'	FA_{16}
CL3'SMITH'	$E2D4C9_{16}$	CL3'S'	$E24040_{16}$
XL2'56789'	6789_{16}	X'56789'	056789_{16}
BL1'100100100'	001001002	B'101'	000001012

Figure 13.3 Examples of Truncated and Padded Constants

For all of the constants on the left, some part of the true value must be truncated to make it fit into the allotted space, since a length is specified in each case. For all the constant types we are discussing except C, excess information is dropped at the left end of the constant, and the rightmost portion is what is eventually assembled; for character constants the excess is trimmed off the right end, as may be verified in the example above. Note that the special rules concerning the apostrophe and ampersand in character self-defining terms also apply to character constants.

For the constants on the right side of Figure 13.3, the opposite situation occurs: in each case the space allotted (either explicitly or implicitly) is more than is required to hold the significant bits of the , given. constants. For the examples of types H and F, the assembled value is simply the rightmost part of an indefinite-length representation in which the sign bit has been extended to the left; this is as has been customary up to now. In the character example, the single letter "S" has been padded with two blanks (with EBCDIC representation 40_{16}) on the right side to fill out the constant to the required three bytes. The last two examples in the right column require further explanation. As was mentioned earlier in this section, no default lengths are assumed for data of types C, X, and B; the general rule is that in the absence of any limitations, the Assembler will use just enough bytes for the constant to ensure that no information is lost, and no more. Thus the lengths of the constants in Figure 13.2 are 33 and 7 bytes respectively (these also are the length attributes of the symbols TITLE and DIGITS); no information has been lost, and no padding was required.

In the Last two examples in Figure 13.3 some padding with zeros was required et the left end of the constants to fill out the partially-specified byte.

Before discussing literals and the definition of storage areas, wc will introduce another type of constant which is'of great use and broad applicability In Assembler Language programming: this is the type A, or address, constant (sometimes abbreviated "adcon"). An address' constant is written differently from the other types we have considered, since the constant is delimited by parentheses rather than apostrophes, as in A(10). The utility of address constants is a consequence of the fact that the constant may be any expression, absolute or relocatable. The latter case of course requires many other considerations having to do with processing by the Linkage Editor and Resident Supervisor, so for the time being we will restrict our attention to cases where the constant in an address constant is an absolute expression.

The A-type constant is similar to F-type constants in that a length of four bytes and a fullword boundary alignment are implied; thus A(10) and F'10' are equivalent operands, as are AL4(10) and FL4'10' . A major difference lies in the ability to specify constants such as A(X'12E') and A(C' ') (which are the same as F'302' and F'64' respectively), in 'which the use of such expressions may greatly simplify the programming task. In particular one may define constants using operands such as A(ABS425) where the symbol ABS425 may have been defined in an EQU statement (as in Section 11) to have some particular value. Though the utility of such constructs is not apparent now, we will see through later examples that clarity and simplicity can be gained through their use.

One further facility is provided by the larger System/360 Assemblers for conversions of types A, F, and H: the value specified may actually be a sequence of values separated by commas (and no blanks), as in DC F'8,8,8' which, as was indicated earlier, is equivalent to DC 3F'8' and DC F'8',F'8',F'8' . Which one is used is largely a matter of taste and convenience; for example, it is simple to specify a group of constants by the use of a statement such as TABLE DC F'1,2,3,4,5,6,7,8,9,10' where each generated constant is a fullword integer aligned on a fullword boundary. In all such cases where multiple constants are specified, the symbol in the name field entry (in this example, TABLE) is given a value and length attribute associated with the first constant generated. It is not possible to specify multiple values in constants of types B, C, and X.

The short table in Figure 13.4 summarizes some of the rules given above for writing operands in DC instructions. The complete set of rules is summarized in the Appendix.

Type	Maximum Length	Implied Length	Implied Alignment	Value is Specified by	Delimiter Used	Truncation, Padding on	Multiple Values?
H	8	2	halfword	decimal digits	' '	left	yes
F	8	4	fullword	decimal digits	' '	left	yes
A	4	4	fullword	any expression	( )	left	yes
B	256	*	none	binary digits	' '	left	no
C	256	*	none	characters	' '	right	no
X	256	*	none	hex digits	' '	left	no

(* the implied length is the minimum number of bytes required to contain all the given information)

Figure 13.4 Summary of Rules for Certain DC Operands

It often occurs that a storage area is needed in a program which need not be initialized to some value by the use of a DC instruction. This facility is provided by the DS ("Define Storage') assembler instruction, which is almost identical in use to the DC instruction. The rules for writing the operand field entry are the same, with the exception that the specification of a value is optionai. Thus the statements DS F and DS F'8' will both cause the Assembler to reserve a four-byte area on a fullword boundary, but no constant will be assembled, even though one is specified in the latter case. Statements-such as DS C'MESSAGE' will reserve an area whose length is computed by the Assembler from the length of the given constant (7 bytes), but there will be no constant assembled into the reserved area. Large blocks of storage may be reserved by statements such as

```
     STØRAGE   DS    100F
```

which reserves one hundred aligned fullwords and assigns to the symbol

STØRAGE the location of the first. Note also that the two statements

```
     AREA1    DS   80C           and           AREA2   DS    CL80
```

both define storage areas of length 80 bytes, but the length attributes of the symbols AREA1 and AREA2 are 1 and 80 respectively, which may be of interest in a program. Note in the former of these cases that in the absence of either a constant or an explicit length, an implied length of one byte is assumed for the C-type specification; the same is true for types B and X, so that DS B and DS X would both cause a single byte to be reserved.

One special case arises in the use of the DS instruction when a duplication factor of zero is specified. In such a case any necessary boundary alignment implied by the type is performed, and then, if a name field symbol is present, the adjusted value of the LC is assigned to its value and its length attribute is determined from the operand; no space is reserved. Thus a DS instruction with duplication factor zero can be used to force a boundary alignment which would not be available otherwise. For example, the two sets of statements

```
     WØRD  DS   OF                 and                  DS    OF
           DC   C'WØRD'                          WØRD   DC    C'WØRD'
```

both serve to define a four-byte character constant on a fullword boundary addressed by the symbol WØRD, which would not in general have been the case if DC C'WØRD' or DC CL4'WØRD' had been specified. Note that DC A(C'WØRD') is incorrect: because the operand in parentheses must be an expression, and because C'WØRD' contains more than the allowed maximum of three characters which is required by the rules for forming self-defining terms, the expression which-forms the value for the address constant is invalid.

If a duplication factor of zero is used in a DC instruction, it behaves just as would the corresponding DS instruction. When bytes are skipped to perform alignments implied by DS statements, the Assembler does not put zeros in the skipped bytes.

This brings us finally to the subject of literals. It often occurs in programs that some constant must be defined which is used only as a constant.

In the sample program segment in Figure 12.1, the two quantities in the fullwords named N and ØNE are both defined by DC instructions, but it is implicit in the use of the symbol "ØNE" that the contents of that fullword should retain the integer value +1 throughout execution of the program. It is of course possible to use constructions such as `EIGHT DC F'5'` in a program, but this cannot be of much help in making the program easier to read or understand, particularly if some part of the program stores data of varying values in that area. The Assembler provides a simple and convenient means for simultaneously defining constants and referring to them, through the use of literals.

A literal is a special kind of symbol, where the value of the contents of the storage area referred to by the literal is contained in the literal itself. A literal is written as an equal sign (=) followed by an operand which conforms to the rules for operand field entries in DC instructions. The following are examples of literals.

```
=F'1'          =C'LØNGLITERAL'     =BL2'111101'
=H'1'          =CL7'BLANK'         =X'765432A'
=A(1)          =F'1,2,3,4'         =AL3(5,X'D7'/C'.')
```

Literals may be used in most places where symbols are permitted, with the following exceptions:

(1) a literal is a term which may not be combined with other terms (thus `IC 0,=F'1'+3` is illegal);

(2) an instruction may not store or modify a literal (thus `ST 7,=F'1'` is illegal);

(3) a literal may not be specified in an address constant (about which more later) (so that `A(=F'1')` is illegal);

(4) multiple operands may not be specified, but multiple values may;

(5) the duplication factor may not be zero;

(6) the alignment of the data described in the literal is that implied by the constant type (so that `L 2,=X'2B'` will probably cause a specification exception).

To illustrate the use of a literal *in* , a program segment, we could rewrite the example in Figure 12.1 in the form given in Figure 13.5 below.

```
               BALR    6,0
               USING   BEGIN,6
      BEGIN    L       2,N
               A       2,=F'1'
               ST      2,N
           "-"----m---I-
      N        DC      F'8'
```

Figure 13.5 Sample Program Using a Literal

In this case the programmer has been relieved of the duty of defining a constant and creating a symbol by which to refer to it, as was the case previously. For this gain in ease of referring to constants there is a corresponding loss in the precision with which one may specify exactly where the constant is to be located, since this must now be determined by the Assembler (a small amount of control is left to the programmer). As literals are encountered by the Assembler in the course of scanning the source program, a separate internal table -- called a literal pool -- is formed which contains all the literals encountered, with duplicates eliminated. This allows the programmer to make liberal use of literals with some small assurance that he will not generate an excessive number of constants. These are placed in the program at an appropriate location, and the Assembler then computes the required displacements which allow the constants to be addressed. We will use literals in many places throughout this presentation, and it should be borne in mind at all times that a literal is a special symbol, and not a piece of data, a storage area, or a value, which are common misconceptions in the use of literals.

We have now covered enough basic material to be able to examine many of the instructions of System/360 in the context of actual programs. In the next several sections we will discuss the use of the general registers for a variety of purposes, and give some examples of program segments which illustrate typical uses of the instruction set.

14. GENERAL REGISTER SHIFTING AND DATA TRANSMISSION

In this section we will discuss the instructions which cause data to be transmitted among the general purpose registers, between the registers and memory, and within the individual registers themselves. Some of the instructions will be treated in detail,- since they are the first of the RS type to be examined.

A notational convenience will be introduced here: because we will often have need to use the phrase "general purpose register r_1" where r_1 indicates the value supplied for an operand in the operand field entry of a machine instruction statement, we will use the abbreviation "Rr_1" instead. Thus if r_1 has the value 5, the register being referred to is R5.

We will first examine the instructions which transmit data between the GPRs and memory. The most important of these are the L (Load) and ST (Store) instructions, which were encountered in several earlier examples. Both are of type RX; both require the effective address to be divisible by 4, so that the use of a fullword operand is indicated. The instruction

```
            L       r1,d2(x2,b2)
```

causes the fullword second operand to replace the contents of Rr_1. The original contents of Rr_1 are lost, and the contents of the fullword area in memory remain unchanged. As a reminder, the term "operand" was used here to mean the data referred to at execution time by the effective address, which was computed from components of the instruction determined during assembly from the second operand in the operand field entry of the instruction statement. As mentioned before, which meaning of the word "operand" is intended will usually be clear from context.

For example, to set the contents of R9 to zero we could write

```
        L    9,=F'0'
```

and to set it to the maximum negative number,

```
        L    9, =F'-2147483648'
```

would suffice.

The inverse operation ST is written explicitly as

ST $r_1,d_2(x_2,b_2)$

and causes the contents of Rr_1 to replace the contents of the fullword area of memory at the effective address of the second operand. The contents of the register are unchanged, and the original contents of the fullword area of memory are lost. For example, to duplicate at B the contents of the fullword at A, we could write

```
        L   0,A
        ST  0,B
```

and to exchange the contents of the fullwords at A and B, we could write

```
    L    1,B          L    0,A          L    0,A              L    0,A
    L    0,A     or   L    1,B     or   L    1,B     but    , ST   0,B
    ST    0,B         ST    0,B         ST   1,A     not      L    0,B
    ST   1,A          ST   1,A          ST   0,B              ST   0,A
```

where we have assumed that R1 is not being used as a base register. The use of L and ST in situations where indexing is desired will be treated later. Both of these instructions are subject to interruptions due to specification and addressing errors, which were mentioned in Section 5; one further interruption may be caused by memory-protection, en optional feature available on System/360 which allows some degree of supervision over the areas of memory acceaeible to a given program. We will examine memory protection in more detail when interruptions are discussed.

It is occasionally necessary or desirable to be able to transmit information between memory and several registers. This can be done with a sequence of L or ST instructions, as in

```
    L    1,A            ST 1,B
    L    2,A+4     or   ST 2,B+4
    L    3,A+8          ST 3,B+8
```

If the number of registers is large, however, this can be cumbersome and slow, and it is more convenient in many cases to use the **LM** (Load Multiple) and **STM** (Store Multiple) **instructions.** Each of these is an RS-type instruction for which three operands must be specified in the operand field entry, as follows:

LM (or STM) $r_1,r_3,d_2(b_2)$

where the components of the assembled instruction are pictured in Figure 14.1.

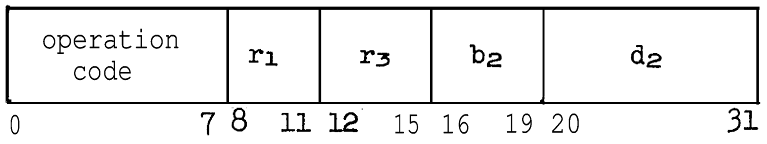


Figure 14.1 Components of an RS Instruction

As usual, r_1 and r_3 must be positive absolute expressions of value 15 or less, and the base and displacement may be given explicitly or left for the Assembler to compute from the value of a symbol or other relocatable expression. The meanings of the register specification digits in the STM instruction are as follows: beginning with Rr_1, transmit the registers in order of increasing number to the **successive fullwords** in memory which start at the effective address of the second operand, until Rr_3 has been transmitted. If r_3 is equal to r_1, only one register is transmitted. If r_3 is less than r_1 then Rr_1 through R15 will be transmitted, followed by R0 through Rr_3; thus R0 may be considered to follow after R15, so that the general registers "wrap around" from the highest to lowest numbered. The LM instruction follows the same rules except that the registers are loaded in sequence from successive fullwords in memory.

For example, **LM 2,6,=5F'0'** would cause the contents of R2, R3, R4, R5, and R6 to be set to zero. Similarly, STM 0,15,SAVE would cause the contents of all sixteen registers to be stored beginning at SAVE, which could be defined in a statement such as SAVE DS 16F which ensures that the proper boundary alignment will be specified for the second operand address. If we assume that R1 contains the address of a list of four fullword constants, we could load them into R7 through R10 by executing the statement LM 7,10,0(1) and if we assume that R13 contains the address of a register save area, then STM 14,12,12(13) would store R14, R15, R0,...R12 in successive fullwords, beginning with the fourth fullword of the area. These last two example8 illustrate certain conventions commonly used in communicatingwith subroutines, which will be treated in detail later. As a final example, suppose we wish to exchange the contents of RO through R7, as a block, with the contents of R8 through R15. We could then write

```
        STM  0,15,SAVE                  STM  8,7,SAVE
        LM    8,7,SAVE        or        I M   0,15,SAVE
        ---                             ---
   SAVE Ds    16F                 -SAVE DS    16F
```

One small but important detail in this example should be noted: one of the general registers must have been specified as a base register so that SAVE could be addressed. The STM and LM instructions will work correctly, since the calculation of the effective address is performed before the execute phase of the LM instruction cycle begins. When execution is completed, however, the base register has been changed, so either the Assembler must be informed that the base register is changed, or the correct value must be put back into the original base register.

The transmission of halfword data between memory and registers is somewhat more complicated, because a halfword requires only half of a general register. The relevant instructions, LH (Load Halfword) and STH (Store Halfword) are similar to L and ST; both are RX instructions, and the operand field entry is written the same way. STH is the simpler of the two: the rightmost 16 bits (the right half) of Rr_1 replaces the halfword at the effective address of the second operand, and Rr_1 remains unchanged, If the contents of the register represent an integer too large to be correctly represented as a 16-bit two's complement integer, some significance is lost; no indication is made that the halfword in memory may not have the desired value. (An example illustrating this will be given shortly,) Conversely, when data is being transmitted from memory to a register by the LH Instruction, it is reasonable to assume that the programmer wants to perform some arithmetic operations on the value transmitted, so that the data should occupy the entire register with the least significant bit at the right-hand end. To give a correct representation in the 32-bit register, the sign bit of the 16-bit halfword operand must therefore be extended to the left to occupy the left half of the general register. One may visualize this process as taking place in two steps,. The halfword operand is brought from memory and placed in the Memory Data Register (MDR), which is an internal register used for communicating between the CPU and memory. The leftmost bit of the halfword is duplicated to the left by 16 positions, providing a 32-bit representation of the original 16-bit two's complement operand. The resulting 32 bits are then transmitted to the designated general register. Though none of the models of System/360 use the MDR inprecisely this fashion, we will find that the descriptions of many instructions can be simplified considerably by supposing it to take an active part in the handling of data passing between memory and the CPU. Note that there is also an instruction with mnemonic MDR; we will indicate which is meant if there is a possibility of confusing the two. Thus the statements LH 0,=H'1' and LH 0,=H'-1' would cause the contents of RO to be set to 00000001_{16} and $FFFFFFFF_{16}$ respectively. As long as the value of the halfword operand X involved satisfies $-2^{15} \leq X < 2^{15}$ it can be correctly represented in 16 bits and will therefore be correctly transmitted by LH and STH instructions. If this is not the case, situations such as those illustrated in the next two examples can arise.

Suppose the sequence of instructions given in Figure 14.2 is executed. The contents of the registers is given in the comments field of the instructions; the notation C(RO) means "contents of RO", and X'n' means the same thing as n_{16}, as in the definition of hexadecimal constants.

```
         L    0,B      C(RO)=X'00010001'
       STH    0,A      C(A)=X'0001'
        LH    1,A      C(R1)=X'00000001'
       ---
    A   D S   H
    B   DC    F'65537'
```

Figure 14.2 Loss of Significant Digits when Using STH

The contents of RO and R1 are different because the quantity in RO being stored by the second instruction is too large. A more awkward result is illustrated in Figure 14.3.

```
         L    0,=F'65535'      C(RO)=X'0000FFFF'
       STH    0,A              C(A)=X'FFFF'
        LH    1,A              C(R1)=X'FFFFFFFF'
       ---
A       DS    H
```

Figure 14.3 Loss of Significant Digits when Using STH

In this case the result in R1 has a different aign and considerably different magnitude from the original operand. From these two examples it is clear that the programmer who chooses to use halfword data must exercise care to be sure he understands what can happen when storing or loading such quantities.

Two further instructions used for transmitting data between the general registers and memory are IC (Insert Character) and STC (Store Character). (IC was used in the addressing examples in Section 5.) The operand field entry is written in exactly the same form as for L and ST, and no particular boundary alignment is required for the address of the second operand, since the data being moved in this case is contained in a single byte.

The instruction STC $r_1,d_2(x_2,b_2)$ causes the rightmost byte of Rr_1 to replace the byte at the effective second operand address. The inverse operation is called "Insert Character' rather than "Load Character", because the specified byte from memory is placed in the rightmost 8 bits of the register without disturbing the remaining 24; no sign extension is performed. Ae an example, the instructions below can be used to reverse the order of the two-characters in the character constant at X and place the result at Y.

```
      IC    0,x
      STC   0,Y+1
      IC    0,X+1
      STC   0,Y
      ---
X     DC    C'AB'
Y     Ds    CL2       BECØMES C'BA'
```

Occasionally when memory space is at a premium it is convenient to use e single byte to contain a small integer constant; its value may be placed in a register using the following instruction sequence.

```
         L    1,=F'0'      CLEAR REGISTER
         IC   1,LITLCØN    INSERT CØNSTANT
LITLCØN  DC   FL1'53'
```

None of the instructions discussed up to now has had any effect on the Condition Code (CC). We now turn our attention to five RR-type instructions which transmit date among the general registers, four of which can change the value of the CC. The instructions are LR (Load Register), LTR (Load and Test Register), LCR (Load Complement Register), LNR (Load Negative Register), end LPR (Load Positive Register). The LR instruction was used in the machine instruction statement in Figure 9.5; it is the one instruction of these flve which does not set the CC. The operand field entry, as noted in Section 11, is written r_1,r_2 and the action of each instruction is summarized in Figure 14.4 below, Note that r_2 need not differfran r_1.

Instruction	Action	CC Values
LR	$C(Rr_1) \leftarrow C(Rr_2)$	not set
LTR	$C(Rr_1) \leftarrow C(Rr_2)$	0,1,2
LCR	$C(Rr_1) \leftarrow -C(Rr_2)$	0,1,2,3
LPR	$C(Rr_1) \leftarrow \lvert C(Rr_2)\rvert$	0,2,3
LNR	$C(Rr_1) \leftarrow -\lvert C(Rr_2)\rvert$	0,1

Figure 14.4 Action of Certain General Register Instructions

The meanings of the CC settings are given below.

CC	Meaning
0	Result is Zero
1	Result is Negative
2	Result is Positive
3	Result has Overflowed

Figure 14.5 Condition Code Settings

As can be seen from Figure 14.4, the actions of LR and LTR are identical except that LTR also sets the CC. It is not uncommon to test the contents of a register by writing an instruction such as LTR 4,4 which has no effect other then to set the CC, which may then be tested by a BC or BCR instruction, which will be discussed later, 'For the other three instructions, the arithmetic operations are those implied by a 32-bit two's complement , representation; thus overflow can occur during execution of LCR or LPR only if C(Rr2) is the maximum negative number, -2^{31}, and no overflow can occur during execution of LNR because all representable positive values have a corresponding two's complement representation of their negative values. The following short instruction sequence illustrates possible uses of the instructions.

```
LM    2,3,=F'1,0'    c(R2)=1,    C(R3)=0, CC NØT SET
LR    7,3            C(R7)=0,    CC NØT SET
LTR   2,2            C(R2)=1,    CC=2
LNR   1,7            C(R1)=0,    CC=0
LCR   4,2            C(R4)=-1,   CC=1
LPR   0,4            C(RO)=+1,   c c = 2
LNR   5,2            C(R5)=-1,   CC=1
```

Figure 14.6 Example of Use of Certain RR Instructions

Two common errors for beginning programmers are to confuse the LR and L instructions, and to try to use an "STR" instruction to "store" one register into another. By substituting L for LR, one can occasionally generate coding errors which are undetected by the Assembler: for example, L 5,8 is a valid instruction referring to location 8 in memory, which is probably not the programmer's intention. As an aid to remembering the difference between related instructions of differing types, note that almost all of the RR instructions end in the letter "R", and the RX, SI, or RS instructions end in other letters,

The shifting instructions to be described next are more interesting, since they allow the programmer to manipulate data in more varied ways than the instructions described up to now. All of the eight shift instructions are RS-type; they differ from LM and STM in the important respect that the r_3 register spacificetion digit (see Figure 14.b) is ignored when the instructions are executed, and thus the operand field entry for shift instructions is written

$$r_1,d_2(b_2)$$

with the r_3 operand omitted. For all of the shifting instructions, the number of bit positions to be shifted is determined from the low-order six bits of the effective address; this allows for the specification of shift amounts between 0 and 63 inclusive. The simplest shifting instructions are SRL (Shift Right Logical) and SLL (Shift Left Logical); we will examine these first.

The basic operation in shifting is the unit shift, in which each bit moves to the right or left by one binary digit position; the vacated bit position on the left or right end is handled differently for logical and arithmetic shift instructions. For the logical shifts, the vacated bit position is always set to zero, and any bits shifted off the opposite end are lost and ignored; for arithmetic shifts this is true only at the right end. Thus, if the contents of R8 are 87654321_{16} and the instruction SLL 8,1 is executed, the result in R8 will be $0ECA8642_{16}$. Note that we could have written SLL 8,1(0) also, because the explicit use of 0 a6 a base register specification.dig3.t causes no base register to be used in the calculation of an effective address. Again supposing R8 to contain 87654321_{16} and R3 to contain $82F3A2B5_{16}$, execution of the instruction SRL 8,16(3) would cause the contents of R8 to be shifted right $xxxxxxB5_{16}+10_{16}$ = 05_{16} (modulo 40_{16}) bit positions, leaving $043B2A19_{16}$ a6 the result.

