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, Form 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 operations 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 computer 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 decimal number 11010, where powers of ten are understood. In fact, the binary 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:

$$26_{10} = 110102, \quad 110 = 1_2,$$

$$10_{10} = 10102, \quad 1000_2 = 8_{10}.$$
As the decimal numbers being represented become larger, the number of binary digits required becomes larger also.

Thus,

\[ 999_{10} = 111100111_{2}. \]

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:

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

TABLE I.
Hexadecimal, Decimal, and Binary Digits
We will call the base sixteen digits in the third column hexadecimal 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:

\[26_{10} = 110102 = {1A}_{16}, \quad 26_{16} = 1001102 = 38_{10}, \quad 1_{10} = 1_2 = 1_{16}, \quad 10_{10} = 10102 = 8_{10} = 8_{16}, \quad 100_{10} = 64_{16} = 11001002.\]

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:

\[\text{D5B}_{16} = 1101 0101 10112, \quad \text{(hexadecimal to binary)}\]
\[11 1110 1002 = 3E8_{16}. \quad \text{(binary to hexadecimal)}\]

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

\[11_{16} = 100012 \text{ rather than 0001 00012.}\]

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 base D to the representation in a new base B:

\[ X = \sum_{k=0}^{m} b_k B^k = b_0 B^0 + b_1 B^1 + b_2 B^2 + \ldots + b_m B^m. \]

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 \( b_k \) in the new representation, the following scheme is used.

Divide \( X \) by \( B \); save the quotient, end the remainder is \( b_0 \). That this is so can be seen from the definition of the quotient and remainder:

\[ X = \text{Remainder} + B \times \text{Quotient} = b_0 + B \times [b_1 + b_2 B + b_3 B^2 + \ldots + b_m B^{m-1}]. \]

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 \( b_0, b_1, \ldots, b_m \); note that they were obtained in order of increasing significance.

**Examples**

1. Convert \( 19_{10} \) to base 2.

   \[
   \begin{array}{c|c|c|c|c|c}
   2 & 9 & 2 & 4 & 2 & 1 \\
   \hline
   b_0 & = & 1 & b_1 & = & 1 \\
   b_2 & = & 0 & b_3 & = & 0 \\
   b_4 & = & 1 \\
   \end{array}
   \]

   Hence, \( 19_{10} = 10011_2 \).

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

   \[
   \begin{array}{c|c|c|c|c|c}
   16 & 1000 & 16 & 62 & 16 & 3 \\
   992 & 62 & 48 & 3 & 0 \\
   b_0 & = 8 & b_1 = 14 \text{ or } E_{16} & b_2 = 3 \\
   \end{array}
   \]

   Hence \( 1000_{10} = 3E_8_{16} \).
3. Convert $627_{10}$ to base 9.

$$\begin{array}{c|c|c|}
69 & 7 & 0 \\
\hline
9)627 & 9)69 & 9)7 \\
63 & 63 & 0 \\
\hline
b_0 = 6 & b_1 = 6 & b_2 = 7 \\
\end{array}$$

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.)

$$\begin{array}{c|c|c|c|}
108 & 13 & 1 & 0 \\
\hline
7)766 & 7)108 & 7)13 & 7)1 \\
762 & 103 & 1 & 0 \\
\hline
b_0 = 4 & b_1 = 5 & b_2 = 5 & b_3 = 1 \\
\end{array}$$

Thus $766_9 = 1554_7$.

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

$$\begin{array}{c|c|c|c|}
89 & 12 & 1 & 0 \\
\hline
7)627 & 7)89 & 7)12 & 7)1 \\
623 & 84 & 1 & 0 \\
\hline
b_0 = 4 & b_1 = 5 & b_2 = 7 & b_3 = 1 \\
\end{array}$$

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,

$$\begin{array}{c|c|c|}
43 & 2 & 0 \\
\hline
20)1413 & 20)43 & 20)2 \\
130 & 40 & 0 \\
\hline
b_0 = 3 & b_1 = 3 & b_2 = 2 \\
\end{array}$$

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

2-5
6. Convert \( 3B8_{16} \) to base 10. In this case it is usually simplest to use the positional notation used earlier:

\[
3B8_{16} = 3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0,
\]

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

\[
3B8_{16} = 3 \times 256 + 14 \times 16 + 8 = 768 + 224 + 8 = 1000_{10}.
\]