For a simple example of the use of the single-register logical shift instructions, suppose we have a large table of data, where each entry is six byte6 long and is aligned on a halfword boundary. Suppose also that the first three bytes contain character information of some sort, and the remaining three bytes are to contain a 24-bit two's complement integer value associated with the characters. We want to load and store the integer value into and from R5, where it will be used for some purpose in the program. Now it is clear that L and ST cannot be used, since it is not possible to obtain the proper alignment of the operand in memory; similarly, LH and STH handle only two of the three bytes. A simple solution is to pack the integer value so that its rightmost eight bits occupy the first byte, and the leftmost 16 bit6 occupy the second and third bytes. Suppose R5 contains FFFA620B$_{16}$, and R12 contains the address of the first byte of the particular 6-byte data entry under consideration. Then the sequence of instructions below can be used to peck the number into memory. (The letters XXYYZZ are meant to represent the hex digits of the three characters in the da-ta entry.)

```
STC   5,3(0,12)    C(DATA ENTRY) = XXYYZZ0B----
SRL   5,8          C(R5) = 00FFFA62
STH   5,4(0,12)    C(DATA ENTRY) = XXYYZZ0BFA62
```

To show that the desired value can be correctly retrieved, we execute the inverse instruction sequence.

```
LH    5,4(0,12)    C(R5)=FFFFFA62
SLL   5,8          C(R5)=FFFA6200
IC    5,3(0,12)    C(R5)=FFFA620B
```

This example also illustrates a situation where the need for efficient use of memory apace outweighs the extra time required to access and store the needed value. If the data entry were expanded to eight bytes, with the characters occupying the first three bytes and the associated value in the last four, then simple L 'and ST instructions could be used, with a considerable increase in speed (an approximate factor of 3) for thia segment of code. Such considerations may be quite important for programs which process large amounts of data -- the example typifies what is called the trade-off' between space and speed. We will see a number of examples where the expenditure of memory space may result in increased processing speeds.

We could also have arranged the data so that the three-byte integer value occupied the first three bytes of the data entry, and the characters occupied the last three byte6. The integer value would then be stored in memory with its bits in the -proper arithmetic sequence; the instructions needed to load the value into R5 would be as follows, assuming that the data entry contained FA620BXXYYZZ.

```
LH    5,0(0,12)    C(R5)=FFFFFA62
SL    5,8          C(R5)=FFFA6200
IC    5,2(0, 12)   C(R5)=FFFA620B
```

It is apparent that the particular arrangement of the data in memory may depend on the programmer's inclinationa, as well as on considerations of ease of programming or speed of execution.

The double-length logical shift **instructions** **SLDL** (Shift Left Double **Logical)** and SRDL (Shift Right Double Logical) work in exactly the same way as SLL and SRL except that a pair of registers is shifted. The register specified by the first operand (Rr_1) must be an even-numbered register; otherwise a specification exception will occur. The next higher numbered register is the low-order half of the double-length register pair, with bits shifted out the right end of Rr_1 entering the left end of Rr_1+1, and vice versa. (This is one of the reasons for showing the general registers in pairs in Figure 3.7.)

To illustrate a trivial application of these two Instructions, suppose we wish to reverse the order of the **halfwords** at A and A+2, where A is on a **fullword** boundary. Then each of the following code sequences will perform the desired task.

```
LH    2,A        LH    2,A        L     2,A        LH    2,A
SRDL  2,16       SRDL  2,16       STH   2,A        LH    3,A+2
LH    2,A+2      LH    2,A+2      SRL   2,16       STH   2,A+2
SLDL  2,16       SRDL  2,16       STH   2,A+2      STH   3,A
ST    2,A        ST    3,A
```

(The third and fourth examples illustrate that when the data happen to be aligned in a particular way, there may be simpler ways to arrive at the same result.) To take a less trivial example, suppose that in a certain application we need to access some integer data which has been packed so that four positive integers fit into a fullword, as shown in Figure 14.7.

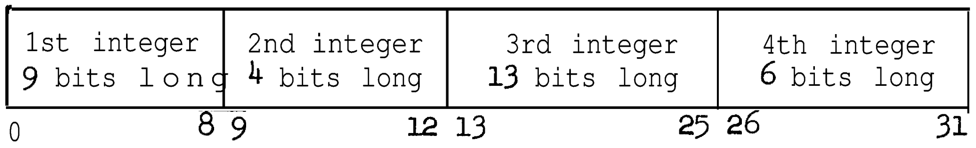


Figure 14.7 Four Integers Packed in a **Fullword**

A sequence of instructions which unpacks the integers and places them in the **fullwords** labeled FIRST, **SECØND,** THIRD, and **FØURTH,** follows; assume that **R9** contains the address of the data word. The comment statements give the binary contents of RO and **Rl:** the bits of the integers are labeled A, **B,** C, **and D;** **X** represents a bit whose value is unknown, **and** 0 is a 0 bit. The " . " is simply to indicate the boundary between RO and **Rl.**

```
         L       0,0(0,9)        GET DATA FULLWORD
*  AAAAAAAAABBBBCCCCCCCCCCCCCDDDDDD.XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         SRDL 0,6                SHIFT OFF 6 BITS
*  000000AAAAAAAAABBBBCCCCCCCCCCCCC.DDDDDDXXXXXXXXXXXXXXXXXXXXXXXXXX
         SRL     1,26            MOVE TO RIGHT END Of R1
*  000000AAAAAAAAABBBBCCCCCCCCCCCCC.00000000000000000000000000DDDDDD
         ST      1,FOURTH        STORE FOURTH INTEGER
         SRDL 0,13               SHIFT OFF 13 BITS
*  0000000000000000000AAAAAAAAABBBB.CCCCCCCCCCCCC0000000000000000000
         SRL     1,19            MOVE TO RIGHT END OF R1
*  0000000000000000000AAAAAAAAABBBB.0000000000000000000CCCCCCCCCCCCC
         ST      1,THIRD         STORE THIRD INTEGER
         SRDL 0,4                SHIFT OFF 4 BITS
*  00000000000000000000000AAAAAAAAA.BBBB0000000000000000000CCCCCCCCC
         ST      0,FIRST         STORE 1ST INTEGER FROM R1
         SRL     1,28            POSITION SECOND INTEGER
*  00000000000000000000000AAAAAAAAA,0000000000000000000000000000BBBB
         ST      1,SECOND        STORE SECOND INTEGER
```

Another code sequence to do the **same** task **is:**

```
         L       2,=F'0'         GET A 0 CONSTANT FOR CLEARING R0
         L       1,0(0,9)        GET DATA FULLUORD
         LR      0,2             CLEAR R0
         SLDL    0,9             SHIFT 9 BITS INTO R0
         ST      0,FIRST         STORE FIRST INTEGER
         LR      0,2             CLEAR R0
         SLDL    0,4             SHIFT 4 BITS INTO R0
         ST      0, SECOND
         LR      0,2             CLEAR R0
         SLDL    0,13            SHIFT 13 BITS INTO R0
         ST      0,THIRD         STORE THIRD INTEGER
         SRL     1,26            REPOSITION FOURTH INTEGER
         ST      1,FOURTH        STORE FINAL VALUE
```

In this example the SRL 1,26 replaces the LR and SLDL used in the first three steps, because It results in less coda and slightly faster execution. The overall raving is quite small, but the choice serves es an example of a small economy which, if applied in several key places in a large program, could result in significant savings,

The arithmetic shift instructions are almost identical to the logical shift instructions, with the differences being in the setting *of* the *CC* and the treatment of the sign bit. The instructions are SLA (Shift Left

Arithmetic), SRA (Shift Right Arithmetic), SLDA (Shift Left Double Arithmetic), and SRDA (Shift Right Double Arithmetic). On <u>right</u> shifts, the sign bit is duplicated in the vacated sign position after each unit shift; thus the arithmetic integrity of the shifted operand is maintained. To illustrate the difference between logical and arithmetic shifts, suppose a right shift of two places is performed on a register containing $FFFFFFF8_{16}$:

```
         L    0,=F'-8'              L    0,=F'-8'
         SRL  0,2                   SRA  0,2
```

After the logical shift, $C(R0)=3FFFFFF2_{16}$, and after the arithmetic shift, $C(R0)=FFFFFFF2_{16}$. For positive operands, the SRL and SRA instructions will leave identical results in the register shifted; SRA will set the CC but SRL will not. The instruction SRDA is similar to SRA except that an even-odd register pair is shifted.

For arithmetic <u>left</u> shifts, the situation can be a little more complicated. When an operand is shifted left there is the possibility that one or more significant bits will be lost. This situation is detected by (1) retaining the original sign bit, and (2) indicating an overflow if any bit shifted out of the bit position just to the right of the sign is different from the sign bit. The following code sequence would produce the results indicated.

```
         L    0,=F'-8'     C(R0)=FFFFFFF8, CC UNCHANGED
         SRL  0,2          C(R0)=3FFFFFF2, CC UNCHANGED
         SLA  0,4          C(R0)=7FFFFF20,  CC SET TO 3,ØVERFLØW
```

Condition Code settings produced by the arithmetic shift instructions are given in Figure 14.8.

Instruction	CC = 0	CC = 1	CC = 2	CC = 3
SLA	Result=0	Result<0	Result>0	Overflow
SRA	Result=0	Result<0	Result>0	Impossible
SLDA	Result=0	Result<0	Result>0	Overflow
SRDA	Result=0	Result<0	Result>0	Impossible

Figure 14.8 CC Settings after Arithmetic Shifts

A CC value of 3 is not possible after the SRA and SRDA Instructions. Note that because the result tested for CC settings for SLDA and SRDA is a double-length operand, these instructions provide a simple means for testing whether both registers contain zero: both SRDA 0,0 and SLDA 0,0 will set the CC to zero if R0 and R1 contain zero.

An important characteristic of the arithmetic shift operations Is that they provide a simple means for multiplying by positive and negetive powers of two. Since the bite of an operand shifted left by a unit shift appear with a weight (in the sum forming the value of the operand) which has increased by two, we can see that so long as no overflow occurs, an arithmetic left shift of n places corresponds to multiplication by 2". Similarly, for a unit right shift each bit has a weight which has decreased by two, so that an arithmetic right shift of n places corresponds to division by 2^n. Because such a "division" can appear to produce fractional results, we must examine what happens when bit8 are lost; consider the two following code sequences.

```
L     3,=F'5'        C(R3) = 00000005
SRA   3,1            C(R3) = 00000002

L     3,=F'-5'       C(R3) = FFFFFFFB = -5
SRA   3,1            C(R3) = FFFFFFFD = -3
```

As we might have expected, the lost bit in the first case simply results in the fractional part of 5/2 being lost, so that the result is simply 2. In the second case the result is -3, not -2; this is because the truncation of the fraction part of a number in the two's complement representation has the effect of always forcing the result to the next <u>lower</u> integer value.

As a simple example, suppose we wish to truncate the integer in R9 to the next algebraically lower multiple of 16, unless it is already a multiple of 16. - Roth of-the following code sequences achieve the desired result.

```
SRA  9,4            SRL 9,4
SLA  9,4            SLL 9,4
```

The logical shifts can be used because whatever bit is shifted out of the sign position by the SRL instruction is put back by SLL. If a CC setting is desired to indicate the status of the result, then the first code sequence must be used; if not, the second is preferable because 'it will operate slightly faster, because the CPU need not bother with duplicating the sign bit nor checking for overflow.

To conclude our discussion of shifting, we will re-examine the problem of unpacking the data contained in the fullword pictured in Figure 14.7, on the supposition that the four integers are in signed two's complement representation rather than the unsigned logical representation assumed before. The following code segment stores the four signed integers as required.

```
L       0,0(0,9)       GET DATA WORD
SRDA    0.6            SHIFT 6 BITS INTO R1
SRA     1,26           EXTEND TO RIGHT
ST      1,FOURTH       STORE FULLWORD RESULT
SRDA    0,13           SHIFT OFF 13 MORE BITS
SRA     1,19           SHIFT WITH SIGN EXTENSION
ST      1,THIRD        STORE SIGNED RESULT
SRDA    0,4            SHIFT OFF LAST 4 BITS
ST      0,FIRST        STORE CORRECT FIRST INTEGER
SRA     1,28           EXTEND SECOND INTEGER
ST      1,SECOND       STORE FINAL RESULT
```

Because the number of positions to be shifted by any shift instruction is determined from an effective address, the number of shifts can be specified at execution time. For example, SLL 9,0(4) will shift R9 by an amount determined by the rightmost six bits of the contents of R4. As was the case for the use of relocatable symbols which named areas of memory, the Assembler will compute displacements and assign bases for absolute expressions. If we write the sequence of statements given below, the instructions would be assembled as indicated in the right-hand column.

```
     USING 6,2
A    EQU   10
     SLL   9,12         89902006
     SLL   9,12(0)      8990000C
     SLL   9,A          89902004
```

Thus we can vary the number of shifts at execution by placing appropriate values in R2. We will find that there are relatively few occasions where an absolute expression will be used as the first expression in a USING instruction.

15. CONDITIONAL BRANCHING

In this section we will discuss two branch instructions whose use is fundamental in almost all. programs. The ability to choose alternative courses of action in a program depending on computed results is one of the most distinctive features of a computer, and we will make use of these instructions in most of the remaining program examples. We will examine the conditional branch instructions before continuing our treatment of general register operations, since we will then be able to give more extensive and realistic sample programs to illustrate the points involved.

Because the Condition Code is contained in a two-bit field of the PSW, the possible values which may be assumed by those two bits are 0, 1, 2, and 3. To test for one of these values, either BC or BCR is used; both are called "Branch on Condition" instructions, with BC being of type RX and BCR being of type RR.

If the condition for branching is not met (and how this is determined will be discussed shortly) no action is taken and execution simply proceeds to the next sequential instruction following the BC or BCR.

If the branching condition is met, the branch address must be determined. For the BC instruction, the branch address is the same as the effective address computed as usual from the base, index, and displacement fields of the instruction; for the BCR instruction, the branch address is given by the rightmost 24 bits of the general register specified by the r_2 digit of the instruction unless r_2 is zero, in which case no branch ever occurs. To complete the execution of the branch instruction, the IA portion of the PSW is replaced by the branch address. The next instruction to be fetched will therefore come from the location specified by the branch address. Branch instructions are also called "jump" and "transfer" instructions, in the sense that a jump is made, or control is transferred, to the branch address.

Whether the branch condition is met or not is determined by examining the bits of the register specification digit r_1. Because this digit does not refer to Rr_1, but is treated simply as a bit pattern (called a mask), we will rewrite the operand field entries as $m_1,d_2(x_2,b_2)$ and m_1,r_2 for the RX and RR cases respectively. Thus we can write BC 7,4(8,2) and BCR 9,4 in which the mask fields are 0111_2 and 1001_2 respectively. At execution time, a match is made between the 1 bits of the mask and the value of the CC, as indicated in Figure 15.1

Instruction Bit	Mask Bit Value	CC Value Matched
8	8	0
9	4	1
10	2	2
11	1	3

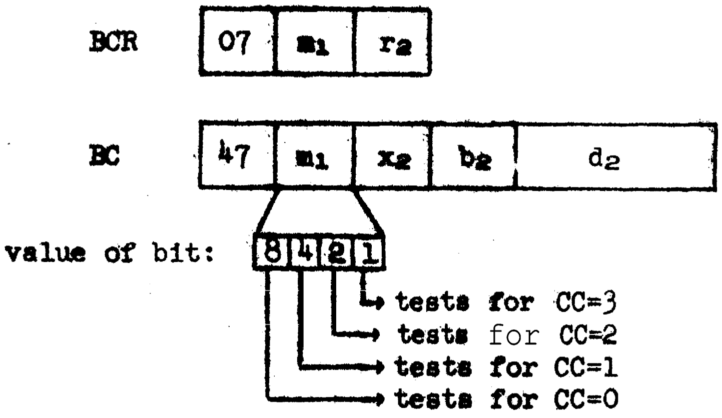


Figure 15.1 Mask Bits and Corresponding CC Values

If the CC has a value which matches a 1 bit in the mask field, the branching condition is met; if the CC has a value which matches a 0 bit in the mask, the branching condition is not met, and no branch occurs. Thus in the examples given above, the BC instruction would branch unless the CC had value 0, and the BCR would branch if the CC had value 0 or 3. Further examples are given below.

1) Branch to X if C(R12)=0.

```
LTR   12,12            SRA   12,0
                 or
BC    8,X              BC    8,X
```

2) Branch to X if C(R0)≠0.

```
LTR   0,0              SLA   0,0
                 or
BC    6,X              BC    7,X
```

(Note that the CC cannot have value 3 after LTR.) In both of the above examples the use of LTR is shorter and faster.

3) Multiply C(R5) by 4 and branch to X if the result does not overflow.

```
SLA   5,2
BC    14,X
```

4) Branch to the address contained in R14.

```
BCR   15,14         (preferred)
   or
BC    15,0(0,14)    (slower)
   or
BC    15,0(14)      (slowest)
```

Since the CC must have a value which matches a bit in the mask, the branch always occurs; this is called an unconditional branch.

5) Place -C(R2) in R8 and branch to X if the result is negative&

```
LCR   8,2
BC    5,X
```

It is not sufficient to use a mask of 4 since the result will also be negative if overflow occurs.

6) A positive nonzero fullword integer at N is to be shifted right as many places as necessary to insure that its rightmost bit is nonzero.

a) Shift left into R4 until R5 has been vacated:

```
         L      5,N         GET INTEGER
         L      4,=F'0'     CLEAR R4
SHIFT    SLDL   4,1         SHIFT LEFT
         LTR    5,5         TEST R5
         BC     7,SHIFT     BRANCH IF NØT ZERØ
         ST     4,N         STØRE RESULT
```

b) Shift right, testing "lost" bits:

```
             L        4,N            GET INTEGER
SHIFT        SRDL     4,1            SHIFT RIGHT
             LTR      5,5            TEST SIGN ØF R5
             BC       10, SHIFT      BRANCH IF NØT -
             SLDL     4,1            MØVE BIT BACK
             ST       4,N            STØRE RESULT
```

Note that this latter example would work for negative integers also if arithmetic shift instructions were used.

This last pair of examples illustrates a loop -- a sequence of instructions which is repeated as many times as is necessary to obtain a desired condition. Loops are such a common aspect of programming that special branch instructions are provided in System/360 which greatly facilitate the coding of loops without either examining or testing the CC; these will be treated in some detail later.

We noted in example 4 above that a mask with all 1 bits provides an unconditional branch (remember that we could have written BCR X'F',14 and BCR B'1111', 14 also), since the branch condition must always be met. There are occasion&when it is useful to be able to execute an instruction with a zero mask field. Thus BC 0,X and BCR 0, any as well as BCR any,0 have no effect; they are sometimes called "no-operation" instructions, and the Assembler actually provides mnemonics for their specification. The instructions NØP s and NØPR r are treated by the Assembler as being the same as BC 0,s and BCR 0,r respectively.

An important use of "no-operation" instructions is in obtaining a desired boundary alignment for a particular instruction. For example, we may wish that an instruction such as BALR 14,15 be followed by an aligned fullword constant such as an address constant; examples of just this sort of usage will be illustrated in the treatment of subroutines. Since BALR is an RR instruction, we must simply insure that its address lies between two fullword boundaries. In a small program it is easy for the programmer to determine the location of the BALR simply by counting, and if it falls on a fullword boundary he can insert a NØPR 0 instruction just before it. However, if the program is large, or if any changes must be made in the code preceding the BALR, it becomes difficult to know whether the NØPR should be used or not.

To relieve the programmer of this worry, the Assembler provides an instruction CNØP (Conditional No-Operation) which ensures the desired alignment. The operand field entry of a CNØP instruction is written b,w where b and w are absolute expressions; b may have values 0, 2, 4 and 6, and w may have values 4 and 8. <u>No</u> name field entry is permitted. The second operand, w, specifies the boundary type relative to which alignment is to be performed, and b specifies the desired byte relative to that boundary, as described in Figure 15.2. The Assembler inserts from 0 to 3 NØPR's to force the LC to the desired boundary.

Instruction	Alignment Performed
CNØP 0,4	Beginning of a fullword
CNØP 2,4	Middle of a fullword
CNØP 0,8	Beginning of a doubleword
CNØP 2,8	Second half'word of a doubleword
CNØP 4,8	Middle of a doubleword
CNØP 6,8	Fourth halfword of a doubleword

Figure 15.2 CNØP Alignments

To achieve the alignment desired in the current example, we would write

```
     CNØP    2,4            ALIGN TØ MIDDLE ØF WØRD
     BALR    14,15          TWØ-BYTE INSTRUCTIØN
     DC      A(ANYTHING)    NØ INTERVENING BYTES
```

Note that we could not write

```
     DS      OH
     BALR    14,15
     DC      A(ANYTHING)
```

because the alignment to a half'word boundary forced by the DS is automatically performed by the Assembler for instructions, so that the BALR could still fall on a fullword boundary; the Assembler would then fill the two bytes between the BALR and the address constant with zeros (remember that A-type constants have an implied fullword alignment). Similarly, we could not write

```
        BALR    14,15
        DS      0F
        DC      A(ANYTHING)
```

since the BALR could again fall on a fullword boundary, leaving two bytes between it and the constant which would be skipped by the Assembler; the contents of the skipped bytes at execution time may be arbitrary, since the Supervisor does not clear the area into which a program is about to be loaded,

Before continuing with our discussion of arithmetic instructions, one important feature of the use of branch instructions should be noted. Due to a peculiarity in the design of System/360, invalid branch addresses (namely odd ones) are not detected at the time that it is found that the branching condition is met, but only when the address is presented, as the IA portion of the PSW, at the <u>next</u> instruction fetch cycle. The error is duly detected and a specification interruption results, but the IA now contains the invalid address rather than the address of the instruction which attempted the illegal branch. This means that there is no direct way to tell where such an error was caused, and therefore that such errors in a program are correspondingly more difficult to detect. The programmer must exercise caution in specifying branch addresses in order to avoid this particular error.

	0	**1**	2	**3**	**4**	5	**6**	**7**	8	9	A	**B**	C	**D**	**E**	**F**
										9						
00*	**0**	1	2	**3**	**4**	**5**	**6**	**7**	8	**25**	**10**	**11**	**12**	**13**	**14**	**15**
01*	**16**	**17**	**18**	19	**20**	**21**	**22**	23	24		**26**	27	**28**	29	**30**	31
02*	32	33	34	35	36	37	**38**	39	**40**	41	**42**	'43	**44**	**45**	**46**	47
03*	**48**	49	**50**	51	**52**	**53**	54	**55**	**56**	**57**	**58**	s9	**60**	61	**62**	63
04*	64	**65**	66	67	6%	69	**70**	71	72	73	74	75	76	77	78	79
05*	**80**	81	82	83	84	a5	a6	**87**	**88**	**89**	90	91	92	93	94	**95**
06*	96	91	98	99	100	**101**	**102**	103	104	**105**	106	107	**108**	109	**110**	**111**
07*	112	113	114	115	**116**	**117**	it8	119	120	**121**	122	123	124	125	126	127
08*	128	**129**	130	131	132	**133**	134	135	136	137	138	139	140	**141**	142	**143**
09*	144	145	144	147	148	**149**	150	151	152	153	154	**155**	156	157	**158**	**159**
0A*	160	161	162	163	164	165	166	167	**168**	**169**	**170**	171	**172**	173	**174**	**175**
0B*	**176**	177	**178**	**179**	180	**181**	A82	**183**	184	185	186	**187**	188	**189**	190	191
0C*	192	193	194	195	196	197	198	199	200	**201**	**202**	203	204	205	206	207
0D*	20%	209	210	**211**	212	213	214	**215**	216	**217**	218	219	220	221	222	223
0E*	224	225	226	227	**228**	229	230	231	232	233	234	235	236	237	238	239
0F*	240	**241**	242	243	244	245	246	247	240	249	250	251	252	253	254	255
10*	256	257	250	259	**260**	**261**	262	263	264	265	266	267	268	269	270	271
11*	272	273	274	275	276	277	**278**	279	280	**281**	282	283	284	285	286	**287**
12*	280	**289**	290	291	292	293	294	295	296	297	298	299	300	301	302	303
13*	304	305	306	307	308	309	3io	31 l	312	313	314	315	316	317	318	319
14*	**320**	321	322	323	324	325	326	327	320	329	330	331	332	333	334	**335**
15*	336	337	338	**339**	340	341	342	343	344	345	346	347	348	349	350	**351**
16*	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	**367**
17*	368	**369**	**370**	371	372	373	374	375	376	377	**378**	379	380	**381**	**382**	383
18*	384	38'5	306	387	388	**389**	**390**	391	392	393	394	**395**	**396**	397	**398**	**399**
19*	**400**	401	402	403	404	405	406	407	408	**409**	**410**	err	422	**413**	**414**	**415**
1A*	416	**417**	**418**	**419**	**420**	**421**	**422**	423	424	425	**426**	-- **427**	428	429	**430**	**431**
1B*	432	433	434	435	436	**437**	438	439	440	44-r	842	443	**444**	445	**446**	**447**
1C*	**448**	449	450	**451**	452	**453**	454	455	456	457	458	459	460	461	**462**	**463**
1D*	464	465	466	467	460	**469**	470	**471**	472	473	**474**	475	**476**	477	478	**479**
1E*	**480**	**481**	482	483	**484**	**485**	**486**	487	488	489	**490**	491	492	493	494	**495**
1F*	**496**	**497**	**498**	**499**	**500**	**501**	**502**	**503**	**504**	**505**	**506**	**507**	**508**	so9	**510**	**511**

	C	1	2	3	4	5	6	7	8	9	A	5	C	D	E	F
20*	512	513	314	515	516	517	518	519	520	521	522	523	524	525	526	527
21*	528	529	530	531	532	533	534	535	536	53t	538	539	540	541	342	543
22*	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559
23*	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575
24*	576	577	578	579	580	581	582	583	584	585	586	587	S88	589	590	591
25*	592	593	594	593	596	597	598	599	600	601	602	603	604	605	606	607
26*	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623
27*	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639
28*	64C	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655
29*	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671
2A*	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687
28*	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	to3
2C*	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719
20*	72C	721	722	723	724	725	726	727	728	729	730	731	732	733	734	7 3 3
2E*	736	t37	730	739	740	741	742	743	744	t45	746	747	748	749	750	751
2F*	752	753	t54	755	756	757	758	759	760	761	762	763	764	765	766	767
30*	768	769	77C	771	772	773	774	775	776	777	778	779	780	781	782	703
31*	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799
32*	80C	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815
33*	816	817	818	819	820	821	822	823	824	825	826	82t	828	829	830	831
34*	832	833	834	835	836	837	8 3 8	839	840	841	842	843	844	845	846	847
35*	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863
36*	864	865	866	867	868	869	870	871	872	873	874	8715	a76	877	878	879
37*	88C	881	882	883	884	885	886	887	888	889	890	891	892	893	a94	895
38*	896	897	898	899	900	901	902	903	904	905	906	907	908	9C9	910	911
39*	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927
3A*	928	S29	930	931	932	933	934	935	936	937	938	939	940	941	942	943
3B*	944	S45	946	947	948	949	950	951	952	953	954	955	956	957	958	959
3C*	S6C	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975
30*	S76	S77	97s	979	980	981	982	983	984	985	986	987	988	989	990	991
3E*	992	993	994	995	996	997	998	999	1000	3001	1002	1003	1004	1CC5	1006	1007
3F*	1008	1C09	101C	1011	1012	1313	1014	1015	1016	1017	to18	1019	1020	1021	i022	1023

A-3

	0	1	2	3	4	5	6	7	8	9	A	5	c	D	E	F
40*	**1024**	**1025**	**1026**	1027	1028	1029	1030	1031	1032	1033	1034	1035	*1036*	**1037**	1038	1039
41*	**1040**	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42*	**1056**	1057	**1058**	1059	1060	**1061**	**1062**	**1063**	*1064*	1065	1046	*1067*	1068	1069	1070	**1071**
43*	*1072*	1073	'1074	1075	1076	**1077**	**1078**	*1079*	1080	**1081**	1082	1083	1084	1085	1086	1007
44*	**1088**	**1089**	1090	1091	1092	**1093**	1094	1095	1096	1097	1098	1099	1100	1101	**1102**	1103
45*	1104	1105	1106	1107	1108	1109	**1110**	1111	1112	1113	1114	1115	1116	**1117**	**1118**	1119
46*	**1120**	1121	1122	1123	1124	1125	**1126**	1127	1128	1129	1130	1131	**1132**	1133	1134	1135
47*	1136	1137	**1138**	1139	1140	**1141**	1142	1143	1144	1145	1146	**1147**	1148	1149	1150	1151
48*	1152	1153	1154	1155	**1156**	**1157**	**1158**	**1159**	**1160**	1161	1162	1163	1164	1165	**1166**	**1167**
49*	**1168**	1169	1170	1171	1172	1173	**1174**	**1175**	**1176**	1177	1178	1 **179**	**1180**	**1181**	1182	1183
4A*	1184	**1185**	**1186**	1187	1188	**1189**	**1190**	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B*	1200	1201	1202	1203	1204	1205	**1206**	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C*	1216	**1217**	**1218**	1219	1220	**1221**	1222	**1223**	1224	1225	1226	1227	1228	1229	1230	**1231**
4D*	1232	1233	1234	1235	**1236**	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E*	1248	1249	*1250*	1251	1252	1253	**1254**	1255	**1256**	1257	**1258**	1259	1260	1261	**1262**	1263
4F*	1264	1265	1266	*1267*	*1268*	**1269**	**1270**	1271	1272	1273	**1274**	1275	1276	1277	1278	1279
50*	**1280**	**1281**	**1282**	1283	1284	1285	**1286**	1287	**1288**	**1289**	**1290**	1291	**1292**	**1293**	1294	*1295*
51*	1296	1297	1298	1299	1300	**1301**	**1302**	1303	1304	1305	**1306**	**1307**	1308	1309	**1310**	1311
52*	**1312**	**1313**	**1314**	1315	1316	**1317**	**1318**	1319	1320	1321	**1322**	1323	**1324**	**1325**	1326	1327
53*	**1328**	1329	**1330**	1331	1332	1333	**1334**	1335	*1336*	**1337**	**1338**	**1339**	**1340**	**1341**	1342	**1343**
54*	**1344**	**1345**	*1346*	*1347*	*1348*	**1349**	**1350**	**1351**	**1352**	1353	1354	**1355**	1356	1357	1358	**1359**
55*	*1360*	**1361**	*1362*	**1363**	1364	**1365**	**1366**	1367	1368	1369	1370	**1371**	1372	**1373**	1374	**1375**
56*	**1376**	**1377**	1378	1379	**1380**	**1381**	**1382**	1383	1384	1385	1386	**1387**	1388	1389	1390	1391
57*	**1392**	*1393*	**1394**	1395	1396	**1397**	**1398**	1399	1400	1401	1402	**1403**	1404	1405	**1406**	1407
58*	**1408**	**1409**	**1410**	**1411**	**1412**	**1413**	**1414**	1415	**1416**	1417	**1418**	1419	**1420**	**1421**	**1422**	**1423**
59*	1424	**1425**	**1426**	**1427**	**1428**	**1429**	**1430**	1431	1432	1433	**1434**	**1435**	**1436**	**1437**	1438	1439
5A*	**1440**	**1441**	**1442**	**1443**	**1444**	**1445**	**1446**	**1447**	**1448**	1449	**1450**	**1451**	**1452**	**1453**	1454	**1455**
5B*	**1456**	**1457**	**1458**	**1459**	**1460**	**1461**	**1462**	**1463**	**1464**	**1465**	**1466**	**1467**	**1468**	1469	1470	**1471**
5C*	1472	**1473**	**1474**	**1475**	**1476**	**1477**	**1478**	1479	1480	**1481**	**1482**	**1483**	1484	**1485**	**1486**	**1487**
5D*	**1488**	**1489**	1 4 9 0	**1491**	1492	1493	**1494**	1495	1496	*1497*	**1498**	1499	**1500**	1501	**1502**	**1503**
5E*	**1504**	1505	1506	**1507**	1508	1509	**1510**	**1511**	**1512**	**1513**	**1514**	1515	**1516**	**1517**	1518	1519
5F*	**1520**	1521	1522	1523	1524	1525	**1526**	**1527**	**1528**	**1529**	**1530**	**1531**	1532	**1533**	**1534**	1535

	C	1	2	3	4	5	6	7	8	9	A	5	C	D	E	F
60*	1536	1537	1538	1539	1540	1541	1542	1543'	1544	1545	1546	1547	1548	1549	1550	1551
61*	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62*	1568	1369	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63*	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64*	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	16 13	1614	1615
65*	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66*	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67*	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68*	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69*	1680	1681	1682	1683	1684	1685	1686	1687	1638	1689	1690	1691	i692	1693	1694	1695
6A*	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B*	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C*	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D*	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E*	1760	1761	1762	1764	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F*	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70*	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71*	1800	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	I.821	1822	1823
72*	1824	1825	1026	1827	1828	1829	1830	1831	1832	1833	1834	1835	1036	1837	1838	1839
73*	1840	1841	1842	1843	1844	i d45	1846	1047	1848	1849	1850	1851	1852	1853	1854	1855
74*	1856	1857	1058	1859	1860	1861	A862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75*	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76*	1888	1889	2890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77*	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	191s	1916	1917	1918	1919
78*	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79*	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A*	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B*	1968	1969.	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C*	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D*	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	20 13	2014	2015
7E*	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F*	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

A5

	G	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80*	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81*	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82*	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83*	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84*	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85*	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86*	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87*	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88*	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89*	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A*	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B*	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C*	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D*	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E*	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F*	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90*	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91*	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92*	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93*	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94*	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95*	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96*	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97*	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98*	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99*	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A*	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B*	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C*	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D*	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E*	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F*	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

	G	1	2	3	4	5	6	7	8	9	A	B	c	D	E	F
A0*	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
Al*	2516	2577	2570	2579	2580	2581	2582	2583	2584	2585	2586	2587	250%	2589	2590	2591
A2*	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3*	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	261%	2619	2620	2621	2622	2623
A4*	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5*	2644	2641	2642	2643	2644	2645	2646	2647	264%	2649	2650	2651	2652	2653	2654	2655
A6*	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7*	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8*	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	269%	2699	2700	2701	2702	2703
A9*	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	27 14	2715	2716	27 17	2718	2719
AA*	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB*	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC*	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2 763	2764	2765	2766	2767
AD*	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE*	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF*	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	28 13	2814	2815
B0*	2816	2817	2818	2819	2820	2821	2822	2823	2824	2025	2826	2827	282%	2829	2830	2831
B1*	2832	2833	2834	2 835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2046	2047
B2*	284%	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2063
B3*	2864	2865	2866	2867	286%	2869	2870	2871	2872	2873	2874	2875	2876	2877	207%	2879
B4*	2880	2881	2882	2883	2884	2885	2806	2887	2888	2889	2890	2891	2892	2893	2094	2095
B5*	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6*	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7*	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2930	2939	2940	2941	2942	2943
B8*	2944	2945	2946	2947	294%	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9*	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA*	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2906	'2907	2988	2989	2990	2991
BB*	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC*	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD*	3024	3025	3026	3027	302%	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE*	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF*	3056	3037	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

	0	1	2	3	4	5	6	7	8	9	A	B	c	D	E	F
co*	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1*	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2*	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3*	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4*	3134	3137	3138	3139	3140	3141	3142	3143	'3144	3145	3146	3147	3148	3149	3150	3151
C5*	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6*	3168	3149	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7*	3184	3185	3186	3187	3188	3169	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8*	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	32 10	3211	3212	3213	3214	3215
C9*	3216	3217	3218	3219	3220	322 1	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA*	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB*	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
cc*	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD*	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE*	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF*	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0*	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1*	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2*	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370,	3371	3372	3373	3374	3375
D3*	3376	3377	3378	3379	3380	3381	3382	3383	33 84	3385	3386	3387	3388	3389	3390	3391
D4*	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5*	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6*	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7*	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8*	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9*	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA*	3488	3489	3490	349 1	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB*	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
CC*	3520	3521	3522	3523	3524	3525	3526	352.7	3528	3529	3530	3531	'3532	3533	3534	3535
DD*	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE*	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3547
DF*	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583

	0	1	2	3	4	5	6	7	8	9	A	B	c	D	E	F
E0*	3584	3585	3584	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1*	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	36 10	3611	3612	3613	3614	3615
E2*	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3*	3632	3633,	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4*	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5*	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6*	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7*	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8*	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9*	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA*	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB*	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC*	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED*	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE*	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF*	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0*	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1*	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2*	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3*	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4*	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5*	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6*	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7*	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8*	3968	3969	3979	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9*	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA*	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	40 10	4011	4012	40 13	4014	4015
FB*	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	402 8	4029	4030	4031
FC*	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD*	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE*	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF*	408 0	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

16. FIXED-POINT ARITHMETIC INSTRUCTIONS

In this section we will discuss the instructions which perform fixed-point two's complement arithmetic in the general purpose registers; the relevant instructions are tabulated in Figure 16.1.

Mnemonic	Type	Instruction
AR	RR	Add Register
A	RX	Add
ALR	RR	Add Logical Register
AL	RX	Add Logical
AH	Rx	Add Half'word
SR	RR	Subtract Register
CR	RR	Compare Register
S	Rx	Subtract
C	Rx	Compare
SLR	RR	Subtract Logical Register
CLR	RR	Compare Logical Register .
SL	RX	Subtract Logical
CL	Rx	Compare Logical
SH	RX	Subtract Halfword
CH	RX	Compare Halfword
MR	RR	Multiply Register
M	Rx	Multiply
MH	RX	Multiply Halfword
DR	RR	Divide Register
D	RX	Divide

Figure 16.1 Fixed-Point Arithmetic Instructions

There are severel instructions missing from the table which one might expect to find: there are no logical halfword instructions, there is no "Divide Halfword", and there are no instructions for performing multiplication and division with logical operands. It is possible, however, to compute logical products and quotients using available instructions.

The operations of the add and subtract instructions are straightforward and are summarized in Figure 16.2 below. Remember that the logical add and subtract produce the same result as the arithmetic add and subtract instructions except that the CC is set differently. For the halfword operations, we may assume (as in the discussion of LH in Section 14) that the second operand is brought from memory to the MDR, extended to a fullword, and then used for the indicated operation. The notation "FW_2" means the fullword operand at the effective memory address in the RX instructions, and "HW_2" means the same for halfword operands.

Instruction	Action	CC Settings
AR SR A S AH SH'	$C(Rr_1) \leftarrow C(Rr_1)+C(Rr_2)$ $C(Rr_1) \leftarrow C(Rr_1)-C(Rr_2)$ $C(Rr_1) \leftarrow C(Rr_1)+C(FW_2)$ $C(Rr_1) \leftarrow C(Rr_1)-C(FW_2)$ $C(Rr_1) \leftarrow C(Rr_1)+C(HW_2)$ $C(Rr_1) \leftarrow C(Rr_1)-C(HW_2)$	0: Result is zero 1: Result is< 0 2: Result is> 0 3: Overflow
ALR SLR AL SL	$C(Rr_1) \leftarrow C(Rr_1)+C(Rr_2)$ $C(Rr_1) \leftarrow C(Rr_1)-C(Rr_2)$ $C(Rr_1) \leftarrow C(Rr_1)+C(FW_2)$ $C(Rr_1) \leftarrow C(Rr_1)-C(FW_2)$	0: Zero result, no carry 1: Nonzero result, no carry 2: Zero result, carry 3: Nonzero result, carry

Figure 16.2 Fixed-Point Add and Subtract Instructions

The CC settings in the rightmost column apply to all the instructions in the same part of the table. It is useful to note several aspects of the CC settings for the logical instructions, which depend on whether a carry occurs out of the leftmost position of Rr_1, and whether the result is zero. By referring to the examples in Section 7, we can see that

(1) a CC setting of zero is possible for AL and ALR only if both the first and second operands are zero.

(2) it is not possible to have a CC setting of zero for SL and SLR, because after the one's complement of the second operand and a low-order 1 bit are added to the first operand, a carry must have occurred if the result is zero.

Suppose we wish to store at ANS the sum of C(X) and C(Y), unless the result is negative, in which case we must also add C(Z) and subtract 2: the instruction sequence

```
         L     5,X
         A     5,Y              C(R5) = C(X) + C(Y)
         BC    11,ST            BRANCH If NOT NEGATIVE
         A     5,Z              ADD C(Z)
         SH    5,=H'2'          SUBTRACT 2
ST       ST    5,ANS            STORE- ANSWER
```

will calculate the required quantity. Note that ST is used both as a symbol and as an instruction mnemonic; no confusion is possible, since the Assembler identifies the instruction only by its appearance as an operation field entry.

Suppose we want to compute the sum of the first n odd numbers, where the positive integer n is stored as a halfword integer at N;. consider the following instruction sequence.

```
         LH    3,N              GET N
         LM    6,9,=F'0,2,1,1'  LOAD R6-R9 WITH 0,2,1,1
ADDUP    AR    6,8              ADD ODD INTEGER TO SUM IN R6
         AR    8,7              NEXT ODD INTEGER IN R8
         SR    3,9              DECREASE N BY 1
         BC    7,ADDUP          BRANCH N-I TIMES
         ST    6,SUM            STORE RESULT
```

One feature of this example is that all calculations inside the loop (third through sixth instructions) are done using RR instructions; this technique is occasionally useful in programs where processing speed is important, and enough registers are available to allow all operands to be carried there instead of in memory. The example is of course mathematically nonsensical because we have expended all this effort to calculate n^2 where a multiply instruction would have sufficed.

To give another simple example of the use of some of these instructions, suppose we wish to canpute NEWSTØCK from the formula

NEWSTØCK = ØLDSTØCK + RECEIPTS - SALES

where all quantities are fullword integers small enough to guarantee that no overflows occur. Both sets of statements below compute the desired result.

```
L      2,OLDSTOCK                          L      2,OLDSTOCK
A      2,RECEIPTS                          AL     2,RECEIPTS
S      2,SALES                             SL     2,SALES
ST     2,NEWSTOCK                          ST     2,NEWSTOCK
```

The compare instructions are useful in testing the relative magnitudes of two operands; the results of the comparison are indicated in the CC setting as shown in Figure 16.3.

Operations	CC Settings
CR	
C	0: Operand 1 = Operand 2
CH	1: Operand 1 < Operand 2
CLR	2: Operand 1 > Operand 2
C L	

Figure 16.3 CC Settings for Compare Instructions

The CC cannot be set to 3 as a result of a compare instruction. It can be seen for the CR, C, and CH instructions that the CC setting is the same as would result from performing SR, S, and SH instructions with the same operands, assuming that no overflow occurs. In fact, this is how the comparison is done by the CPU -- a subtraction is performed internally and the CC is set to reflect the sign and the magnitude of the difference, which would have been placed back in R_{r_1} for the subtract instructions. Further analysis of the original operands is required in the CPU if the internal result overflows. The logical-comparisons do <u>not</u> give the same results as arithmetic comparisons, since numbers in the logical representation are always considered to be positive. The following instruction sequence may help to illustrate the differences.

```
LM      0,3,=F'1,0,-1,-2147483647'
CR      1,3               CC = 2
CLR     1,3               CC = 1
CR      0,2               CC = 2
CLR     0,2               CC = 1
CR      2,3               CC = 2
CLR     2,3               CC = 2
LPR     4,3               C(R4) = X'7FFFFFFF', CC = 2
CR      4,3               CC = 2
CLR     4,3               CC = 1
C       0,=F'1'           CC = 0
CL      2,=F'+2'          CC = 2
CH      1,=H'5'           CC = 1
CH      1,=F'5'           CC = 0
```

The last of the statements in the above example is a programming error that occasionally occurs; note that the Assembler gives no indication of the conflicting data types implied by the instruction and the operand.

As an example of the use of a **compare** instruction, let us recalculate the sum of the first n odd integers using a different scheme than before.

```
        LH    4,=H'1'       C(R4) = ACCUMULATED SUM
        LR    7,4           C(R7) = COUNT OF ADDITIONS
TEST    CH    7,N           COMPARE COUNT TO C(N)
        BC    8,STORE       BRANCH IF N TERMS ADDED
        LR    0,7           COMPUTE NEXT ODD INTEGER
        AR    0,0           COUNT + COUNT
        AH    0,=H'1'       ADD 1, GIVING NEXT ODD TERM
        AR    4,0           ADD TERM TO SUM
        AH    7,=H'1'       INCREMENT COUNT BY 1
        BC    15,TEST       BRANCH TO SEE IF FINISHED
STORE   ST    4,SUM         STORE RESULT
```

This example is rather cumbersome, but yields the desired result; we will see that there are many ways to perform-the same computation with varying degrees of elegance. It is worth noting that programming is often as much an art as a science, since many different programs of varying effectiveness can be written to achieve a given objective; an important part of learning to program is understanding where efficiency can be increased.

As another example, suppose we wish to force the value of the integer in R6 to be a multiple of 8, in such a way that if it is not already so, the next higher multiple of 8 will be chosen. This would be required of the relative origin assigned to a program: the Assembler chooses the next higher multiple of 8 if the programmer assigns a relative origin which is not already a multiple of 8. Consider the following segment of code.

```
         SR     7,7             CLEAR R7
         SRDL   6,3             SHIFT 3 BITS INTO R7
         LTR    7,7             SEE IF THE BITS ARE ZERO
         BC     8,A             BRANCH IF YES
         A      6,=F'1'         IF NOT, ADD 1 TO R6
A        SLL    6,3             MULTIPLY BY 8
```

First, note that we have cleared R7 by subtracting it from itself -- this is the fastest and simplest way to do so and will be used generally except in situations where the condition code must not be set. In such circumstances, an instruction such as L 7,=F'0' might be used, though there are other ways which are sometimes more efficient. Second, we can use a shift instruction to divide by 8, and since a double-length shift is used, the "remainder" bits shifted into the three high-order bit positions of R7 are not lost, which would be the case of SRL 6,3 had been used. The BC instruction branches only if the remainder bits are all zero -- that is, if the number in R6 was already a multiple of 8. The same 'calculation can be done more simply:

```
         A      7,=F'7'     FØRCE CARRY IF PØSSIBLE
         SRL    7,3         DRØP ØFF 3 BITS
         SLL    6,3         MULTIPLY BY 8
```

where in this case the presence of any 1 bit in the three rightmost bit positions of the original number cause a carry into the 2^3 bit position (that is, bit 28 of R6); the result is the same as before except for the final CC setting.

To illustrate the use of logical arithmetic, suppose we are required to perform additions and subtractions on 8-byte integers: double-length integers too large to fit in a single fullword. Such operations are infrequently required, but an examination of the methods used provides insight into the properties of some of the pertinent instructions. Double-length integers will occasionally be encountered as products and dividends. Consider first the problem of finding the two's complement of such a number. Since we know that the two's complement can be found by adding a low-order 1 bit to the one's complement of the number, we might proceed as in the following example, where the number to be complemented is stored beginning at ARG. By C(RO,R1) we mean the contents of the double-length register formed by RO and R1.

```
          L      0,=F'-1'
          LR     1,0            C(RO,R1) IS ALL 1 BITS
          S      0,ARG          1'S COMPLEMENT OF HIGH-ORDER PART
          S      1,ARG+4        1'S COMPLEMENT OF LOW-ORDER PART
          AL     1,=F'1'        ADD LOW-ORDER 1 BIT
          BC     12,NC          BRANCH IF NO CARRY
          A      0,=F'1'        ADD CARRY BIT TO RO
NC        STM    0,1,ARG        STORE' COMPLEMENTED RESULT
          - - -
          DS     0D             ALIGN ON DOUBLEWORD BOUNDARY
ARG       CC     FL8'1234567876543211'
```

The AL instruction in the fifth statement must be used rather than A because the high-order bit of R1 is not a sign bit, but an arithmetically significant bit with weight 2^{31}; if a carry out'of R1 occurs, it must be detected and propagated into the low-order bit of RO, since there is no --provision for having this done automatically. The same calculation is performed by the following code sequence, but in a less direct and obvious way.

```
          LM     0,1,ARG        GET DOUBLE-LENGTH OPERAND
          LCR    0,0            COMPLEMENT HIGH-ORDER WORD
          LCR    1,1            COMPLEMENT LOW-CRDER WORD
          BC     8,X            JUMP IF C(R1) = 0
          S      0,=F'1'        SUBTRACT 1 FROM RO
X         STM    0,1,ARG        STGRE RESULT
          - - -
          DS     0D             ALIGN
ARG     - DC     FL8'98765434567890'
```

In this case, we use the first LCR instruction to form the two's complement of C(RO) immediately; that is, we have already added a low-order 1 bit to the one's complement of C(RO). The following LCR complements the low-order 32 bits and sets the CC. Now if C(R1) had been zero, its one's complement would be all 1 bits, and adding a low-order 1 bit would cause a carry out the left end of R1. For any other bit pattern, no such carry would have occurred, and we must correct C(RO) by subtracting 'off the low-order bit added during the execution of the first LCR.