This type of conversion is considerably simplified 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.
<table>
<thead>
<tr>
<th>Hex Digit</th>
<th>\times 1</th>
<th>\times 16^1</th>
<th>\times 16^2</th>
<th>\times 16^3</th>
<th>\times 16^4</th>
<th>\times 16^5</th>
<th>\times 16^6</th>
<th>\times 16^7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>16</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
<td>1,048,576</td>
<td>16,777,216</td>
<td>268,435,456</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>32</td>
<td>512</td>
<td>8,192</td>
<td>131,072</td>
<td>2,097,152</td>
<td>33,554,432</td>
<td>536,870,912</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>48</td>
<td>768</td>
<td>12,288</td>
<td>196,608</td>
<td>3,145,728</td>
<td>50,331,648</td>
<td>805,306,368</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>64</td>
<td>1024</td>
<td>16,384</td>
<td>262,144</td>
<td>4,194,304</td>
<td>67,108,864</td>
<td>1,073,741,824</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>80</td>
<td>1280</td>
<td>20,480</td>
<td>327,680</td>
<td>5,242,880</td>
<td>83,886,080</td>
<td>1,342,177,280</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>96</td>
<td>1536</td>
<td>24,576</td>
<td>393,216</td>
<td>6,291,456</td>
<td>100,663,296</td>
<td>1,610,612,736</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>112</td>
<td>1792</td>
<td>28,672</td>
<td>458,752</td>
<td>7,340,032</td>
<td>117,440,512</td>
<td>1,879,048,192</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>128</td>
<td>2048</td>
<td>32,768</td>
<td>524,288</td>
<td>8,388,608</td>
<td>134,217,728</td>
<td>2,147,483,648</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>144</td>
<td>2304</td>
<td>36,864</td>
<td>589,824</td>
<td>9,437,184</td>
<td>150,994,944</td>
<td>2,415,919,104</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>176</td>
<td>2816</td>
<td>45,056</td>
<td>720,896</td>
<td>11,534,336</td>
<td>184,549,376</td>
<td>2,952,799,016</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>192</td>
<td>3072</td>
<td>49,152</td>
<td>786,432</td>
<td>12,582,912</td>
<td>201,326,592</td>
<td>3,221,225,472</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>208</td>
<td>3328</td>
<td>53,248</td>
<td>851,968</td>
<td>13,631,488</td>
<td>218,103,808</td>
<td>3,489,660,928</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>224</td>
<td>3584</td>
<td>57,344</td>
<td>917,504</td>
<td>14,680,064</td>
<td>234,881,024</td>
<td>3,758,096,384</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>240</td>
<td>3840</td>
<td>61,440</td>
<td>983,040</td>
<td>15,728,640</td>
<td>251,658,240</td>
<td>4,026,531,840</td>
</tr>
</tbody>
</table>

**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](image)

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.
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.

```
0 1 2 3 4 5 6 7
1 1 0 1 0 0 1 0
```

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 address or location in the memory unit. The memory may be thought of as a linear string of bytes arranged in order of increasing addresses.

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 leftmost 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 halfword, fullword, and doubleword data, respectively.
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](image)

![Figure 3.7 General Purpose Registers](image)
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 $R_n$ to refer to general register $n$, so that $R_0$ means register 0, $R_{14}$ 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.

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.

<table>
<thead>
<tr>
<th>System Mask</th>
<th>Key</th>
<th>AMWP</th>
<th>Interruption Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>11 12 15 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ILC</th>
<th>CC</th>
<th>Program Mask</th>
<th>Instruction Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>33</td>
<td>34 35 36</td>
<td>39 40 63</td>
</tr>
</tbody>
</table>

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](image)

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.

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Register Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 8 15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Register Specification</th>
<th>Addressing Syllable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 8 15 16</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Register Specification</th>
<th>Addressing Syllable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 8 15 16</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Immediate Operand</th>
<th>Addressing Syllable</th>
<th>Addressing Syllable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Register Specification</th>
<th>Addressing Syllable</th>
<th>Addressing Syllable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 8 15 16 31 32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \textit{xxxxxx} is meant to indicate the remaining six bits of the eight-bit operation code.