At this point it should be evident what we must do to add two double-length integers; we will simply write a code sequence without further explanation.

```
         LM     0,1,A          GET A
         AL     1,B+4          ADD LOW ORDER PARTS
         BC     12,NC          BRANCH IF NO CARRY
         A      0,=F'1'        PROPAGATE CARRY BIT TO HIGH-ORDER PART
NC       A      0,B            ADD HIGH-ORDER PARTS
         STM    0,1,C          STORE DOUBLE-LENGTH SUM
         - - -
C        DS     D              RESERVE 8 BYTES, ALIGNED
B        DC     FL8'222333444555'
A        DC     FL8'888777666555'
```

Subtraction is performed in the same way, except that the condition code setting after the first subtraction will require explanation.

```
         LM     0,1,A          GET FIRST OPERAND
         SL     1,B+4          SUBTRACT LOW-ORDER PART OF SECOND OPERAND
         BC     3,CAR          BRANCH IF THERE'S A CARRY
         S      0,=F'1'        REDUCE C(R0) BY 1 (BORROW 1)
CAR      S      0,B            SUBTRACT HIGH-ORDER PART OF SECOND OPERAND
         STM    0,1,C          STORE DOUBLE-LENGTH DIFFERENCE
         - - -
C        DS     D
B        DC     FL8'123456787654321'
A        DC     FL8'234567898765432'
```

In performing a subtraction, the one's complement of the second operand and a low-order 1 bit are added to the first operand. If a carry occurs out of the high-order bit position, then the result is correctly represented; if a carry does not occur, then the resultcannot be correctly represented, in the sense that we have tried to generate a "negative" integer in the logical representation. Hence we must "borrow" a 1 bit from the next highest bit position, which accounts for the subtraction of F'1' if the branch condition is not met. It may be helpful to review the examples in Section 7 to clarify the cases of "overflow" in the logical representation.

Multiplication and division work essentially in the manner described in Section 8. Except for MH, a double-length register is required for product and dividend, and the various operands are placed in the expected registers before and after the operation.

For the multiplication instructions MR and M, the r_1 digit must be even; as was the case for the double-length shift instructions, the even-numbered register is the high-order half of an even-odd register pair, with the next higher odd-numbered register being the low-order half. The multiplicand is placed in the odd-numbered register, and the multiplier is the second operand. The product replaces the original contents of the pair of registers. Thus, the following instructions will produce the indicated results.

```
MR     2,7              C(R2,R3) = C(R3)*C(R7)
MR     0,1              C(R0,R1) = C(R1)*C(R1)
MR     8,8              C(R8,R9) = C(R8)*C(R9)
M      4,X              C(R4,R5) = C(R5)*C(X)
M      12,=F'932'       C(R12,R13) = C(R13)*932
LR     5,4              MOVE MULTIPLICAND TO R5
MR     4,4              C(R4,R5) = C(R5)*C(R4)
```

The last two instructions illustrate a situation where we wish to square the integer in R4 -- the LR is required to place the operand into the odd-numbered register; note that we could have used MR 4,5 also, giving C(R5)*C(R5). The presence of the multiplier in the even-numbered register does not cause it to be lost when that register is cleared at the beginning of the multiply sequence, since the multiplier must be moved internally to a separate register in the CPU; we can visualize the multiplication taking place after the multiplier has been moved to the MDR.

It is important to remember that the product generated by the M and MR instructions is 64 bits long. If we were to perform the following sequence of instructions (note that $65536 = 2^{16}$)

```
         L      1,=A(X'10000')      C(R1) = 65536
         MR     0,1                 SQUARE IT
         ST     1,PRODUCT
         - - -
 PRODUCT DS     F
```

we would find that the fullword stored at PRØDUCT was zero and that C(R0) = 1; and if we executed the instruction sequence (note that $32768 = 2^{15}$)

```
L       1,=A(X'10000')      C(R1) =   65536
M       0,=A(X'8000')MULTIPLY BY 32768
ST      1,PRODUCT
```

we would find that C(PRØDUCT) = -2^{31}. There are thus two situations the programmer should be aware of: first,that the size of the product may be such that it overflows the low-order register, and second, that whether or not the high-order register contains significant bits, the leftmost bit of the low-order register is <u>not</u> a sign bit, but contains an arithmetically significant digit.

The MH instruction produces a single-length result, which is the low-order 32 bits of the product of C(Rr_1) and the half'word'second operand. Because only a fullword result is retained, r_1 need not be even, and a specification exception will occur only if the effective address of the halfword operand is odd. Because fewer shifts and adds are needed during multiplication, some small economies may be achieved by the use of MH, particularly on the smaller models of System/360. Thus, MH 5,=H'100' is a simple way to multiply the contents of R5 by 100. If X and Y are both halfword operands, their product may be found by writing

```
LH    9,X
MH    9,Y
```

and R8 is undisturbed. And to square the halfword integer n at N we could write

```
LH    6,N
MH    6,N
```

Note that because both operands are halfwords of at most 15 significant bits, the product will fit in a single register; the only halfword whose magnitude requires 16 bits (namely -2^{15}) when squared yields 2^{30}, which requires only 31 bits. We note in passing that none of the multiply instructions affect the condition code.

-As an example of the use of a multiply instruction, suppose we want to calculate A = B + G * D, where all quantities are fullword integers, and it is assumed that all results are small enough so that no overflows occur.

```
         L      7,G                C(R7) = C(G)
         M      6,D                C(R6,R7) = G*D
         A      7,B                C(R7) = B+G*D
         ST     7,A                STORE RESULT
```

Note that we have used the letters A, B, G, and D to denote both the names of fullword areas of memory and the names of the contents of these areas; this usage is typical of procedural languages, where little distinction is made between the name associated with an area of memory, the contents of the area, and the value associated with the contents. We will explore such considerations further after more data representations have been discussed.

. As a second example of the use of multiply instructions, suppose we wish to compute the sum of the cubes of the first n integers, where n is stored in the fullword at NBR. We will assume that n is a small enough positive integer that the sum is representable in a single fullword. The quantity k will be the index in the sum

$$\sum_{k=1}^{n} k^3.$$

```
         SR     5,5                SUM CARRIED IN R5
         L      4,=F'1'            K CARRIED IN R4
RPT      LR     1,4                C(R1) = K
         MR     0,1                C(R0,R1) = K*K
         MR     0,4                C(R0,R1) = K CUBED
         AR     5,1                ACCUMULATE SUM
         A      4,=F'1'            INCREMENT K
         C      4,NBR              COMPARE TO UPPER LIMIT
         BC     12,RPT             BRANCH IF K NOT BIGGER
         ST     5,SUM              STORE SUM OF CURES
```

A slightly different version'of the same program which counts from n down to 1 follows.

```
         SR     5,5                INITIALIZE SUM TO ZERO
         L      6,=F'1'            C(R6) = 1, USED AS CONSTANT
         L      4,NBR              INITIALIZE K TO C(NBR) = N
RPT      LR     1,4                C(R1) = K
         MR     0,4                C(R0,R1) = K*K
         MR     0,4                C(R0,R1) = K CUBED
         AR     5,1                ADD TO SUM
         SR     4,6                DECREMENT K BY 1
         BC     2,RPT              RRANCH IF K STILL POSITIVE
         ST     5,SUM              STORE RESULT
```

Division is always performed using a double-length dividend and remainder. As was the case for the fullword multiply instructions, the r_1 digit must be even, and specifies the register pair containing the?. dividend; the CC is unaffected. As indicated in Section 8, the quotient replaces the iow-order half of the dividend in the odd-numbered register, and the remainder replaces the high-order part of the dividend in the even-numbered register; If a , valid quotient cannot be computed, a fixed-point divide exception occurs. For example, to divide the double-length number in (R8,R9) by the number in R13, we can write DR 8,13 and to divide the same number by 10 we could write D 8,=F'10' . To illustrate the use of a divide instruction, suppose we want to compute the product of C(A) and C(B), and force the result to the next largest multiple of 29 if it is not already a multiple. We will assume that the product is small enough-that a fixed-point divide exception will not occur when dividing by 29, and that the final result is contained in a single fullword.

```
         L      3,A             C(R3) = C(A)
         M      2,B             C(R2,R3) = C(A)*C(B)
         D      2,=F'29'        QUOTIENT IN R3
         LTR    2,2             TEST REMAINDER IN R2
         BC     8,MPY           BRANCH IF C(R2) IS ZERO
         A      3,=F'1'         INCREASE QUOTIENT BY 1
MPY      M      2,=F'29'        FORM CORRECT MULTIPLE OF 29
         ST     3,RESULT        STORE PROPER RESULT
```

As a final example of division, suppose there is a positive integer at N which we want to divide by 10, and then store a <u>rounded</u> quotient at Q. This means that if the remainder is 5 or larger the quotient must be increased by 1.

```
         L      7,N             GET N AS LOW-ORDER PART OF DIVIDEND
         SR     6,6             CLEAR HIGH-ORDER PART OF DIVIDEND
         D      6,=F'10'        DIVIDE BY 10
         C      6,=F'5'         COMPARE REMAINDER TO 5
         BC     4,OKAY          BRANCH IF SMALLER THAN 5
         A      7,=F'1'         OTHERWISE ROUND UP
OKAY     ST     7,Q             STORE ROUNDED RESULT
```

Suppose now that the integer at N might be negative; it is apparent that the instruction sequence above will not work correctly, for two reasons.

First, the initial value of the dividend would not have a correctly extended sign bit for negative arguments; second, because the sign of the remainder is always the same as the sign of the original dividend, the compare instruction would always (when C(N) is negative) cause the following branch instruction to transfer control to ØKAY independent of the magnitude of the remainder. To obtain a correctly represented dividend it is simplest to use the SRDA instruction, as shown.

```
         L       1,=F'1'         SET UP ROUNDING BIT
         L       6,N             C(R6) = C(N)
         SRDA    6,32            C(R6,R7) = 64-BIT DIVIDEND
         BC      11,DIV          JUMP IF NON-NEGATIVE DIVIDEND
         LCR     1,1             OTHERWISE SET ROUNDOFF TO -1
DIV      D       6,=F'10'        DIVIDE BY 10
         LPR     6,6             ABSOLUTE VALUE OF REMAINDER
         C       6,=F'5'         COMPARE TO 5
         BC      4,OKAY          BRANCH IF SMALLER THAN 5
         AL      7,1             ADD CORRECTLY-SIGNED ROUNDOFF
OKAY     ST      7,Q             STORE ROUNDED QUOTIENT
```

We note that a simple check may be made to insure that a fixed-point divide interruption does not occur: if the inequality

$$|C(Rr_1)| < 1/2\ |\text{second operand}|$$

is satisfied, the quotient can be computed correctly.

17. LOGICAL OPERATIONS ANC INSTRUCTIONS

The basic capabilities of a computing system are derived from the many interconnections of basic circuits which perform simple logical functions. Some of these same functions may also be performed on operands in memory and in the general registers through the use of logical instructions, though their applications are of course different. We will discuss some of the instructions which perform logical operations and give a few simple, example6 of their use; other important uses of logical operations will be treated when some of the SI instructions are examined.

Although it is not what we usually would consider a logical instruction, the LA (Load Address) instruction is classified as such, and has many and varied uses in System/360 programming. It is a very simple RX-type instructian: the effective address replaces the contents of Rr1, with the high-order byte being set to zero. Thus, for example, a positive integer n between 0 and 4095 can be placed in a register by executing an LA r,n instruction, where the index and base digits are implicitly zero and the displacement contains the constant n. Instead of writing L 2,=F'1' which requires 8 bytes (4 for the instruction and 4 for the constant), or LH 2,=H'1' which-requires 6 bytes, we can write either LA 2,1 or LA 2,1(0,0) which requires 4 bytes and less execution time, because no memory access is required. Also, because LA does not affect the CC we can clear a register without disturbing a CC setting which may be required at a later point in the-program. For example, suppose we wish to add C(A) and C(B)' and clear the result to zero if it overflows, without changing the CC setting. The two instruction sequences which follow perform the desired task.

```
         L      0,A                                L      0,A
         A      0,B                                A      0,B
         BC     14,ST                              BC     14,ST
         LA     0,0                                L      0,=F'0'
ST       ST     0,ANSWER                  ST       ST     0,ANSWER
```

Because the LA instruction computes an effective address, it also provide6 a simple way to increment the content6 of a register by a small positive amount. For example, LA 4,17(0,4) will increase the contents of R4 by 17, if the original contents'of R4 are between -17 and 2^{24}-18. This restriction is of course due to the fact that the high-order byte of the register into which the result is placed will be set to aero; thus the use of LA for incrementing registers is usually limited to cases where the quantity being incremented is an address or reasonably small integer. For example, suppose we want to perform the shifting operation described in example 6 of Section 15, where it was required that the fullword at N be shifted right enough places so that its rightmost bit is a 1 bit; we will also require that the halfword at CØUNT contain the number of positions shifted.

```
         L      4,N           GET INTEGER
         L      3,=F'-1'      INITIAL SHIFT COUNT
SHIFT    SRDL   4,1           SHIFT A BIT INTO R5
         LTR    5,5           TEST SIGN OF R5
         LA     3,1(0,3)      INCREMENT R3 BY 1
         BC     10,SHIFT      BRANCH IF R5 NOT NEGATIVE
         SLDL   491           MOVE BIT BACK IN PLACE
         ST     4,N           STORE SHIFTED INTEGER
         STH    3,COUNT       STORE SHIFT COUNT
```

By setting the shift count to -1 initially, we guarantee that the correct value will be in R3 when we exit from the loop; the first time the LA instruction is executed, the result will be zero and the setting of the leftmost byte to zero is what we want. The placement of the LA instruction between the LTR and the ensuing BC was done to show that no adverse effects are caused; one would normally place the LTR just before the BC because the relation between the two is then clearer to anyone reading the program,

A third use of the LA instruction, and possibly the most important, is in generating addresses for actual operands in memory. For example, we may require the address of some operand to be in a given register during the execution of a segment of code. Suppose we want to add three integers, and branch after all additions are completed to NØERR if no overflow occurs, and to ERR1 if one or more overflows occur. Let the integers be stored in successive fullwords beginning at Q.

```
        LA     9,NOERR          SET BRANCH ADDRESS FOR NO ERRORS
        L      2,Q              GET FIRST INTEGER
        A      2,Q+4            ADD SECOND INTEGER
        BC     14,OK1           BRANCH IF NO OVERFLOW
        LA     9,ERR1           SET BRANCH ADDRESS FOR 1 OVERFLOW
OK1     A      2,Q+8            A D D THIRD INTEGER
        BCR    14,9             BRANCH IF NO OVERFLOW
        BC     15,ERR1          BRANCH, SOME ADDITION OVERFLOWED
```

It should be noted that the instruction with a mask digit of 15 could also be written **BC 1,ERR1** without affecting the operation of the code, since the inetructicm is reached only if the branching **condition** for the immediately **preceding** instruction is not met; by specifying a mask of 15 it is clear **that** the branch must always be taken. There is one important assumption underlying the use of the two LA instructions: the instructions named **NØERR** and **ERR1** must be addressable, since the LA instruction will simply perform the address **computation** specified by the base and displacement assigned by the Assembler. As mentioned earlier, we are assuming that all symbols (and expressions 'such as Qt8) are 'addressable and that the appropriate
-- base register information has been established elsewhere in the program. It is occasionally easy to forget that the symbols used in LA instructions must be addressable, since no reference is being made to any memory location -- **only** an address is being generated, and no checks for the validity of **that** address are made.

We will give a number of examples later where the **LA** instruction can be used to give-the effect of indexing for instructions for which indexing is not actually possible, namely RS, SI, and SS instructions.

The three logical operations provided by System/360 are AND, OR, and EXCLUSIVE OR. These are relations between pairs of bits, which produce a result depending only on the values of the two bits participating in the operation. The effect of the three operations is *given* in the figure below.

$\wedge$	0	1
0	0	0
1	0	1

AND

$\vee$	0	1
0	0	1
1	0	1

OR

$\oplus$	0	1
0	0	1
1	1	0

EXCLUSIVE OR

Figure 17.1 Logical Functions in **System/360**

In the first case, the result bit is 1 only if the first AND the second operand bits are 1; in the second case the result bit is 1 if either the first OR the second operand bits (or both) is 1; and in the last case, the result bit is 1 if either the first OR second operand bits is 1, EXCLUSIVE of the case where both are 1. Henceforth we will abbreviate EXCLUSIVE OR by XOR. For the instructions listed in Figure 17.2, the operands are fullwords; however, the result of the operation is obtained by matching the corresponding bits of each word, with no interactions between neighboring bits. A few examples will help to clarify this. As before, "FW_2" means the fullword second operand specified by the effective address.

Mnemonic	Type	Action	CC Settings
NR	RR	$C(Rr_1) \leftarrow C(Rr_1) \wedge C(Rr_2)$	0: all result bits are zero
N	Rx	$C(Rr_1) \leftarrow C(Rr_1) \wedge C(FW_2)$	
ØR	RR	$C(Rr_1) \leftarrow C(Rr_1) \vee C(Rr_2)$	1: result bits are not all zero
Ø	RX	$C(Rr_1) \leftarrow C(Rr_1) \vee C(FW_2)$	
XR	RR	$C(Rr_1) \leftarrow C(Rr_1) \oplus C(Rr_2)$	
X	Rx	$C(Rr_1) \leftarrow C(Rr_1) \oplus C(FW_2)$	

Figure 17.2 Logical Instructions

Suppose $C(R4) = 01234567_{16}$, and $C(R9) = EDA96521_{16}$. Then if the instructions indicated are executed, the final contents of R4 will be as shown below the instruction.

NR 4,9	ØR 4,9	XR 4,9
01214521_{16}	$EDAB6567_{16}$	$EC8A2046_{16}$

To see in more detail how these results are obtained, we will examine the fourth hexadecimal digit of each case in binary form in the figure below.

3 $\wedge 9$	0011 $\wedge 1001$	3 $\vee 9$	0011 $\vee 1001$	3 $\oplus 9$	0011 $\oplus 1001$
1	0001	B	1011	A	1010
AND		OR		EXCLUSIVE OR	

Figure 17.3 Examples of Logical Operations

One important use of the N and NR instructions is for "masking" operations in which it is desired to isolate or extract portions of a word. For example, suppose we wanted only the third of the four positive integers packed in the data word illustrated in Figure 14.7. This could be done by shifting as follows:

```
         L      0,DATAWORD     GET INTEGERS
         SRL    0,6            DROP OFF FOURTH ONE
         SRDL   1,13           MOVE THIRD INTO RI
         SRL    1,19           POSITION FOR STORING
         ST     1,THIRD
```

or as follows:

```
         L      0,DATAWORD
         SLL    0,13           DROP OFF FIRST AND SECOND INTEGERS
         SRL    0,19           DROP OFF FOURTH, POSITION FOR STORING
         ST     0,THIRD
```

(If the integers were allowed to have negative-values as well, the SRL instructions would be replaced by SRA.) However, the following instruction sequence using a logical AND is considerably faster:

```
         L      1,DATAWORD     AAAAAAAAABBBBCCCCCCCCCCCCCDDDDDD
         N      1,MASK         0000000000000CCCCCCCCCCCCC000000
         SRL    1,6            0000000000000000000CCCCCCCCCCCCC
         ST     1,THIRD        STORE DESIRED INTEGER
         - - -
         DS     OF             ALIGN TO FULLWORD BOUNDARY
MASK     DC     X'0007FFC0'
```

First, note that the DS OF is required to insure that MASK falls on a fullword boundary -- type X constants have no implied alignment. Second, the mask has 1 bits only in those positions which correspond to the bits (labeled "C") of the third integer in the data word. When the N instruction is executed, all of the bit positions in which the mask is zero will be set to zero, since a 0 bit ANDed to any other bit gives a zero result. In all of the mask's bit positions which are 1 bits, the result is the same as the original bit from the data word, because a 1 bit ANDed to any other bit gives a result identical to that bit.

To illustrate the use of a logical ØR instruction, suppose we want to store a new value for the third integer into the proper part of the data word.

We can do this by shifting the various pieces into place:

```
          L       0,DATAWORD      GET INTEGERS
          S R     D L0,6          MOVE FOURTH INTO R1
          L       0,NEWTHIRD      GET NEW VALUE OF THIRD INTEGER
          SRDL    0913            MOVE IT IN WITH FOURTH
          L       0,DATAWORD      GET INTEGERS AGAIN
          SRL     0,19            DROP OFF  THIRD  AND FOURTH
          SRDC    0,13            MOVE FULL WORD INTO R1
          ST      1,DATAWORD      STORE NEW DATAWORD
```

Alternatively, we can use the logical AND and ØR to do the same:

```
          L       0,DATAWORD      GFT INTEGERS
          N       0,MASKA         CLEAR SPACE FOR THIRD
          L       1,NEWTHIRD      GE? NEW VALUE OF THIRD INTEGER
          SLL     1,6             SHIFT INTO PROPER POSITION
          OR      001             'OR' IWO PLACE
          ST      0,DATAWORD      STORE NEW DATAWORD
          - - -
          DS      OF
    MASKA D C     X'FFF8003F'
```

In this case, the N causes all the bit positions into which the third integer will be placed to be set to zero. The ØR instruction then forms the logical OR of all the bits of R0 and R1. Since the only bits in R1 which may be 1's are in the 13 positions corresponding to the space provided in the word in R0, and because the result of ORing a 0 bit to any other bit is the value of the other bit, the effect is to insert the new value of the third integer in its proper position in R0. This of course assumes that the contents of NEWTHIRD is a positive integer of at most 13 significant bits; if not, an instruction such as   N   1,MASK   should be inserted before the ØR to insure that no extraneous bits are ORed into R0.

The-X and XR instructions are used mainly for inverting the value of a bit or a group of bits: it can be seen from Figure 17.1 that the result of XORing a 0 bit to any other bit is to leave it undisturbed, and the result of XORing a 1 bit is to invert it from 1 to 0 or vice versa. Thus, for example., we can form the one's complement of the number in R7 by subtracting it from a word of all 1 bits, or by executing   X   7,=F'-1'   which does the same thing. We can rewrite the example above to use an X instruction (though in a somewhat roundabout way) as follows:

```
         L     0,DATAWORD     GET INTEGERS
         0     0,MASK         SET THIRD SPACE TO 1 BITS
         X     0,MASK         NOW SET THEM TO ZEROS
         L     1,NEWTHIRD     ETC
         SLL   1,6            ETC
         N     1,MASK         BE SURE THERE ARE- NO EXTRA BITS
         OR    0,1            ETC
         ST    0,DATAWORD
         - - -
         DS    OF
MASK     DC    X'0007FFC0'
```

As another example of the use of the XOR function, suppose we again want to force the integer in R9 to be the next larger multiple of 8 if it is not already a multiple of 8; consider the following code sequences.

```
         A     7,=F'7'        FORCE CARRY SF ANY 1 BITS
         N     7,=F'-8'       SET LAST 3 BITS TO ZERO
```

This is the fastest method, but space is required for the constants.

```
         LA    0,7            C(R0) = 7
         AR    9,0            FORCE CARRY IF ANY 1 BITS
         OR    9,0            FORCE THE THREE BITS TO 1'S
         XR    9,0            NOW SET THEM TO ZERO
```

In terms of space required, this method is superior to the ones illustrated previously.

We will find that the logical operations have considerable use in examining and manipulating individual bits in memory, particularly through the use of certain SI-type instructions. As a final example, suppose we are required to shift the integer contents of R6 (assumed nonzero) left so that the first significant bit is immediately to the right of the sign bit, and store at NØRM the number of positions-shifted.

```
         SR    8,8            SET SHIFT COUNT TO ZERO
SHIFT    SLA   6,1            SHIFT LEFT ONE BIT POSITION
         8C    1,FINIS        IF OVERFLOW, JUMP
         LA    8,1(0,8)       INCREMENT SHIFT COUNT
         SC    15,SHIFT       TRY AGAIN
FINIS    SRA   6,1            REPOSITION
         x     6,DIGIT        RESTORE THE LOST BIT
         ST    8,NORM         STORE SHIFT COUNT
         - - -
NORM     DS    F
DIGIT    D C   X'40000000'
```

In this case we shift left until the overflow indicates that a bit different from the sign bit has been shifted out of bit position 1. The right shift moves everything back, but instead of restoring the lost bit, extends the sign bit into the second bit position of R6 from which the most significant bit was just lost, Since the sign is known to be the opposite of the lost bit, the X operation inverts the second bit to give the desired result.

18. LOOPING, INDEXING, AND SIMPLE ARRAYS

Much of the power of a digital computer derives from its ability to execute sequences of statements repetitively until some condition has been satisfied. Programming with loops is therefore basic to most programs of any size and complexity; we will examine in this section several instructions which simplify the coding of loops, and sane typical uses involving arrays of data.

As a simple example which will be used to illustrate sane of the basic principles, suppose there is a string--- a one-dimensional array -- of 80 bytes beginning at STR and ending at STR+79 which contains character data in the EBCDIC representation. We are required to scan the string and replace all special (non-alphanumeric) characters by blanks: specifically, any character with representation less than C'A' (referring to Table III, it can be verified that this is equivalent to 193_{10}=X'C1') should be replaced by C' ', which has representation X'40', so that letters and digits will be unchanged.

First, consider the following code sequence, which performs the desired processing in a straightforward but rather clumsy way.

```
         SR    0,0            CHARACTERS INSERTED INTO R0
         LR    1,0            CHARACTER COUNT IN R1, INITIALLY 0
         LA    2,C'A'         C(R2) = X'000000C1'
         LA    3,C' '         C(R3) = X'00000040'
         LA    4,STR          FIRST BYTE ADDRESS IN R4
GETCHAR  IC    0,0(0,4)       GET BYTE FROM STRING
         CR    0,2            COMPARE TO LETTER 'A'
         BC    10,OKAY        BRANCH IF LETTER OR DIGIT
         STC   3,0(0,4)       OTHERWISE REPLACE BY A BLANK
OKAY     LA    4,1(0,4)       INCREMENT CHARACTER ADDRESS BY 1
         CA    1,1(0,1)       INCREASE CHARACTER COUNT BY 1
         C     1,=F'80'       COMPARE TO 80
         BC    4,GETCHAR      BRANCH IF LESS THAN 80 TO DO MORE
         - - -
STR      CC    CL80'THIS.IS*80)BYTES-TO,BE(SCANNED+FOR@SPECIAL=CHAR#
```

We will see later that this particular problem can be solved more efficiently in a variety of ways. For the time being, note that the character comparisons are made in the rightmost bytes of registers 0 and 2, and that the address of the byte to be examined is regularly incremented in R4 after being initialized to the location of'the first character. The branch instruction at the end of the loop must branch if C(R1) is less than 80, not if it is less than or equal to 80, since the final test in the latter case would cause the byte at STR+80 to be examined and possibly changed.

A second version of this program which makes use of the indexing capabilities of the IC and STC instructions follows.

```
         SR    0,0             CLEAR R0 FOR CHARACTERS FROM STRING
         LR    1,0             INITIALIZE INDEX TO 0
         LA    3,C' '          C(R3) = BLANK AT RIGHT END
GETCHAR  IC    0,STR(1)        GET CHARACTER FROM STRING
         C     0,=A(C'A')      COMPARE TO LETTER 'A'
         BC    10,OKAY         JUMP IF NOT LESS THAN X'C1'
         STC   3,STR(1)        REPLACE BY BLANK
OKAY     LA    1,1(0,1)        INCREMENT INDEX BY 1
         C     1,=F'80'        COMPARE TO UPPER LIMIT
         BC    4,GETCHAR       BRANCH IF NOT DONE
```

A trivial difference in this version is that the fullword containing the EBCDIC representation of the letter A is now in memory, specified by the literal =A(C'A') rather than in R2 as before: note that =F'193' and =A(X'C1') would give identical results. The addressing of the byte to be examined is now computed using R1 as an index register. The first time the instruction named GETCHAR is executed, C(R1)=0 and the effective address generated will be the actual relocated address of STR, assuming that the necessary base register(s) have been set up correctly. On the last execution of the IC instruction, C(R1)=79 and the last byte of the string will be inserted into R0 for examination. When the LA instruction named ØKAY is executed, C(R1) will be increased to 80, the branching condition for the final BC instruction will not be met, and control will pass to the following instruction.

To illustrate another use of indexing, consider the example of Section 17, where three integers at Q are to be added; in this case, however, after the sum is complete a branch to NØERR is to be taken if no overflows occurred, to ERR1 if exactly one overflow occurred, and to ERR2 if two.

```
         SR     1,1           SET OVERFLOW COUNT TO ZERO
         L      0,Q           GET FIRST INTEGER
         A      0,Q+4         ADD SECOND
         BC     14,A1         BRANCH IF NO OVERFLOW
         LA     1,4(0,1)      INDICATE ONE OVERFLOW
A1       A      0,Q+8         ADD THIRD INTEGER
         BC     14,A2         BRANCH IF NO OVERFLOW.
         LA     1,4(0,1)      INDICATE AN OVERFLOW
A2       BC     15,B(1)       BRANCH INTO BRANCH TABLE
B        BC     15,NOERR      0-ERROR BRANCH
         BC     15,ERR1       1-ERROR BRANCH
         BC     15,ERR2       2-ERROR BRANCH
```

When the instruction named A2 is reached, R1 contains four times the number 'of overflows. This number is used as an index in computing the effective address of the BC instruction at A2, which will be B, B+4, or B+8; the appropriate branch instruction will then cause control to be transferred to the desired location. Note that B need not be on a fullword boundary; the index in R1 must simply be incremented by 4 to account for the length of the BC instructions. Such branch tables often provide a fast and effective way to route control to different parts of a program.

We will now consider the Branch on Count (BCTR and BCT) instructions, which simplify counting operations such as those in the above example. As was the case for the BCR and BC instructions, the branch address is obtained either from Rr_2 for BCTR (unless r_2=0, in which case no branch can be taken) or from the effective address for BCT. In this case, after the branch address is canputed, the branching condition is determined by first algebraically reducing the contents of Rr_1 by one, and then branching unless $C(Rr_1)$=0. Note that the CC is unchanged and has no effect on the branching condition. We-can rewrite our first'example to use a BCT by working backwards along the string of characters from STR+79 to STR, which also allows the use of the same quantity both as an index and a counter.

```
         SR     0,0           CLEAR R0
         LA     1,80          SET R1 TO NUMBER OF PASSES
         LA     2,C'A'        C(R2) = LETTER A
         LA     3,C' '        C(R3) = BLANK
NEXT     IC     0,STR-1(1)    GET CHARACTER
         CR     2,0           COMPARE 'A' TO CHARACTER
         BC     12,OKAY       BRANCH IF SATISFACTORY
         STC    3,STR-1(1)    OTHERWISE BLOT IT OUT
OKAY     BCT    1,NEXT        COUNT DOWN BY 1, JUMP IF NOT 0
```

The use of the expression STR-1 in the second operands of the IC and STC instructions is dictated by the fact that the possible values of C(R1) run between 80 and 1, rather between 0 and 79 as before. This can be thought of as reflecting a difference in the enumeration of the bytes in the string: if we number them from 0 to 79 they would be addressed STR(1), and if the bytes were numbered (in perhaps a more natural fashion) from 1 to 80, they must be addressed STR-1(1). On the final pass through the loop, C(R1)=1; when the BCT instruction is executed, C(R1) is reduced to zero, the branching condition is not met, and control passes to the next sequential instruction. One immediate gain in program efficiency can be seen simply by counting the instructions inside the loop: we have reduced this number from seven to five, which will give approximately the same ratio in processing speeds.

The BCT and BCTR instructions are especially useful in situations where a certain number of passes through a loop is, needed, and no special attention must be paid to indexing quantities, To illustrate several uses of these instructions, consider the following variations on some examples from previous sections.

(1) The fullword at NBR contains a positive integer n; compute the sum of the cubes of the first n integers,

```
         L      4,NBR           C(R4) = INDEX 'K', INITIALLY N
         SR     5,5             INITIALIZE SUM TO ZERO
NEXT     LR     1,4             C(R1) = K
         MR     0,1             K*K
         MR     0,4             K CUBED.
         AR     5,1             ADD TO SUM
         BCT    4,NEXT          DECREASE K BY 1, LOOP
         ST     5,SUM           STORE SUM
```

(2) The halfword at N contains a positive integer n; store at NSQ the sum of the first n odd integers.

```
         SR     0,0             CLEAR SUM TO ZERO
         LH     1,N             GET N FROM MEMORY
LOOP     LA     2,0(1,1)        (COUNT+COUNT) IN R2
         BCTR   2,0             2 * COUNT - 1
         AR     0,2             ADD TO SUM
         BCT    1,LOOP          REDUCE COUNT AND BRANCH
         ST     0,NSQ
```

Because n is contained in a halfword integer, we may use the LA instruction to compute (n + n) in one step, since the result is known to fit in the rightmost 24 bits of R2. The following BCTR instruction cannot branch, since r_2 = 0; hence the only effect is to reduce C(R2) by one, as required. (Remember that the k-th odd integer is 2k-1).

(3) Find the two's complement of the double-length integer stored at ARC.

```
        LM     0,1,ARG         GET DOUBLE-LENGTH NUMBER
        LCR    0,0             COMPLEMENT HIGH-ORDER PART
        LCR    1,1             COMPLEMENT LOW-ORDER PART
        BC     8,X             BRANCH IF " CARRY OUT OF R1
        BCTR   0,0             OTHERWISE REDUCE C(R0) BY 1
X       STM    0,1,ARG         STORE COMPLEMENTED RESULT
```

This is identical to the example in Section 16 except that the BCTR replaces

```
   S    0,=F'1'
```

and thus the CC setting may be different when the STM is executed. The BCTR instruction with r_2=0 may be used in this fashion anywhere in a program; it is shorter and faster than subtracting a constant 1 from memory, but has the possible disadvantage that the CC is not set.

As a further example of the use of the BCT instruction, we present below two examples of program segments which store the cubes' of the integers from 1 to 10 in a table of ten successive fullwords, the first of which is labeled CUBE.

```
        LA     4,10            C(R4) = NUMBER TO BE CUBED
MULT    LR     3,4             MOVE IT TO R 3
        MR     2,3             SQUARE IT
        MR     2,4             AND CUBE X T
        LR     1,4             SET UP INDEX IN R1
        SLL    1,2             MULTIPLY BY 4 FOR FULLWORD LENGTH
        ST     3,CUBE-4(1)     STORE IN CORRECT TABLE POSITION
        BCT    4,MULT          BRANCH BACK 9 TIMES
```

In this case we have used the integer argument being carried in R4 to index the desired word in the table; since the table entries are fullwords, the index must be multiplied by four for successive items, which is why the SLL is used. Because the first entry in the table corresponds to 1 cubed, the expression in the operand field of the ST must be CUBE-4 so that the address of each entry will be correctly calculated'. Another method of doing the same calculation is as follows.

```
          LA    1,CUBE+0*4      ADDRESS OF FIRST TABLE ENTRY
          LA    2,CUBE+9*4      ADDRESS OF LAST TABLE ENTRY
          LA    3,1             C(R3) = NUMBER TO BE CUBED
   MULT   LR    5,3             MOVE MULTIPLICAND
          MR    4,3             SQUARE
          MR    4,3             CUBE
          ST    5,0(0,1)        STORE IN TABLE
          LA    3,1(0,3)        INCREMENT NUMBER TO BE CUBED
          LA    1,4(0,1)        INCREMENT TABLE ADDRESS
          CR    1,2             COMPARE TO END ADDRESS
          BC    12,MULT         BRANCH BACK IF NOT PAST END OF TABLE
```

In this case an explicit address in the ST instruction is used, rather than an implied address as in the first method. This is because the loop termination condition is determined from address arithmetic rather than from tests on any of the quantities being calculated in the loop; we will see that cases often arise where it is convenient to perform such addressing calculations explicitly, rather than rely on the Assembler to assign all bases and displacements. The "index" of the entries in the table may be thought of as running from 0 to 36 in steps of 4.

In most of the programming examples we have examined in which loops were used to perform some iterative task, the termination condition depended on some kind of counting operation. More specifically, many such applications require that some quantity be established as an index whose value is changed regularly by an increment, compared to some comparand, and a branch then be made depending on some condition established by the comparison. Note that the term "index" as used here is meant only to indicate the variable quantity which controls or determines completion of the loop; it may or may not be related to a quantity to be used as an index (that is, specified by an index register specification digit) in an RX instruction, as in the two examples above which compute a table of cubes. In the first illustration, the index of the loop (in R4) is also used (in R1) to index the ST instruction; in the second illustration, the index of the loop is the address contained in R1, but no indexing is performed in any of the RX instructions. The increment may be a negative quantity, in which case it might be more appropriate to call it a decrement; rather than try to use names to distinguish the sign of the quantity to be added to the index, we will assume that the increment can be positive or negative.

For the Branch on Count instructions, the quantities involved are all implied by the instruction: the index is in Rr_1, the increment is -1, the comparand is zero, and the condition for branching is inequality. As might be inferred from the preceding examples, this somewhat restricted set of possibilities is often insufficient to enable the programmer to code a loop effectively. Because loops are such a crucial part of many programs, the System/360 instruction repertoire contains the BXH (Branch on Index High) and BXLE (Branch on Index Low or Equal) instructions to facilitate coding of loops. As was the case for BCT and BCTR, both of these instructions provide the three functions of incrementation, comparison, and conditional branching, but with much greater flexibility.

Both BXH and BXLE are RS-type instructions requiring two register specifications digits r_1 and r_3, as indicated in Figure 14.1. Like the STM and LM instructions, the use of registers other than Rr_1 and Rr_3 may be implied, but in a less simple way. The index is always in Rr_1, and the increment is always in Rr_3. The comparand is contained either in Rr_3+1 (if r_3 is even) or in Rr_3 (if r_3 is odd), That is, if WC write BXLE 0,4,NEXT then the index is in RO, the increment is in R4, and the comparand is in R5, whereas if we write BXLE 0,5,NEXT the index is again in R0, but <u>both</u> the increment and the comparand are in R5. There is a simple notational device which illustrates the fact that the comparand is <u>always</u> contained in an odd-numbered register (if r_3 is even, the comparand is in Rr_3+1, and if r_3 is odd, the comparand is in Rr_3): we will write $Rr_3 \vee 1$ to indicate that the register containing the comparand may be determined by ORing a 1 digit into the r_3 digit. Thus $R8 \vee 1$ refers to R9, and $R9 \vee 1$ is the same as R9. The operation of BXH and BXLE, which is diagrammed in Figure 18.1, is as follows: the sum of the index and increment is computed internally and then compared algebraically to the comparand. Whether or not the branching condition is met is noted -- for BXH this means that the sum is algebraically greater than the comparand, and for BXLE that the sum is algebraically less than or equal to the comparand. It is important to observe that the branching condition is <u>not</u> reflected in a setting of the CC but is determined internally; none of BCT, BCTR, BXH, or BXLE change the CC. The sum then replaces the index, and the branch is taken if the branching condition is met. Note that because the branch address is computed during the "Decode" portion of the instruction cycle <u>before</u> incrementation takes place, the effective address may not be as expected if r_1 and b_2 are the same (unless both are zero, which is unlikely since the branch address would have to be less than 4095). Note also that the comparison takes place <u>before</u> the sum replaces the index; we will give some examples of situations where this is important.

The upper portion of the figure below is a verbal description of the execution of BXH and BXLE; the lower portion indicates explicit register usage by the two instructions.

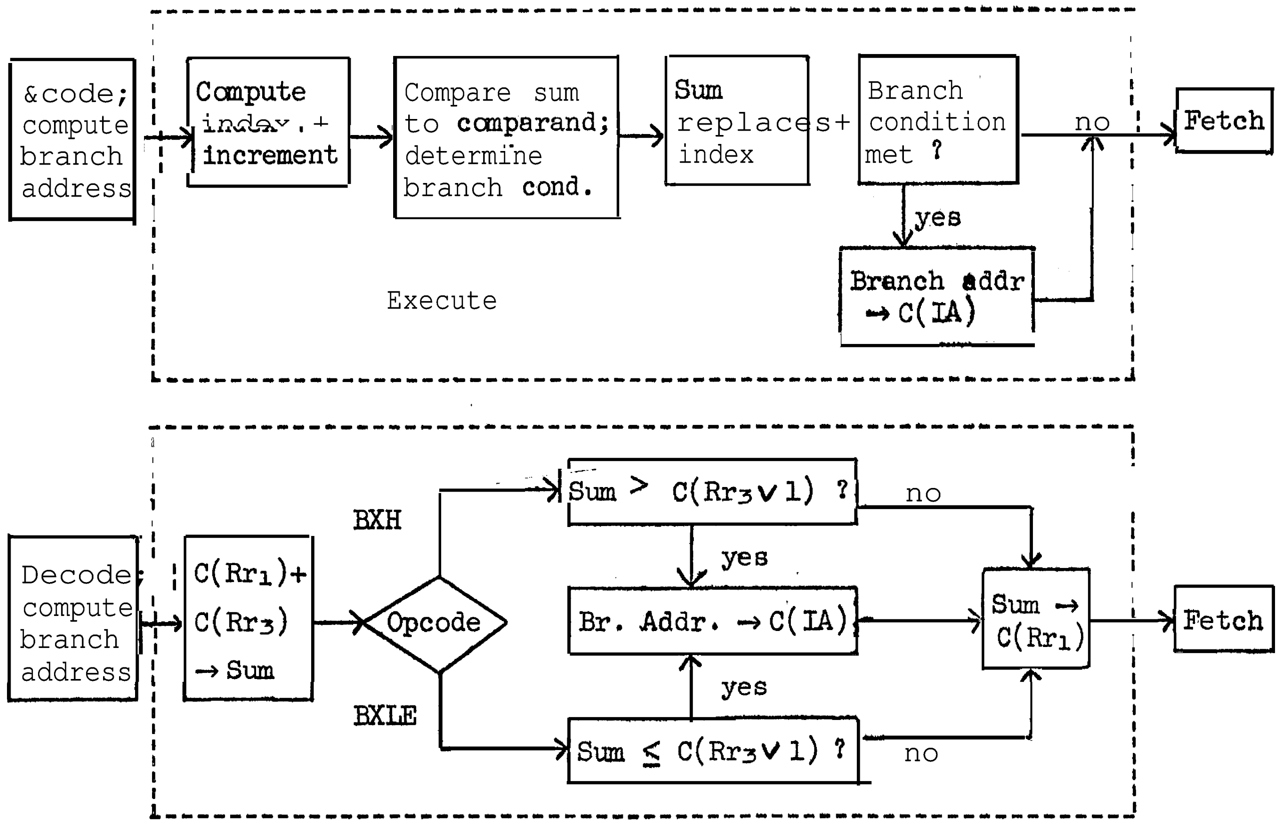


Figure 18.1 Operation of BXH and BXLE Instructions

To illustrate the use of BXH and BXLE, consider the example given at the beginning of this section, where we wish to replace non-alphanumeric characters by blanks. We will rewrite the code sequence to use a BXLE instruction.

```
         LM    0,3,=F'0,0,1,79'
*   CHARACTERS INSERTED INTO R0, INDEX IN R1, INCREMENT IN R2,
4   AND COMPARAND IN R3.
         LM    4,5,=A(C'A',C' ')
8   LETTER 'A' IN R4, AND A BLANK IN R5.
GETCHAR  IC    0,STR(1)         GET A CHARACTER FROM THE STRING
         CR    0,4              COMPARE TO LETTER 'A' IN R4
         BC    10,ALPHANUM      BRANCH IF ALPHANUMERIC
         STC   5,STR(1)         OTHERWISE STORE A BLANK
ALPHANUM BXLE  1,2,GETCHAR      INCREMENT AND BRANCH
```

Note that the values of the index run from 0 to 79; when control reaches the BXLE instruction, the increment in R2 (namely +1) is added to C(R0), and because R2 is an even-numbered register, the sum is compared to the comparand C(R3). If the sum is less than or equal to 79, the branching condition is met and control will. be transferred to the instruction named GETCHAR after the sum is placed back in R1. When control finally passes to the instruction following the BXLE, the contents of R1 will be 80_{10}.

To give an example where the use of BXLE is perhaps more natural, we will rewrite the code segment which computes a table of the cubes of the first 10 integers, starting at CUBE.

```
         LA    7,1              INITIAL INTEGER=1
         LR    8,7              C(R8) =1 FOR INCREMENTING N
         SR    4,4              SET INDEX TO ZERO
         LA    2,4              INCREMENT OF +4 FOR INDEX
         LA    3,36             COMPARAND = 36, IN R3
MULT     LR    1,7              N IN R1
         MR    0,1              N*N
         MR    0,7              N CUBED
         ST    1,CUBE(4)        STORE IN TABLE
         AR    7,8              INCREASE N BY 1
         BXLE  4,2,MULT         INCREASE INDEX BY 4 AND LOOP
```

This segment of code has been written in such a way as to use fewer instructions inside the loop, at the expense of some extra instructions outside the loop. The following two code segments perform the same calculation, but are set up slightly differently.

```
         LA    7,1            INITIAL VALUE OF N=1
         LA    4,4            SET INCREMENT IN R4 TO 4
         LR    2,4            INITIAL INDEX IN R2 IS 4 ALSO
         LA    5,40           COMPARAND IN R5 = 40
MULT     LR    1,7            C(R1) = N
         MR    0,1            N SQUARED
         MR    0,7            N CUBED
         ST    1,CUBE-4(2)    STORE IN TABLE
         LA    7,1(0,7)       INCREMENT N
         BXLE  2,4,MULT       COUNT AND LOOP
```

In this example, the index runs from 4 to 40 in steps of 4, rather than from 0 to 36 as previously. In general there -is no difference between the two methods, except that the second method can be conceptually simpler: since the integer N runs from 1 to 10 by steps of 1, the multiplication by 4 to account for the length of the fullword result makes it natural to have the index run from 4 to 40 in steps of 4. We will. examine some cases shortly where such considerations are important. The use of the LA instruction can yield very slightly increased speeds, since it is faster on some models of System/360 than an AR instruction; the programmer interested in such details should consult the instruction timing tables for the particular CPU he is using. A variation on the above example is given below, where the index and comparand quantities are addresses.

```
         LA    4,CUBE+0*4     SET INDEX TO INITIAL TABLE ADDRESS
         LA    2,4            INCREMENT = 4 FOR FULLWORDS
         AA    3,CUBE+9*4     COMPARAND = FINAL TABLE ADDRESS
         AA    7,1            INITIAL VALUE OF N = 1
MULT     LR    11,7           N
         MR    10,11          N*N
         MR    10,7           N*N*N
         ST    11,0(0,4)      STORE IN TABLE
         CA    7,1(0,7)       INCREMENT N
         BXLE  4,2,MULT       INCREMENT ADDRESS AND LOOP
```

To illustrate the use of the BXH instruction, two of the previous code segments will be rewritten so that the indexing runs in the opposite direction.

```
         LA    7,10           INITIAL VALUE OF N
         A     8,=F'-1'       C(R8) = -1 FOR INCREMENTING N
         LA    4,40           INITIAL INDEX = 40
         L     2,=F'-4'       INCREMENT = -4
         SR    3,3            COMPARAND = 0
MULT     CR    1,7            N
         MR    0,7            N*N
         MA    0,7            N*N*N
         ST    1,CUBE-4(4)    STORE IN TABLE
         AR    7,8            ADD -1 TO N
         BXH   4,2,MULT       COUNT AND LOOP
```

When the instruction following the BXH is reached, the index in R4 will be zero. In fact, we can use -4 for both the increment and comparand as in the following example.

```
        LA    7,10          INITIAL VALUE OF N IS 10
        LA    4,36          INITIAL INDEX = 36
        L     5,=F'-4'      INCREMENT AND COMPARAND ARE -4
MULT    LR    1,7           N
        MR    0,7           N SQUARED
        MR    0,7           N CUBED
        ST    1,CUBE(4)     STORE IN TABLE
        BCTR  790           DECREASE N BY 1
        BXH   4,5,MULT      COUNT DOWN AND LOOP
```

In this' case the r_3 digit is odd, so $Rr_3 \vee 1$ is the same register as Rr_3; the BXH will increment the index in R4 by -4 and branch until the resulting sum becomes -4 also, when control will gass to the instruction following.

Some specialized uses of BXH and BXLE may be obtained by various combinations of register specification digits. For example, suppose the contents of an odd-numbered register such as R9 is zero. Then the instruction BXLE 4,9,X will branch to X only if C(R4) is less than or equal to zero; similarly, BXH 4,9,X would branch to X only if C(R4) is greater than zero. Since the BXH and BXLE neither set nor test the condition code, this technique can be used in situations where a condition code reflecting the state of the contents of R4 is not available, or the current CC setting must be undisturbed, or if it is desirable to avoid using instructions such as LTR followed by a BC.

Suppose we want to perform the inverse of the BCT instruction, namely increment a register by +1 and branch. If C(R7)=1 and the contents of R2 is some integer greater than zero, then BXH 2,7,X will branch to X after incrementing C(R2) by 1 unless the sum overflows. Similarly, if there' is some negative integer in R2, BXLE 2,7,X will branch to X so long as the resulting sum does not exceed +1. If C(R4)=1, the instruction BXH 5,4,X will increment the contents of R5 by 1 and then branch to X if the sum does not overflow; this example is instructive because the index and comparand are in the same register. If the canparison was made _after_ the sum was placed in R5, an equality would always be indicated and the BXH would never branch. Tricky usage of BXH and BXLE as described above is relatively rare, and these instructions find their major use in applications such as table searching and loop control.

In the examples given up to now of loops involving indexing in an array, the choice of a method to perform the indexing arithmetic and the selection of initial and final index values was left open; no formal technique was described. Since arrays and array processing techniques are heavily used, we will examine some general methods for handling arrays.

One-dimensional arrays are relatively simple, since each successive element may be obtained by adding the element length to the address of the preceding element. If for example the halfword integers ko, k_1, --- k_{10} are stored starting at K, then k_n is found at K+2n; if the array elements were fullwords or doublewords, the corresponding addresses would be K+4n arid K+8n respectively. On the other hand, if $k_4 \ldots k_8$ are stored beginning at K, and the length of a single array element is L, then k_n Is found at K+L(n-4). The required subscript arithmetic should be evident -- if the lowest-subscripted element km is stored at K, then the location of k_n (where n > m) is K+L*(n-m). (It is also evident that n need not be greater than m; it i's merely customary to store arrays this way.) An example will help to illustrate this.

Suppose an array of fullword integers x_5 . . . x_{17} is stored beginning at X, and we are required to store their sum at T. The lower and upper subscript bound6 of 5 and 17 are stored at LOWER and UPPER.

```
         SR     0,0          INITIALIZE SUM
         L      1,LOWER      INITIALIZE SUBSCRIPT N,LOWER BOUND = 5
A        LR     2,1          INDEX CALCULATED IN R2
         S      2,LOWER      (N-M)
         SAC    2.2          4*(N-M)
         A      0,X(2)       SUM = SUM + X(N)
         LA     1,1(0,1)     INCREMENT N BY 1
         C      1,UPPER      COMPARE N TO UPPER BOUND
         BC     12,A         IF NOT GREATER, BRANCH
         ST     0,T
         - - -
LOWER    DC     F'5'         LOWER SUBSCRIPT BOUND
UPPER    CC     F'17'        UPPER SUBSCRIPT BOUND
T        cs     F
X        DC     13F'1'       FOR EXAMPLE
```

Now, suppose that the lower and upper subscript bounds of the elements forming the required sum do not have known values,. but we still know that x_5

is stored at X. We can include a portion of the indexing arithmetic in the program at assembly time so that it need not be performed at execution time, namely the factor L*(-m).

```
         SR     0,0            INITIALIZE SUM TO ZERO
         LA     4,4            INCREMENT = ELEMENT LENGTH
         L      2,LOWER        GET LOWEST SUBSCRIPT
         SLL    2,2            MULTIPLY BY ELEMENT LENGTH
         L      5,UPPER        GET HIGHEST SUBSCRIPT
         SLL    5,2            *4 ALSO
ACC      L      0,X-20(2)      ADD AN ELEMENT, CORRECTLY ADDRESSED
         BXLE   2,4,ADD        INCREMENT INDEX AND LOOP
         ST     0,T            STORE TOTAL
```

It can be seen if C(LØWER) = 5 and C(UPPER) = 17 that the same result will be obtained; the first element to be added will be at X-20+(4*5) = X, as desired. The Assembler will of course require that the expression X-20 be addressable; this requirement is sometimes a limitation on the use of this time-saving technique.

Two- and higher-dimensional arrays present a few further complications, which can be handled fairly easily; we will examine two methods for addressing array elements. First, it is necessary to find some way to reorganize the rectangular form of an array into a linear arrangement which conforms to the machine's natural method of addressing successive bytes in memory. A common method is to store successive columns of the array one after another, as indicated below.

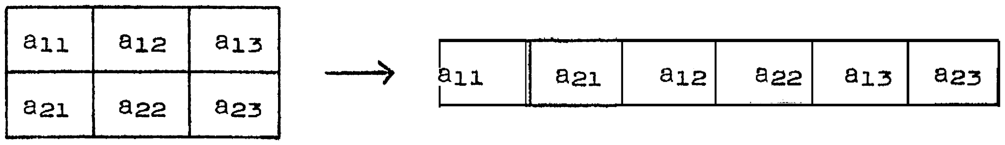


Figure 18.2 Storing an Array in Column Order

It is apparent that any desired arrangement is actually possible, and that a choice-between possibilities must be based on considerations such as convenience and the time and space required to retrieve a particular element. For the example above, the arithmetic necessary to retrieve the element a_{ij} is as follows, assuming that a_{11} is stored at A: to obtain the address of the first element in a given column, we need the address A+L*(j-1)*2 where L is the element length in bytes, and the factor of 2 accounts for the presence of 2 elements in each column. Once having obtained that address the i-th element in the indicated Column is found by adding L*(i-1) to the partially computed address, giving A+L*(2*(j-1)+(i-1)). The quantity added to A is sometimes called a subscripting function or a mapping function, and gives the correspondence between the array subscripts i and j of a particular element and the "linear subscript" which gives the difference between the locations of a_{ij} and a_{11}. It can be seen that if a column-ordered array has r rows, the subscripting function is L*(r*(j-1)+(i-1)). For example, suppose we have an array of fullwords of 5 rows and 7 columns stored at A, and wish to store a_{ij} at X, where i and j are fullwords stored at I and J respectively.

```
         L      6,J              GET COLUMN INDEX J
         BCTR   6,0              FORM J-1
         MH     6,=H'5'          MULTIPLY BY NUMBER OF ROWS
         A      6,I              ADD ROW INDEX I
         BCTR   6,0              DECREASE BY 1
         SLL    6,2              MULTIPLY BY ELEMENT LENGTH, 4
         L      3,A(6)           GET A(I,J)
         ST     3,X              STORE AT X
         - - -
I        DC     F'3'             POSSIBLE VALUE FOR I
J        DC     F'6'             POSSIBLE VALUE FOR J
A        DS     35F              SOMEBODY ELSE COMPUTES THE VALUES
```

As was the case for one-dimensional arrays, part of the subscripting arithmetic can be absorbed into the address of the instruction which references the array element. Thus, the address of a_{ij} becomes A-L*(r+1)+L*(r*j+i), and only the final term need be computed at execution time; the code sequence above can be rewritten as follows.

```
         L      6,J              COLUMN INDEX
         MH     6,=H'5'          *(NUMBER OF ROWS)
         L      6,I              + ROW INDEX
         SLL    6,2              (ALL)*(ELEMENT LENGTH)
         L      3,A-4*(5+1)(6)      C(R3) = A(I,J)
         ST     3,X              STOKE AT X
```

Figure 18.3 Example of Array Subscripting Arithmetic

The address A-L*(r+1) can be seen to be the address of the element "a_{00}" (which may not actually exist) and is sometimes called the address of the "base element" of the array or (unfortunately) the "base address" of the array. Since this almost always has nothing to do with a base address to be used by the Assembler in computing displacements, it is best to avoid the latter terminology.

In the examples above we have assumed that the subscripts could take positive values only, and always had a lower bound of 1; this is not a necessary condition, and if the lower subscript bounds on i and j are i_0 and j_0 respectively, the subscripting function becomes $L*(r*(j-j_0)+(i-i_0))$. In such cases it is usually more difficult to include the factor $-L*(r-j_0+i_0)$ in an expression at assembly time, since the result may not be addressable. We will adopt the convention that all subscripts run upwards beginning at 1 unless the contrary is stated.

A second method of array addressing is useful when processing speeds are important, and occasionally also finds application to arrays of irregularly-spaced or irregular-length data. This involves pre-computing the addresses of portions of the array, and storing those addresses in a separate table, For example, suppose the addresses of the elements a_{11}, a_{12}, and a_{13} in Figure 18.2 are stored as fullwords at CØLADDR, as indicated in Figure 18.4. The notation A(x) means "address of x".

Location	Contents
CØLADDR	A(a_{11})
CØLADDR + 4	A(a_{12})
CØLADDR + 8	A(a_{13})

a_{11}	a_{21}	a_{12}	a_{22}	a_{13}	a_{23}

Figure 18.4 Addressing with Tables of Addresses

The code to store a_{ij} at X might then be as follows.

```
L      7,J            GET COLUMN INDEX
BCTR   7,0            DECREASE BY 1 FOR INDEXING
SLL    7,2            MULTIPLY BY ADDRESS LENGTH = 4
L      6,COLADDR(7)   GET ADDRESS OF COLUMN J
L      5,I            GET ROW INDEX I
BCTR   5.0            DECREASE BY 1
SLL    5.2            MULTIPLY BY ARRAY ELEMENT LENGTH = 4
L      3,0(5,6)       GET A(I,J)
ST     3,X            STORE AT X
```

The main advantage of this scheme is that it avoids the previously required multiplication by the number of rows. The additional expense is In the space required for the table, and the time required for forming it (either during assembly or at execution time). As a final example, suppose we want to store at X the element a_{ij} of a 5-by-5 array of fullwords stored in column order at A; first we will compute a table of column addresses and store them at ADDRTAB. We actually compute not the true addresses of the first element in each column, but that address minus 4, because this then allows us to use the subscript i directly without subtracting 1 during the accessing of the desired array element. The table contents are shown in Figure 18.5 below, where the zero subscript indicates the subtraction of one element length from the address of the beginning of the column.

```
         LH     6,NROWS        C(R6) = NUMBER GF ROWS
         SLL    6,2            MULTIPLY FOR INDEXING BY ELEMENT LENGTH
         LH     5,NCOLS        NUMBER OF COLUMNS IN R5 FOR LOOP COUNT
         LA     9,ADDRTAB      BEGINNING ADDRESS OF TABLE
         LA     0,A-4          ARRAY ADDRESS - ( ELEMENT LENGTH)
STACR    ST     0,0(0,9)       STORE AN ADDRESS IN TABLE
         AR     0,6            INCREASE ADDRESS TO NEXT COLUMN
  -      LA     9,4(0,9)       INCREASE TABLE ADDRESS TO NEXT WORD
         BCT    5,STADR        LOOP UNTIL ALL ADDRESSES COMPUTED
         - - -
NCOLS    DC     H'5'           NUMBER OF COLUMNS
NROWS    CC      H'S'          NUMBER OF ROWS
ACCRTAB  DC     ,5F            SPACE FOR ADDRESSES
```

Location	Contents	Element Addressed
ADDRTAB	A(A-4)	a_{01}
+4	A(A-4+20)	a_{02}
+8	A(A-4+40)	a_{03}
+12	A(A-4+60)	a_{04}
+16	A(A-4+80)	a_{05}

Figure 18.5 Example of Addressing Table Contents

To use this table to perform the desired calculation, we can write the following code sequence.

```
L       2,I                 UET ROW INDEX
L       3,J                 GET COLUMN INDEX
SLDL    2,2                 MULTIPLY BOTH BY  4
L       4,ADDRTAB-4(3)         G E T COLUMN ADDRESS
L       0,0(2,4)            GET A(I,J)
ST      0,X                 STURE A  TX
```

This segment of code gives much faster access to the desired element; the subscripting arithmetic (all but the last two instructions) on a System/360 Model 50 requires 18 microseconds, while the same arithmetic as performed in Figure 18.3 requires 33 microseconds. It should be noted that the faster example uses the SLDL instruction to take advantage of the fact that the array elements and the entries in the address table (sometimes called an 'access table") are of the same length, which might-not be true in general.

In closing this discussion, we will mention that the address table can be constructed by the Assembler if the necessary quantities are known in advance. The items in the middle column of Figure 18.5 can be used as operands in DC statements; remember that in the discussion of A-type constants (address constants) in Section 13, 'it was stated that the abnstant may be relocatable. Though we are not yet in a position to be able to discuss how the correct addresses are eventually placed in the program; We will simply write a sequence of statements Which generates the same address table at assembly time.

```
NRCWS      EQU     5                        NUMBER CF RCWS
L          EQU     4                        LENGTH CF ARRAY E L E M E N T
ACCRTAB D   C      A(A-L)                   A(FIRST CCLUMN - 4 . 1
           GC      A(A+L*(NROWS)-L)         A(SECOND CCLUMN - 4)
           DC      A(A+L*(NROWS*2)-L) A     ?THIRD CCLUMN - 4)
           CC      A(A+L*(NROWS*3)-L) A(FOURTH CCLUMN - 4)
         -  cc     A(A+L*(NROWS*4)-L) A(FIFTH COLUMN - 4)
```

The expressions in the address constants are written in such a way that the programmer need only specify the value to be given to NRØWS in the first EQU statement, and the required addresses are calculated by the Assembler.

19. SI INSTRUCTIONS

Most of the instructions discussed up to now have referred to data which was either in a register or was to be found in memory at a given location. One exception we have encountered is the LA instructions, in which the operand to be placed in Rr_1 was constructed using part of the instruction itself. In particular, writing statements such as LA 5,12 provides a way to place data into a register without an additional memory reference, which would be required if we wrote L 5,=F'12' instead. Instructions which contain one of the operands of the operation to be performed in the instruction itself are called <u>immediate</u> instructions, in the sense that an operand is immediately available. Thus, we could call LA a "Load Immediate" Instruction in those situations where the base and index register specification digits are zero, since the immediate operand comes from the displacement field of the instruction.

The six Instructions to be discussed here make use of an immediate operand contained in the second byte of the instruction, as denoted by "I_2" in Figure 19.1.

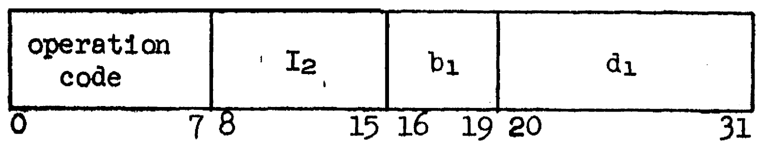


Figure 19.1 SI Instruction Format

In writing SI instruction statements, the first operand will usually be a relocatable expression; the second operand must be a positive absolute expression of value less than 256, so that it will fit into a single byte. The instructions are given in Figure 19.2; the notation "C_1" is meant to indicate the single character or byte at the effective memory address computed from the addressing syllable.

Instruction	Mnemonic	Action	CC Set?
Move	MVI	$C(C_1) \leftarrow I_2$	no
AND	NI	$C(C_1) \leftarrow C(C_1) \wedge I_2$	yes
OR	∅I	$C(C_1) \leftarrow C(C_2) \vee I_2$	yes
XOR	XI	$C(C_1) \leftarrow C(C_1) \oplus I_2$	yes
Compare	CL1	$C(C_1) : I_2$	yes
Test Under Mask	TM	Test Selected Bits of $C(C_1)$	yes

Figure 19.2 SI Instructions

The operation of the first four of these instructions is straightforward, and is illustrated below.

```
(1)  MVI   X,0       sets the byte at X to zero
(2)  MVI   X,255     sets the byte at X to all 1 bits
(3)  MVI   X,C'X'    puts an EBCDIC "X" at X
(4)   NI   X,0       equivalent to (1), except CC = 0
(5)  ∅I    X,255     equivalent to (2), except CC = 1
(6)  ∅I    X,2       sets bit 6 at X to 1
(7)  NI    X,253     sets bit 6 at X to 0
(8)  XI    X,2       inverts bit 6 at X
```

It is occasionally clearer to use other than decimal self-defining terms; example (7) could be written NI X,B'11111101' with the bit to be zeroed immediately indicated. The CC settings after NJ, ∅I, and XI are given in Figure 17.2.

The CLI instruction performs a logical comparison between two 8-bit quantities, which are treated as unsigned integers for the comparison arithmetic.' The result of the comparison is indicated by the CC setting, as given in Figure 16.3. Thus, the statements below would result in the indicated CC settings.

```
CLI   =C'A',X'C1'         CC=0
CLI   =X'00',0            CC=0
CLI   =C' ',B'1000000'    CC=0
CLI   =X'1',X'2'          CC=1
CLI   =C'A',250           CC=1
CLI   =c'XYZ',C'X'-1      cc=2
CLI   =X'1',X'0'          cc=2
```

It is important to remember that the first operand in the comparison canes fran memory. We can rewrite the sample program fram Section 18 which blanks out the special characters in the string at STR by making use of the CLI and MVI instructions; the latter simply stores the second byte of the instruction at the first operand address.

```
         LA    1,80         INITIALIZE LOOP COUNT
NEXT     LA    2,STR-1(1)   CONSTRUCT CHARACTER ADDRESS WITH INDEXING
         CLI   0(2),C'A'    COMPARE ADDRESSED CHARACTER TO LETTER 'A'
         BC    10,ANUM      BRANCH IF NOT LESS THAN 'A'
         MVI   0(2),C' '    BLANK OUT IF NON-ALPHANUMERIC
ANUM     BCT   1,NEXT       COUNT DOWN AND LOOP
```

Because SI instructions cannot be indexed, the LA instruction named NEXT must be used to construct the desired memory address for the character to be tested. The CLI instruction compares the eight bits in memory to the immediate operand C'A', and if the byte in memory contains a bit pattern whose value is greater than or equal to 193_{10}, the following BC will branch around the MVI instruction. If the branching condition is not met, the MVI stores the bit pattern corresponding to the EBCDIC representation of a blank into the character string. It can be seen that the use of these two SI instructions allows considerably simpler coding than in the previous examples of the same processing.

The TM instruction is one of the most useful in the System/360 instruction set for applications where individual bits must be examined. Because no means is provided for addressing individual bits, data in bit form must be treated differently. The immediate operand of the TM instruction is used as a mask to indicate which bits of the addressed byte are to be examined; wherever a 1 bit appears in the mask, the corresponding bit position of the memory operand is examined, and wherever a 0 bit appears in the mask, the corresponding bit of the memory operand is ignored. The result of the examination is indicated in the setting of the Condition Code, as shown in Figure 19.3.

CC	Indication
0	Bits examined are all zeros or mask is zero
1	Bits examined are mixed zero and one
3	Bits examined are all ones

Figure 19.3 CC Settings after TM Instruction

One special case of the TM instruction can arise if the mask specified by I_2 is zero (indicating that no bits are to be examined); the CC is simply set to zero. To illustrate the use of the TM instruction, consider the following examples,

1) Branch to MINUS if the fullword integer stored at NUM is negative. (This technique can be used to avoid having to load a register.)

```
        TM      NUM,X'80'       TEST LEFTMOST BIT
        BC      1,MINUS         BRANCH IF A 1 BIT
```

2) Branch to EVEN if the fullword integer stored at NUM is even.

```
        TM      NUM+3,1         TEST RIGHTMOST BIT OF FULLWORD
        BC      8,EVEN          BRANCH IF ZERO
```

3) Branch to MIXED if the bits in the byte at B are not all zero or all one.

```
        TM      B,255           TEST ALL BITS
        BC      4,MIXED         BRANCH IF MIXED 0 AND 1
```

4) Branch to SMALL if the value of the halfword integer at HNUM is between -512 and 511.

```
        TM      HNUM,X'FE'      TEST LEFTMOST 7 BITS
        BC      9,SMALL         BRANCH IF ALL 0 OR 1
```

When used in conjunction with the NI, ØI, and XI instructions, TM provides a simple means of setting and testing yes-no indicators in a program. For example, suppose we wish to add the three fullword integers stored beginning at Q, and afterwards branch to NØERR if no overflows occurred and to ERRØR if one or more overflows occurred.

```
        MVI     FLAG,0          SET INDICATOR FOR NO OVERFLOWS
        L       0,Q             GET FIRST INTEGER
        A       0,Q+4           ADD SECOND INTEGER
        BC      14,NEXTA        BRANCH IF NO OVERFLOW
        OI      FLAG,1          SET OVERFLOW FLAG 'ON' (TO 1)
NEXTA   A       0,Q+8           ADD THIRD INTEGER
        BC      1,ERROR         BRANCH IF OVERFLOW TO ERROR
        TM      FLAG,1          OTHERWISE EXAMINE OVERFLOW FLAG BIT
        BC      8,NOERR         IF BIT WAS ZERO, NO OVERFLOWS
        BC      1,ERROR         IF ONE, OVERFLOW OCCURRED
         - - -
FLAG    DS      X               OVERFLOW FLAG BYTE
Q       DS      3F              INTEGERS TO BE ADDED
```

The ØI instruction ORs a 1 bit into the rightmost bit position of the byte named FLAG, thus setting it to a 1. Note that only the rightmost 'bit of the byte is being used; the other bits might be used to indicate other conditions detected elsewhere in the same program.

As another representative example of the use of these instructions, suppose we are required to process a list of n halfword integers stored at LIST, where the positive nonzero fullword integer n is stored at N. Suppose that the processing requires that the elements of the list be added together, except that alternate elements of the list are to be added twice; the rightmost bit of the byte named SWITCH is set to 1 if the first element is to be added twice.

```
         LA     4,LIST          INITIAL LIST ADDRESS IN R4
         L      3,N             NUMBER OF ELEMENTS IN R3
         SR     6,6             INITIALIZE SUM TO ZERO
LOAD     LH     5,0(0,4)        GET A HALFWORD LIST ELEMENT IN R5
         AR     6,5             ADD TO SUM ONCE
         TM     SWITCH,1        TEST SWITCH BIT
         BC     8,ONCE          BRANCH IF 0, ADD ONLY ONCE
         AR     6,5             ADD A SECOND TIME
ONCE     LA     4,2(0,4)        INCREMENT LIST ADDRESS BY 2
         XI     SWITCH,1        INVERT SWITCH BIT
         BCT    3,LOAD          BRANCH TO GET NEXT ELEMENT IF NOT DONE
```

Since the XOR of a 1 bit and any other bit inverts the value of the latter, the XT instruction alternately sets the switch bit to 0 and 1. The TM instruction examines only the rightmost bit. of SWITCH; the branching condition will be met if that bit is zero.

A technique which occasionally finds use in such an application involves changing the mask field of a branch instruction so that italternately contains B'1111' and B'0000', causing an unconditional branch to alternate with a-no-operation. The above code sequence can be rewritten to use such a technique as shown below.

```
         L      1,N               GET NUMBER OF ELEMENTS TO BE ADDED
         LA     0,2               SET UP INCREMENT OF 2 IN R0
         AR     1,1               2*N
         SR     1,0               2*(N-1) IN R1 = COMPARAND FOR BXLE LOOP
         SR     2,2               INITIALIZE INDEX IN R2 TO ZERO
         LR     3,2               SAME FOR SUM IN R3
         OI     BRNCH+1,X'F0'     SET SWITCH FOR SINGLE ADD ON FIRST PASS
         TM     SWITCH,1          CHECK SWITCH TO SEE IF SETUP IS CORRECT
         BC     8,ADD             JUMP IF BRANCH HAS BEEN SET CORRECTLY
         NI     BRNCH+1,X'0F'     OTHERWISE SET UP TO ADD TWICE ON 1ST PASS
ADD      AH     3,LIST(2)         ADD A TERM
BRNCH    BC     0,FLIP            MASK FIELD HERE IS ALTERNATED BY XI
         AH     3,LIST(2)         ADD AGAIN IF NECESSARY
FLIP     XI     BRNCH+1,X'F0'     INVERT BRANCH MASK BITS
         BXLE   2,0,ADD           COUNT AND LOOP
         ST     3,RESULT          STORE ANSWER APPROPRIATELY
```

There. are several features of this example to be noted. First, the mask field of the second BC instruction must be addressed at BRNCH+1 rather than at BRNCH, because the latter is the name of the byte containing the operation code. Second, the instructions preceding the loop which initialize the mask field might be necessary because this segment of code may be part of a larger program which executes it many times, and we have no *assurance* that the mask field will be preset correctly. Third, the instructions which manipulate the mask bits are written in such a way as to leave untouched the index register specification digit in the second byte of the instruction at BRNCH. This is necessary because we do not want to insert extraneous bits (thereby causing indexing to be performed), and because in general there can be information there which must be unmodified.

The above technique of actually modifying an instruction in memory can occasionally yield higher processing speeds, but it is not generally considered a good programming practice for the following reasons:

(a-) the coding tends to be more. difficult to understand, since a reader cannot tell with any degree of certainty what is to be done by a given instruction if it is subject to modification by other parts of the program;

(b) checking out the program is more difficult, since it is usually easier to keep track of data (such as at SWITCH in the previous example) than parts of instructions;

(c) if it is necessary to 'rewrite a portion of the program it may be difficult to find all the instructions which modify others;

(d) if the program must be re-enterable (a property of coding which is involved in multiprogramming applications and interruption processing, which will be treated later) such a technique is forbidden.

This might appear to contradict the earlier statements that the flexibility of a computer is derived from its ability to modify the instruction sequences it executes; by this we simply meant that the program can control its paths of execution, rather than that it modifies the actual instructions as was done here. A degree of instruction modification is provided by the Execute instruction, to be discussed later.

To show that the above example need no-t rely on program modification, we give two further code segments which perform the same calculation more rapidly; the first uses two separate add sequences.

```
         L      1,N              SET UP COMPARAND IN R1
         BCTR   1,0              N-1
         SLL    1,1              2N-2 IN R1
         LA     0,2              INCREMENT IN R0
         SR     3 9 3            INITIALIZE SUM TO ZERO
         LR     2,3              SAME FOR INDEX
         TM     SWITCH,1         TEST WHETHER FIRST TERM ADDS TWICE
         BC     1,TWICE          BRANCH IF BIT=1, MEANING YES
ONCE     AH     3,LIST(2)        ADD A TERM ONCE
         BXH    2,0,NEXT         INCREMENT INDEX AND LEAVE LOOP IF DONE
TWICE    AH     3,LIST(2)        ADD A TERM
         AH     3,LIST(2)        ...TWICE
         BXLE   2,0,ONCE         INCREMENT INDEX AND LOOP
NEXT     - - -                   CONTINUATION OF PROGRAM
```

The second adds all the terms in one loop and the alternate ones in another.

```
         L      1,N              GET  N
         BCTR   1,0              N-1
         AR     1,1              COMPARAND = 2(N-1)
         LA     0,2              INCREMENT = 2
         SR     393              INITIALIZE SUM TO ZERO
         SR     2,2              INITIALIZE INDEX TO ZERO
ADD1     AH     3,LIST(2)        ADD ALL TERMS ONCE
         BXLE   2,0,ADD1         INDEX THROUGH ENTIRE LIST
         LR     2,0              NOW SET INDEX TO 2 INITIALLY
         AR     0,0              SET INCREMENT TO 4 FOR ALTERNATE TERMS
         TM     SWITCH,1         SEE IF FIRST TERM ADDS SINGLY
         BC     8,ADD2           BRANCH IF YES
         SR     2,2              OTHERWISE RESET INITIAL INDEX TO ZERO
ADD2     AH     3,LIST(2)        ADD AN ALTERNATE TERM FOR SECOND TIME
         BXLE   2,0,ADD2         INCREMENT INDEX BY 4 AND LOOP
```

This last example is slightly slower than the previous one, because more , branching instructions are executed; in particular, it will not work correctly if n = 1.

The above examples have illustrated the use of logical instructions mainly for control purposes. Another important application is the manipulation of data in bit form -- that is, data which assume only two values. For example, suppose that part of the record of a person carrying automobile insurance requires the following yes-no information: (1) age less than 25? (2) male? (3) driver training course completed? (4) married? (5) any previous claims? (6) assigned risk?: Let the "yes" answers be represented by 1 bits in the first six bit positions of the byte named STATUS. The following tasks may be performed by the indicated instruction's.

1) The policy holder has passed his 25th birthday.

```
        NI      STATUS,B'01111111'
```

2) The policy holder has married.

```
        TM      STATUS,B'00010000'
        Bc      1,BIGAMY
        ØI      STATUS,B'00010000'
```

3) The policy holder has submitted a claim; if it is the first, branch to TSK, otherwise branch to TSKTSK.

```
        TM      STATUS,B'1000'
        BC      1,TSKTSK
        BC      15,TSK
```

4) If the policy holder is single, male, less than 25, and has not completed a driver training course, branch to HIGHCØST.

```
        JM      STATUS,X'30'   TEST MARRIED AND TRAINING
        BC      7,NEXT
        JM      STATUS,X'C0'   TEST AGE AND SEX
        BC      1,HIGHCOST     IF YOUNG MACE, BRANCH
NEXT  - - -
```

5) If the policy holder is an assigned risk, indicate that he has previous claims if he also has no driver training.

```
        TM      STATUS,X'4'
        BC      8,NEXT
        TM      STATUS,X'20'
        BC      1,NEXT
        OI      STATUS,X'8'
NEXT  - - -
```

6) If the policy holder is married or has completed driver training, branch to LØRISK.

```
         TM      STATUS,MARRIED+TRAINING
         BC      5,LORISK
MARRIEC  EQU     16
TRAINING EQU     32
```

As a final example of the use of SI instructions, suppose there is a fullword integer stored at I!? which we wish to convert to a character string of decimal digitswhich can be printed, with the sign of the number preceding the first significant digit; if the number is zero, the characters "+0" should be placed at the right-hand end of the character string. 'Since a fullword integer can be at most 10 decimal digits long, we will reserve 11 bytes for the result at NBR. The conversion is performed according to the scheme given in Section 2.

```
         LA      2.10           SET UP TO BLANK GUT RESULT AREA
BLANK    LA      3,NBR-1(2)     CONSTRUCT BYTE ADDRESS
         MVI     0(3),C' '      STOKE BLANKS IN FIRST 10 BYTES
         BCT     2,BLANK        BRANCH BACK 9 TIMES
         L       1,N            GET NUMBER TO BE CONVERTED
         LPR     1.1            TAKE ITS MAGNITUDE
         LA      3,NBR+10       SET UP ADDRESS OF RIGHTMOST DIGIT
CNVTLP   SR      0.0            CLEAR HIGH-ORDER REGISTER
         0       0,=F'10'       GENERATE A DIGIT BY DIVISION
         STC     0,0(0,3)       STURE THE REMAINDER BYTE
         OI      0(3),C'0'      GIVE DIGIT CGRRECJ EBCDIC REPRESENTATION
         BCTR    3,0            MOVE CHARACTER POINTER 1 BYTE TO THE LEFT
         LTR     1.1            SEE IF DONE, CUOTIENT GOES TO ZERO
         BC      2,CNVTLP       IF NOT ZERO, GENERATE MGRE DIGITS
         MVI     0(3),C'+'      ASSUME SIGN XS +, STORE THAT CHARACTER
         TM      N,X'80'        CHECK ACTUAL SIGN OF ARGUMENT
         BC      8,ALLDONE      BRANCH IF IT WAS INDEEED POSITIVE
         MVI     0(3),C'-'      OTHERWISE PLANT A - SIGN IN THE STRING
ALLCONE  - - -                  REST OF PROGRAM

NBR      OS      CL11           OUTPUT CHARACTER STRING
N        GS      F              NUMBER TO BE CONVERTED
```

20. SS Instructions

As the name implies, Storage-to-Storage instructions work with operands which are entirely in memory; except for TRT end EDMK, the only reference to or use of the general registers by SS instructions is for addressing purposes. This allows considerable freedom in the arrangement of operands in memory, particularly since the data to be manipulated by SS instructions may be of variable length . Our concern in this section will be with the first nine instructions in Table VII, which are listed for convenience in Figure 20.1. The remaining SS instructions, which are primarily used for handling data in pecked decimal format, will be discussed later.

Mnemonic	Instruction	Mnemonic	Jnstructkon
Mvc	Move	ØC	OR
MVN	Move Numerics	NC	AND
MVZ	Move Zones	XC	Exclus ive OR
TR	Translate	CLC	Compare
TRT	Translate and Test		

Figure 20.1 Some Storage-to-Storage Instructions

All of the above instructions have the format illustrated in Figure 20.2 below,

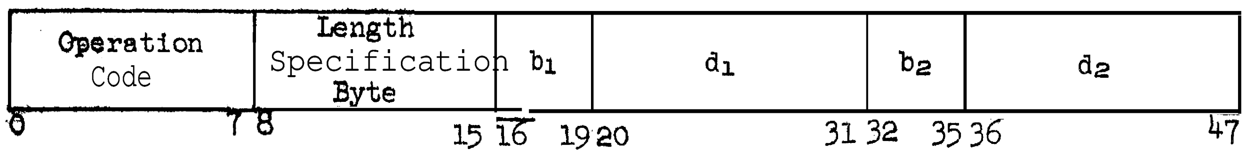


Figure 20.2 Format of Some Storage-to-Storage Instructions

Before discussing the instructions themselves, we must examine some of the details involved in specifying the number to be placed. by the Assembler in the Length Specification Byte, the second byte of the instruction. As can be seen from Figure 20.2, five operand-field quantities in all must be provided: the base and displacement of the address of the first and second operands, and e number which specifies the length in bytes of the date to be manipulated. To illustrate one way of giving this information, suppose we wish to move 23 bytes from the area of memory beginning at A to the area beginning at B; we could write MVC B(23),A to perform the task. Note that only two operands are specified in the operand field entry of these instructions, and that the number in parentheses is not an index register specification but the number of bytes to be moved; it is expected that 'the Assembler will compute displacements and assign bases for us, since we have used implied operand addresses. There are several other ways to specify the length specification byte; these are shown in Figure 20.3. For an explicit length specification, the value provided is used; for an implied length, the Assembler will determine an appropriate value in a way to be described shortly.

Explicit Length	Implied Length
$s_1(L),s_2$	s_1,s_2
$d_1(L,b_1),s_2$	$d_1(,b_1),s_2$
$s_1(L),d_2(b_2)$	$s_1,d_2(b_2)$
$d_1(L,b_1),d_2(b_2)$	$d_1(,b_1),d_2(b_2)$

Figure 20.3 Length Specification for Some SS Instructions

To illustrate the writing of an explicit length, suppose we again want to move 23 bytes from A to B, and we know that if R9 is used as a base, the displacements computed for A and B will be 125_{16} and $47D_{16}$ respectively. Then to achieve the desired result we could write any of the following four instructions corresponding to the first column of Figure 20.3:

```
MVC    B(23),A
MVC    X'47D'(23,9),A
MVC    B(23),293(9)
MVC    1149(23,9),X'125'(9)
```

where equivalent decimal and hexadecimal self-defining terms have been used to specify the displacements d_1 and d_2.

It is often the case, however, that one does not want to be required to specify an explicit length, particularly in cases where the length should be apparent from the operands involved. For example, suppose the symbol B is defined in a DC or DS statement as in the program segment below.

```
     MVC       B,=120C' '              SET FIELD AT B TØ BLANKS
     ---
  B  DS        CL23
```

It is apparent that if more than 23 bytes were moved by the WC instruction that the data or instructions following the byte at B+22 could be overwritten; thus the length should be determined from the first, or receiving, operand rather than the second. This, in fact, <u>is</u> what the Assembler does : if no explicit length is given, the length attribute of the symbol or expression in the first operand is used as the length specification. In the example above it is evident that the length attribute of the symbol B is 23, so that the correct result is obtained. If the first operand is an expression rather -than a single term, the length attribute is determined from the following rule :

1. The length attribute of an expression is the length attribute of the leftmost term.

Thus, if we wrote MVC B-4+X'5'-1,=120C' ' the length specified would be 23, whereas if we wrote MVC X'5'+B-5,=120C' ' the length specified would be 1, because

2. The length attribute of a self-defining term is always 1.

In this example, a knowledge of the base and displacement to be assigned when addressing the symbol B (namely 9 and $47D_{16}$) does not give the correct Length when an implied length is given: MVC X'47D'(,9),A specifies a length of 1 rather than 23, because X'47D' is a self-defining term, and

3. If an explicit base and displacement are given, the length specification is the length attribute of the expression Written for the displacement.

These rules are summarized in Figure 20.4.

Form of First Operand	Address Specification	Length Specification	Length Used
s_1	Implied	implied	length attribute of s_1
s_1(L)	implied	explicit	L
d_1(,b_1)	explicit	implied	length attribute of d_1
d_1(L,b_1)	explicit	explicit	L

Figure 20.4 Determination of Length Specification Byte

Because situations occasionally arise where it is useful to specify an implied length with an explicit base and displacement, and the desired length is not the same as the length attribute of the displacement expression, an alternative technique is provided. We could have written

```
MVC     B-B+X'47D'(,9),A
```

in the example above, and the length attribute of the displacement expression would then be computed to be equal to the length attribute of' B. Such constructions are cumbersome, and it is preferable to use a Symbol Length Attribute Reference, which was mentioned in the discussion of terms In Section 11.

A Symbol Length Attribute Reference is written as an L followed by an apostrophe followed by a symbol, as in L'B; it is an absolute term with a value equal to the length attribute of the symbol. Because symbols can be defined in several ways, the following additional rules are needed:

1. The length attribute of a Location Counter Reference (*) is the-length of the instruction in which it appears; thus MVC B(L'*),A will assign a length of six.
2. If the symbol was defined in an EQU statement with * or a self-defining term in the operand field, the length attribute assigned will be 1.
3. The length attribute of a literal is not defined; thus constructions such as MVC B(L'=C'RAY'),=C'RAY' are incorrect,

Thus we can rewrite our simple example above, which uses an explicit base and displacement, as MVC X'47D'(L'B,9),A

Before discussing the various instructions in Figure 20.1, one further detail must be noted. Because the length specification fits in a single byte, it may assume one of the 256 possible values between 0 and 255: these specify lengths between 1 and 256. This somewhat peculiar construction is due to two factors: first, every SS instruction always operates on at least one byte; second, while all the instructions listed in Figure 20.1 process data from left to right (in order of increasing addresses), there are other SS instructions which process data from right to left (in order of decreasing addresses). In these latter cases, before performing any operations the CPU must be able to construct the address of the rightmost byte of the operand string (remember that all operands are addressed at the lowest-numbered location). It is simplest to do this by adding the appropriate length specification to the effective address of the operand in question, because there are k+1 bytes in a string beginning at location n and extending through location n+k. Such considerations will normally be of little interest to the programmer, since he will allow the Assembler to determine the necessary quantities from the operands provided in the instruction statement. However, it is sometimes necessary at execution time to compute the number of bytes to be manipulated, so that the relationship between the actual contents of the Length Specification byte and the number of bytes involved becomes important. An illustration of this is given in example (4) later in this section. Thus, in summary, the Length Specification Byte contains a number which is one less than the number of bytes to be operated on, unless an explicit length of zero is given, in which case a zero is assembled also. The following instructions would therefore be assembled as indicated, assuming the same displacements for the symbols A and B relative to C(R9) as previously.

```
        INSTRUCTION              ASSEMBLED FORM

        MVC   B(23),A            D216 9470 9125
        MVC   B(1),A             D200 947D 9125
        MVC   B(0),A             0200 947D 9125
        MVC   0(L'*),29(12)      D205 0000 C01D
        MVC   15(L'B-4,3),B      D212 300F 9470
        MVC   B,A                D216 947D 9125
        MVC   H(L'H,H),H         0200 8008 0008
        MVC   H(H,H),H(H)        D207 8008 8008
        MVC   H+B-A(,9),A        0200 9360 9125
        MVC   T,B-4              D216 947D 9479
        MVC   B-A+4(9),A         D208 035C 9125
        - - -
e       OS    CL23
T       EQU   B
H       EQU   8
```

As indicated earlier, the MVC instruction moves the specified number of bytes from an area whose lowest-addressed byte is at the effective second operand address to an area starting at the first operand address. There are no restrictions on overlapping of the two areas, so that various functions such as propagating a character through an area or shifting the bytes in an area may be performed as in the following examples; we need only remember that all SS instructions are executed in such a way that each byte is stored before the next byte to be operated on is retrieved from memory.

(1) Set the 120-byte area beginning at LINE to blanks.

```
         MVI   LINE,C' '         STORE EBCDIC BLANK AT LINE
         MVC   LINE+1(119),LINE  PROPAGATE THROUGH REMAINING AREA
```

This requires less storage space than

```
         MVC   LINE(120),=120C' '
```

(because space is required for the literal) but slightly more execution time.

(2) Shift the 80-byte character string beginning at STR to the left; by two characters, leaving blanks in the vacated positions.

```
         MVC   STR(78),STR+2
         MVC   STR+78(2),=C' '   TWO BLANKS TO END
```

(3) Exchange the contents of the halfword integers at A and B.

```
         MVC   TEMP,A            MOVE A TO TEMPORARY LOCATION
         MVC   A,B               MOVE B TO A
         MVC   B,TEMP            MOVE OLD C(A) FROM TEMP TO B

TEMP     DS    XL2
A        DS    H
B        DS    H
```

Note that no registers were changed in the above instruction sequence.

(4) R8 and R9 contain respectively the address and length of a message of less than 120 characters. Move the message to the area named LINE.

```
         BCTR  9,0               DECREASE LENGTH BY 1 FOR CPU
         STC   9,MVC+1           STORE AS LENGTH BYTE OF MVC INSTRUCTION
MVC      MVC   LINE(0),0(8)      MOVE CORRECT NUMBER OF CHARACTERS
```

The BCTR is used to reduce the character count from its "true" value to the value required by the CPU in the execution of the MVC, namely one less than the number of bytes to move.

The MVN and MVZ instructions work in exactly the same way as MVC, except that only the rightmost 4 bits (the "Numeric" postion of a character) and leftmost 4 bits (the "gone" portion of a character) arc moved, respectively, While these two instructions are occasionally useful for other purposes,, their main applications concern data in packed decimal format. To illustrate some simple uses, consider the following two examples.,

(5) Convert the positive halfword integer at N to a string of 5 EBCDIC characters beginning at NDEC which give the decimal representation of C(N).

```
         LH    1,N            GET NUMBER TO BE CONVERTED
         LA    2,5            COUNT NUMBER OF DIGITS IN R2
X        SR    0,0            CLEAR HIGH-ORDER FEGISTER
         D     0,=F'10'       GENERATE A DIGIT
         STC   0,NDEC-1(2)    STORE DIGIT IN OUTPUT STRING
         BCT   2,X            COUNT AND BRANCH UNTIL DONE
         MVZ   NDEC(5),=5X'FF'   ATTACH ZONES FOR EBCDIC REPRESENTATION
         - - -
NDEC     DS    CL5
```

-Note that we could have used the literals =5C'0' or =5C'9' in the MVZ instruction, with the same results.

(6) Convert the 5-digit decimal number in EBCDIC form at NDEC to a fullword binary integer and store it at M.

```
         MVN   TEMP,NDEC      RETRIEVE NUMERIC PORTIONS OF DIGITS
         LA    3,TEMP         ADDRESS OF CURRENT DIGIT IN R3
         LA    2,5            NUMBER OF DIGITS
         SR    0,0            CLEAR R0 FOR DIGITS
         LR    1,0            AND R1 FOR NUMBER BEING GENERATED
MULT     MH    1,=H'10'       MULTIPLY ACCUMULATED PART BY 10
         IC    0,0(0,3)       INSERT DIGIT FROM INPUT, NO ZONES
         AR    1,0            ADD TO PARTIAL SUM
         LA    3,1(0,3)       INCREMENT DIGIT ADDRESS
         BCT   2,MULT         COUNT AND LOOP
         ST    1,M            STORE RESULT
         -
TEMP     DC    XL5'0'         ZONES PRESET TO ZERO, DIGITS MOVED IN
```

We note with reference to these two examples that there are instructions available in System/360 which considerably simplify the conversion of numbers between binary and decimal forms; they will be treated later.

The logical instructions NC, ØC, and XC perform the logical operations described in Figure 17.1 upon two strings of bytes, leaving the result in the first operand string, and set the CC as in Figure 17.2. Consider the following examples.

(7) Clear the 120-byte area at LINE to zero.

```
        XC      LINE(120),LINE
```

Note that we could also have used the same technique as in example (1) above; the use of XC is usually slightly slower due to the necessity for actually performing the XOR operation, but requires less space in the program.

(8) Branch to YES if the fullword integer at LUMP is zero.

```
        ØC      LUMP(4),LUMP                NC      LUMP(4),LUMP
                                  or
        BC      8,YES                       BC      8,YES
```

In each case the first and second operands are identical so the only result of the logical operation is to set the CC; no data is changed. This technique is useful when a register is not free so that performing the sequence L followed by LTR would be awkward, or when the data is not aligned; it will usually be slower, however.

(9) Suppose there are two fullwords X and Z in memory which contain Pour positive integers each, packed as illustrated in Figure 14.7. Replace the second of the integers in the word at X by the corresponding value from the word at Z.

```
        MVC     TEMP,Z          MOVE NEW VALUE TO TEMPORARY LOCATION
        NC      TEMP,MASK       ELIMINATE ALL BUT SECOND INTEGER
        OC      X,MASK          SET ALL BITS TO 1 IN 2D INTEGER POSITION
        XC      X,MASK          NOW SET THEM TO ZERO
        OC      X,TEMP          INSERT NEW VALUE INTO WORD AT X
        - - -
TEMP    DS      XL4
MASK    DC      XL4'00780000'   MASK BITS FOR SECOND INTEGER POSITION
```

The CLC instruction compares two strings of bytes, one byte at a time, until either an inequality is discovered or the required number of bytes has been compared. As was the case for the CLI instruction, the comparison is made between unsigned positive logical quantities.

(10) Two positive fullword integers are stored at S and T. Branch to TBIG if C(T) is algebraically larger than C(S).

```
CLC     T(4),S
BC      2,TBIG
```

(11) Two negative fullword integers are stored at S and T. Branch to TNB if C(T) is algebraically less than or equal to C(S);.

```
CLC     T(4),S
BC      12,TNB
```

(12) A list of 100 names and occupations, each contained in a block of 60 bytes, is stored beginning at LIST. If any of the blocks matches the name and occupation at WHØ, branch to FØUND.

```
        LA    1,LIST          INITIALIZE TO ADDRESS OF FIRST BLOCK
        LA    2,100           SET COUNT TO NUMBER OF BLOCKS
TEST    CLC   0(60,1),WHO     COMPARE BLOCKS
        BC    8,FOUND         BRANCH IF BLOCKS ARE EQUAL
        LA    1,60(0,1)       OTHERWISE INCREMENT ADDRESS BY 60
        BCT   2,TEST          COUNT DOWN FROM 100 AND BRANCH
        8C    15,NOTFOUND     NO MATCHING BLOCK WAS FOUND
```

The remaining two instructions to be examined are TR and TRT. These are flexible instructions which can greatly simplify many complex programming tasks; they appear complicated when first encountered, but in reality are quite straightforward in their operation. We will examine TR first.

Like MVC, the TR instruction moves bytes from the second operand location to the ffrst operand location, but in a less direct way. The operation actually performs a sort of pseudo-indexing, as follows:

(a) an "argument" byte is obtained from the first operand location;

(b) the value of that byte (as an 8-bit logical integer) is used as an index to access a "function" byte from the second operand location: the address of the accessed byte is the effective second operand address plus the value of the argument byte from the first operand;

(c) the accessed function byte replaces the argument byte frcm the first operand string;

(d) this process continues until the number of bytes indicated by the length specification byte has been translated.

For example, suppose the string of 5 argument bytes at P contains X'0201040503', and the character string at G contains C'ABCDEF'. Then if we execute the instruction TR P(5),G the final contents of the 5 bytes at P will be C'CBEFD'. This is easily seen to be the correct result, as follows: the first argument byte taken from the first operand location is 02_{16}; the function byte at G+X'02' is C'C', and this replaces the first byte at P. Similarly, the fifth and last byte at P is 03_{16}; the byte at G+X'03' is C'D', which is the final byte placed in the string at P. We can use RX instructions to simulate the action of the TR instruction as follows, where it is assumed that the symbols L, B1, D1, B2, and D2 have the same values as in the TR instruction being simulated; for purposes of the example, assume that B1 and B2 have values other than 1 or 2.

```
*           TR     D1(L,B1),D2(B2) IS THE INSTRUCTION BEING SIMULATED
            LA     0,L             SET COUNTER IN R0 TO NUMBER OF BYTES
            SR     1,1             SET FIRST OPERAND INDEX TO 0
            SR     2.2             FOR INDEXING TABLE AT 2ND OPERAND ADDRESS
GETARG      IC     2,D1(1,B1)      GET ARGUMENT BYTE, USE AS XNOEX
            IC     2,D2(2,B2)      REPLACE IT BY FUNCTION BYTE FROM TABLE
            STC    2,D1(1,B1)      STORE IN STRING AT FIRST OPERAND LOCATION
            CA     1,1(0,1)        INCREMENT FIRST OPERAND INDEX BY 1
            BCT    0,GETARG        LOOP UNTIL L ARGUMENT BYTES ARE PROCESSED
```

The full power of the TR instruction can be appreciated if we consider the first example from Section 18, where a character string was to be processed in such a way that all special characters whose EBCDIC representations are numerically less than C'A' are converted to blanks. By setting up an appropriate table, the entire process can be done by one instruction, as follows. The method used to construct the 256-byte table is neither elegant nor general; better ways will be illustrated later.

```
            TR     STR(80),TBL     TRANSLATE ALL SPECIAL CHARACTERS TO BLANK
            - - -
TBL         DC     193C' '         ANYTHING LESS THAN C'A' IS BLANKED
            DC     C'ABCDEFGHI'    LETTERS ARE UNCHANGED.
            DC     7C' '           BLANK THE NON-PRINTING CHARACTERS BETWEEN
            DC     C'JKLMNOPQR'    PRINT LETTERS AS IS
            DC     CL8' '          BLANK OUT NON-PRINTING CHARACTERS
            DC     C'STUVWXYZ'
            DC     6C' '           BLANKS FOR ANYTHING BETWEEN C'Z' AND C'0'
            DC     C'0123456789'   DIGITS PRINT AS IS
            DC     6C' '           TAIL-ENDERS ARE BLANKED TOO
```

As a second example of the use of the TR instruction, suppose we want eventually to print the contents of the fullword at W as 8 hexadecimal digits, and are required to place the 8 EBCDIC characters representing the digits in a string starting at HEX. (We will see later that the UNPK instruction does this more simply.)

```
         L      1,W               GET FULLWORD TO BE CONVERTED
         LA     2,HEX             ADDRESS OF CHARACTER BEING STORED IN R2
         LA     3,8               COUNT IN R 3
CLEAR    SR     0,0               CLEAR R0 FOR SHIFTING
         SLDL   0,4               SHIFT A HEX DIGIT INTO R 0
         STC    0,0(0,2)          STORE IN STRING AT HEX
         LA     2,1(0,2)          INCREMENT CHARACTER ADDRESS BY 1
         BCT    3,CLEAR           BRANCH UNTIL 8 DIGITS ARE STORED
         TR     HEX(8),=C'0123456789ABCDEF'   TRANSLATE TO EBCDIC
```

We can also index in the opposite direction, as follows:

```
         L      0,W               GET FULLWORD TO BE CONVERTED
         LA     2,8               COUNTER AND INDEX IN R2
SHIFT    SRDL   0,4               SHIFT A DIGIT INTO R1
         SRL    1,28              POSITION FOR STORING
         STC    1,HEX-1(2)        STORE IN CHARACTER STRING
         BCT    2,SHIFT           DECREASE INDEX AND SHIFT AGAIN
         TR     HEX,TAB           TRANSLATE DIGITS TO EBCDIC REPRESENTATION
         - - -
HEX      DS     CL8
TAB      DC     C'0123456789ABCDEF'
```

The TRT instruction is identical to TR in the first two steps which were labeled (a) and (b) above; it is quite different in that the accessed byte from the table addressed by the second operand does not replace the argument byte from the first operand string. The accessed function byte is examined instead, and if it is not zero, (1) it is placed in the rightmost byte of R2, (2) the address of the argument byte (which caused a nonzero function byte to be accessed) is placed in the rightmost 24 bits of R1; the remaining bits of R1 and R2 are unchanged, and (3) the operation terminates. The CC is set to indicate the conditions tabulated in Figure 20.5.

CC Setting	Indication
0	All accessed function bytes were zero.
1	Nonzero function byte was accessed before the last argument byte was reached.
2	The nonzero function byte accessed corresponds to the last argument byte.

Figure 20.5 Condition Code Settings for TRT Instruction

As an example suppose we are to scan a string of 80 characters beginning at CARD for punctuation in the form of periods, commas, and apostrophes; when one of them is found, a'branch should be made to P, C, or A respectively, with the address of the character in R1. If none are found, branch to NØPUNCT. First', we will write a program segment using CLI instructions.

```
         LA    1,CARD          INITIALIZE CHARACTER ADDRESS
         LA    2,80            NUMBER OF CHARACTERS TO EXAMINE
TESTP    CLI   0(1),C'.'       COMPARE TO PERIOD
         BC    8,P             BRANCH IF FOUND
         CLI   0(1),C','       COMPARE TO COMMA
         BC    8,C             BRANCH IF FOUND
         CLI   0(1),C''''      COMPARE T O APOSTROPHE
         BC    8,A             BRANCH IF FOUND
         LA    1,1(0,1)        OTHERWISE INCREMENT CHARACTER ADDRESS BY 1
         BCT   2,TESTP         COUNT AND LOOP
         BC    15,NOPUNCT      TAKE THE BRANCH IF NONE FOUND
```

The TRT instruction allows us to do the same processing much more rapidly but at a cost of more memory space.

```
         SR    2,2             CLEAR R2 TO BE USED AS AN INDEX
         TRT   CARD(80),TBL    SCAN FOR PUNCTUATION
         BC    8,NOPUNCT       BRANCH IF NONE FOUND
BRCH     BC    15,BRCH(2)      USE FUNCTION BYTE AS INDEX FUR BRANCH
         BC    15,P            PERIOD
         BC    15,C            COMMA
         BC    15,A            APOSTROPHE
TBL      DC    (C'.')X'00',X'04'
         DC    (C','-C'.'-1)X'00',X'08'
         DC    (C''''-C','-1)X'00',X'0C'
         DC    (255-C'''')X'00'
```

The three nonzero function bytes are located in the positions of the table which correspond to the values of the EBCDIC representations of the characters being sought; the nonzero values are multiples of 4 so they can be used to index the branch instruction at BRCH, which could also have been written BC 15,*(2) . If the conditional branch to NØPUNCT had been omitted, the program could have gone into an infinite loop at BRCH.

To give a final example of the use of several of these SS instructions to process variable-length data, suppose we are given a string of characters at NAMES which contains some unknown number of names separated by commas and terminated with a period. Our first task is to construct a table at LIST of fullword addresses of the first character of each name; the first byte of each address will contain the number of characters in the name (which must therefore be less than 256 letters in length), and when the table is complete the number of names encountered should be stored in the fullword at NBRNMS. To protect against omitted punctuation or other errors, branch to LØNGNAME if no punctuation is found within 256 characters of the start of a name.

```
         SR    3,3           R3 CONTAINS INDEX FOR LIST
         LR    2,3           CLEAR FUNCTION BYTE SWITCH IN R2
         LA    1,NAMES       INITIALIZE SCAN ADDRESS
SCAN     LR    4,1           SAVE INITIAL CHARACTER ADDRESS IN R4
         TRT   0(256,1),TRTB SCAN FOR PERIOD OR COMMA
         BC    8,LONGNAME    BRANCH IF SOMETHING FUNNY HAPPENED
         ST    4,LIST(3)     STORE ADDRESS OF NAME IN LIST
         SR    1,4           COMPUTE NAME LENGTH
         STC   1,LIST(3)     STOKE LENGTH OF NAME IN FIRST BYTE
         LA    3,4(0,3)      INCREMENT LIST ADDRESS
         LA    1,1(4,1)      MOVE ADDRESS TO START OF NEXT NAME
         BCT   2,SCAN        BRANCH IF A COMMA WAS ENCOUNTERED
         SRL   3,2           IF PERIOD, NO BRANCH. COMPUTE AND STORE
         ST    3,NBRNMS      NUMBER OF NAMES FOUND
         - - -
TRTB     DC    (C'.')X'00',X'01'   FUNCTION = 1 FOR PERIOD
         DC    (C','-C'.'-1)X'00',X'02'    FUNCTION = 2 FOR COMMA
         DC    (255-C',')X'00'     ZERO OTHERWISE
         - - -
NAMES    DC    C'BROWN,GREEN,WONKA,OFSTRAND,JONES,SMEDLEY,DOE,APPLE'
         DC    C',DOE,SMITHWICK,SOFTNARD,SMITH,DOELFUL,JONES,LURP.'
FLAG     DS    C
NBRNMS   DS    F
LIST     DS    50F
```

The only unusual feature of the above program segment is in the use of the function byte as a branching switch; if a period is encountered, the contents of R2 will be 00000001_{16} and the BCT instruction will not branch.

Suppose now that the list of addresses is to be sorted so that the names pointed to will be addressed in alphabetical order if the addresses are taken in succession beginning at LIST. We will sort by making repeated passes over the list, making pairwise comparisons among the names and exchanging addresses when they are not in order, and terminating when no exchanges have been made on one full pass aver the list.

```
         L      0,NBRNMS          GET NUMBER OF NAMES
         BCTR   0,0               MINUS 1 TO GIVE NUMBER OF COMPARISONS
START    LR     1,0               INITIALIZE COMPARISON COUNTER
         CA     2,LIST            INITIAL ADDRESS IN LIST OF ADDRESSES
         MVI    FLAG,0            SET FLAG TO SHOW NO EXCHANGES YET
GETADR   L      3,0(0,2)          GET AN ADDRESS FROM THE LIST
         L      4,4(0,2)          AND THE NEXT HIGHER ONE
         CLC    0(256,3),0(4)     COMPARE THE NAMES
         BC     12,NOEXCH         BRANCH IF IN CORRECT ORDER ALREADY
         ST     3,4(0,2)          OTHERWISE EXCHANGE ADDRESSES IN LIST
         ST     4,0(0,2)
         MVI    FLAG,1            INDICATE THAN AN EXCHANGE OCCURRED
NOEXCH   CA     2,4(0,2)          INCREMENT ADDRESS LIST POINTER
         BCT    1,GETADR          JUMP TO 00 ANOTHER COMPARISON
         TM     FLAG,1            NOW, SEE IF ANY EXCHANGES WERE MADE
         BC     1,START           IF YES, BRANCH TO MAKE ANOTHER PASS
```

In doing the name comparison above, we have relied on the fact that the punctuation character at the end of a name has an EBCDIC representation of smaller value than that of letters -- this state of affairs is often expressed by saying that special characters are lower in the EBCDIC collating sequence (the natural ordering implied by the value of the character) than letters. Thus "SMITH, " will compare smaller than "SMITHW", and shorter names will sort ahead of longer ones with the same beginning letters. If two identiaal names are found, the comparison will either branch on equality and no exchange will be made, or the inequality will be determined by whatever the characters in the following name happen to be; the addresses of the identical names will still be adjacent in the sorted list.

Finally, suppose we are required to place the names in alphabetical order in a string beginning at SØRT, again separated by commas and terminated with a period.

```
         L      1,NBRNMS          COUNTER FOR NUMBER OF NAMES'
         LA     2,LIST            R2 CONTAINS ADDRESS OF CURRENT LIST ENTRY
         SR     0.0               R0 WILL CONTAIN LENGTH OF NAME
         LA     4, SORT-1         R4 WILL HAVE ADDRESS OF OUTPUT NAME
ACRGUT   L      3,0(0,2)          GET ADDRESS f-RUM LIST
         XC     0,0(0,2)          GET LENGTH BYTE FROM TABLE
         STC    0,MOVE+1          STORE IN M V C LENGTH FIELD
         LA     4,1(0,4)          MOVE ADDRESS T O START O F NEXT N A M E
MCVE     MVC    0(0,4),0(3)       MOVE NAME INTO OUTPUT A R E A
         AR     490               FORM ADDRESS OF FOLLOWING PUNCTUATION
         MVI    0(4),C','         STORE COMMA AFTER NAME
         LA     2,4(0,2)          INCREMENT ADDRESS O F LIST ITEM
         BCT    1 ,ADROUT         COUNT, BRANCH TO GET NEXT NAME ADDRESS
         MV1    0(4),C'.'         REPLACE LAST COMMA BY A PERIOD
```

In this portion of the program, the punctuation after each name was moved with the name, but a comma was stored in all cases because the period after the last name at the end of the original string was likely to appear in a different position in the final output. Two things should be noted in the MVC instruction: firs%, the explicit length specification of zero is a convenient notation for indicating that the actual length to be used is a variable quantity to be specified at execution time; and second, since the true length of the name is stored in the Length Specification Byte, one additional byte (the punctuation) is moved.

21. THE EXECUTE INSTRUCTION

The execute instruction is one of the most unusual In the System/360 instruction repertoire, eincc it allows the programmer to specify that the execution of another instructian should be performed. It is an RX-type instruction with mnemonic EX which works as follows:

1. The effective address is computed, and the r_1 digit of' the EX inetruction is saved.
2. The instruction at the effective address in memory (called the subject instruction) is placed in the Instruction Register (IR); note that the IA in the PSW is unchanged, and still contains the address of the inetruction following the EX.
3. If the new instruction in the IR is another EX, a program interruption -"occurs; we shall see shortly that there is a good reason for this,
4. If the r_1 digit which was saved is zero, proceed to step 5. Otherwise, the rightmost byte of Rr_1 is ORed into the second byte of the IR; Rr_1 remains unchanged.
5. The (possibly modified) subject instruction in the IR is now decoded and executed as 'though it were the original instruction fetched from memory.

First, consider a few examples of the use of EX in which the r_1 digit is zero, 80 that no ORing takes place in the IR.

(1) Store at C the quantity 2*C(A)-C(B), where A and B are fullwords.

```
         SR    1,1          CLEAR INDEX TOO
         'CA . 2,4          INCREMENT =4, LENGTH OF EXECUTED INSTNS
         LA    3,12         COMPARAND = 12
EX       EX    0,INST(1)    EXECUTE AN INSTRUCTION
        BXLE   1,2,EX       INCREMENT BY 4 AND LOOP
         - - -
 INST    L     0,A          LOAD RO FROM A         (4-BYTE INSTRUCTION)
         AR    0,0          DOUBLE C(RO)           (2-BYTE INSTRUCTION)
         NOPR    0          PADDING INSTRUCTION (2-BYTE INSTRUCTION)
         S     0,B          SUBTRACT c (B)         (4-BYTE INSTRUCTION)
         ST    0,C          STORE RESULT           (4-BYTE INSTRUCTION)
```

This program segment performs a simple four-instruction calculation in a roundabout way; the list of instructions at INST could of course be executed quite independently of the first five instructions, giving the same result much more rapidly. It illustrates a way to execute instructions which are "out-of-line" and not directly in the normal stream of program execution.

(2) Suppose we wish to add three fullword integers stored beginning at Q, and branch to NOERR, ERR1, or ERR2 respectively if 0, 1, or 2 overflows occur.

```
         SR    212            CLEAR OVERFLOW COUNTER
         L     0,Q            GET FIRST INTEGER
         A     0,Q+4          ADD SECOND INTEGER
         BC    14,*+8         BRANCH IF NO OVERFLOW
         LA    2,4            INDICATE ONE OVERFLOW
         A     0,Q+8          ADD THIRD INTEGER
         BC    14,*+8         BRANCH IF NO OVERFLOW
         LA    2,4(0,2)       INDICATE ANOTHER OVERFLOW
         EX    0,*+4(2)       EXECUYE A BRANCH INSTRUCTION
         BC    15,NOERR       O-ERROR BRANCH
         BC    15,ERR1        1-ERROR BRANCH
         BC    15,ERR2        2-ERROR BRANCH
```

In this example, the executed instruction will be one of three unconditional branches: since this results in the IA being changed, the next instruction to be executed 'will be located at the branch address, as expected.

(3) Suppose we are required to place in R6 the address of some quantity in memory, and that the desired address is known only to be the effective address of some RX instruction. To complicate matters, suppose further that the addressing calculation implied by the RX instruction could make use of any register but R14 and R15; we will assume that R15 is currently being used as a base register and R14 contains the address of the RX instruction in question. The technique to be used here will be to construct a LA instruction in memory with the same index, base, and displacement fields as the RX instruction, and then execute that instruction.

```
         MVC   BLDLA(4),0(14)     MOVE RX INSTRUCTION TO WORK AREA
         NI    BLDLA+1,X'0F'      CLEAR OLD R1 DIGIT POSITION
         OI    BLDLA+1,X'60'      SET R1 DIGIT TO 6
         MVI   BLDLA,X'41'        INSERT 'LA' OPCODE INTO INSTRUCTION
         EX    0,BLDLA         EXECUTE THE CONSTRUCTED 'LOAD ADDRESS'
         - -                   R6 NOW CONTAINS THE DESIRED ADDRESS
BLDLA    DS    2H                 4 BYTES ON HALFWORD BOUNDARY
```

The above instruction sequence changes noregisters (even though R0 was available) and illustrate6 a technique that can be used when all register content6 must remain untouched.

More powerful use can be made of the EX instruction when its r_1 digit is not zero, implying modification of a part of the instruction placed in the IR. For example, suppose we wish to move to LINE a message whose address and length are in R8 and R9 respectively, a6 in example (4) of Section 20.

```
         BCTR   9,0             DECREASE LENGTH SPECIFICATION BY 1
         EX     9,MOVE          EXECUTE THE MOVE INSTRUCTION
         - - -
MOVE     MVC    LINE(0),0(8)    EXECUTED INSTRUCTION, LENGTH = 0
```

In this case the Length Specification byte is inserted by ORing into the groper position in the IR, which has been preset to zero by an explicit length specification of zero in the MVC instruction. An advantage of this method is that no modification is made of the instruction in storage.

A6 another example, suppose we wish to branch to YES if the rightmost byte of R3 contains 00011111~.

```
         EX     3,CLI           EXECUTE THE COMPARISON
         BC     8,YES           BRANCH IF EQUALITY IS FOUND
         - - -
CLI      CLI    CHECK,0         EXECUTED INSTRUCTION
CHECK    DC     B'00011111'     COMPARISON QUANTITY
```

This could also be done by the following method, which modifies storage but does not use an EX instruction.

```
         STC    3,TEMP.         STORE THE BYTE TO BE TESTED
         CLI    TEMP,X'1F'      COMPARE TO DESIRED PATTERN
         BC     8,YES           BRANCH IF EQUAL
         - -
TEMP     DS     C
```

(4) Store at T the sum of the contents of registers R0 through R10.

```
         LA     11,10           COUNT IN R11
COOP     EX     11,ADDER        EXECUTE THE ADD INSTRUCTION
         BCT    11,LOOP         DECREASE COUNTER AND REGISTER DIGIT
         ST     0,T             STORE SUM AT T
         - - -
ADDER    AR     0,0             R2 DIGIT MODIFIED IN EXECUTION
```

The r_2 digit of the AR instruction is modified in the IR to contain values which run from 10 down to 1. In practice it is relatively rare that EX instructions are used to modify register specification digits in executed instructions.

As a final example, suppose R5 contains tin unknown integer which specifies a number of bytes to be moved from a string beginning at A to an area whose address is contained in R7.

```
         LTR    5,5              CHECK NUMBER OF BYTES T O  B E MOVED
         BC     12,FINIS         EXIT IFNOT GREATER THAN ZERO
         LA     1,A              R1 CONTAINS ' FROM' ADDRESS
TEST     c      5,=F'256'        SEEIF BYTE COUNT EXCEEDS 256
         BC     4,LAST           IF NOT, DO LAST MOVE
         MVC    0(256,7),0(1)   MOVE 2 5 6 BYTES
         LA     1,256(0,1)       INCREMENT 'FROM' ADDRESS
         LA     7,256(0,7)       INCREMENT 'TO' ADDRESS
         S      5,=F'256'        DECREASE BYTE COUNT B Y 256
         BC     7rTEST           IF NOT ZERO, TEST FOR FINISH
         BC     8,FINIS          I F  C O U N T IS ZERO, ALL DONE
LAST     BCTR   5,0              DECREASE BYTE COUNT BY 1 FOR EXECUTE
         EX     5,LMVC           MOVE LAST PART OF CHARACTER STRING
FINIS - - -
        - - -
LMVC     MVC    0(0,7),0(1)      MOVES LAST 'PART OF BYTE STRING
```

The underlined operands in the instructions listed in Figure 21.1 indicate the modifiable portions of each instruction type when it is the subject instruction of an EX. The last form of operand field entry far SS instructions, in which two Length Specification Digits are provided, will be discussed later.

Type	Operand
RR	$\underline{r_1},\underline{r_2}$
Rx	$\underline{r_1},d_2(\underline{x_2},b_2)$
Rs	$\underline{r_1},\underline{r_3},d_2(b_2)$
	$\underline{r_1},d_2(b_2)$
ST	$d_1(b_1),\underline{I_2}$
SS	$d_1(\underline{L},b_1),d_2(b_2)$
	$d_1(\underline{L_1},b_2),d_2(\underline{L_2},b_2)$

Figure 21.1 Modifiable Portions of Subject Instructions

Two final comments should be made concerning the execute instruction. First, the reason that an EX may not be the subject instruction of an EX (as stated in step 3 of the description above) is that it would be possible for the CPU to remain in a Fetch-Decode Loop (comprising steps 1 through 4) if the EX instruction tried to execute itself, or if a sequence of EX instructions was circular. This is a very awkward situation to get the CPU out of, and is avoided most simply by not allowing the execution of Execute instructions. Second, the EX instruction is sometimes treated as a branch instruction by saying that it causes an unconditional branch to the subject instruction followed by an unconditional branch back to the instruction following the EX, unless the subject instruction is itself a successful branch. This incorrectly describes the contents of the IA, which remains at the address of the instruction following the EX, and obscures the method of modification of the second byte of the subject instruction, which is occasionally described only by stating "the instruction is modified, but remains unchanged in memory'. While the above discussion involving the IR may not describe precisely the method used in a given model of System/360 for handling Execute instructions, it provides a correct description of the effect of the instruction.