<table>
<thead>
<tr>
<th>00xxxxxx</th>
<th>01xxxxxx</th>
<th>10xxxxxx</th>
<th>11xxxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>Rx</td>
<td>RS,SI</td>
<td>ss</td>
</tr>
</tbody>
</table>

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 pairs 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 \text{ILC}) what instruction caused the error. (This is of course precisely what \textit{is} done; we will note for now that if the \text{ILC} were not saved, it would not be possible to determine the exact location of the offending instruction, since the location of the next instruction to be executed is what appears in the PSW and the length of the bad instruction is \textit{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](image)

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 RO 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](image)

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 $10110001011010101$ in binary (or $\text{B2D5}$ in hex) and suppose that the contents of general purpose register $1110$ is $110010111010010000101111$ in binary (or $\text{C7B90AF}$ in hex). Then the effective address of the instruction is (giving both binary and hex):

   $$
   \begin{align*}
   \text{0000} & \quad \text{0000} \quad \text{0000} \quad \text{0010} \quad \text{1101} \quad \text{0101} \\
   + \quad \text{0011} & \quad \text{1110} \quad \text{1001} \quad \text{0000} \quad \text{1010} \quad \text{1111} \\
   \text{0011} & \quad \text{1110} \quad \text{1001} \quad \text{0011} \quad \text{1000} \quad \text{01002} \\
   \text{0002DF} & \quad \text{displacement} \\
   \text{3E90AF} & \quad \text{base (from R11)} \\
   \text{3E9B8416} & \quad \text{16-bit addend (1111)}
   \end{align*}
   $$
2. Suppose the addressing syllable of the same instruction is 0468. Then the effective address is 00468₁₆, since RO cannot be used for a base.

3. Suppose an RX-type instruction is 130A468₁₆, and that the contents of R7 is 12345678₁₆ and the contents of R10 is FEDCBA98₁₆. (Note that the base register specification digit, namely 7₁₆, means that R7 will be used. The instruction chosen for this and the next two examples would, if executed by the CPU, cause the contents of the byte at the memory location given by the effective address to replace the rightmost byte of RO.) Then the effective address is

\[
\begin{align*}
0000 & \ 0000 \ 0000 \ 0100 \ 0110 \ 1000 & + & \text{000468 displacement} \\
0011 & \ 0100 \ 0101 \ 0110 \ 0111 \ 1000 & + & \text{345678 base (from R7)} \\
1101 & \ 1100 \ 1011 \ 1010 \ 1001 \ 1000 & + & \text{DCBA98 index (from R10)} \\
+ & \text{0001 0001 0011 0111 0112} \\
\end{align*}
\]

(The carry out the left end is ignored.)

4. Suppose an RX-type instruction is 1500468₁₆ and that the contents of register 7 is as in example 3. Then the effective address is

\[
\begin{align*}
0000 & \ 0000 \ 0000 \ 0100 \ 0110 \ 1000 & + & \text{000468 displacement} \\
0011 & \ 0100 \ 0101 \ 0111 \ 1110 \ 0000 & + & \text{345678 base} \\
\end{align*}
\]

5. Suppose an RX-type instruction is 13070468₁₆ and that the contents of register 7 is as in example 3. Then the effective address is

\[
\begin{align*}
0000 & \ 0000 \ 0000 \ 0100 \ 0110 \ 1000 & + & \text{000468 displacement} \\
0000 & \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 & + & \text{000468 base} \\
0011 & \ 0100 \ 0111 \ 0011 \ 0110 \ 1100 & + & \text{345678 index} \\
-0011 & \ 0100 \ 1010 \ 0111 \ 0000 \ 0002 & + & \text{345AB0₁₆ effective address} \\
\end{align*}
\]

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 $2^{16} = 10000_{16} = 65536$ 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 $[58\,40\,0\,123]$ 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 \(00000001_{16}\),
- \(130\) is represented by \(00000082_{16}\),
- \(2^{31}\) is represented by \(80000000_{16}\),
- \(2^{32}-1\) is represented by \(FFFFFFFFFF_{16}\),
- \(2^{32}+1\) 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 \(b_i\) is

\[
X = \sum_{i=0}^{31} b_i 2^i. \quad \text{(logical representation)}
\]
This representation is the most common 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 set 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 \( \sum b_i 2^i \) 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

\[
x = -2^{31} b_{31} + \sum_{i=0}^{30} b_i 2^i.
\]

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 \( X \):

\[
X = \sum_{i=0}^{30} b_i 2^i.
\]
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

\[
X = 2^{32} + x \pmod{2^{32}}.
\]

(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 described, 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 complement 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) One's Complement: 1111 1101
   3) Add one: 1111 1110

2. Find the two's complement of \(75\).
   1) Representation of \(75\): 0010 1011
   2) One's Complement: 1101 0100
   3) Add one: 1101 0101

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 \(1111 1110\).
   1) One's Complement: 0000 0001
   2) Add one: 0000 0010

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 \(0000 0000\).
   1) One's Complement: 1111 1111
   2) Add one: 1111 1110

(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 0000\text{2}.
   1) One's Complement: 0111 1111
   2) Add one: \[ +1 \]
      \[
      \begin{array}{c}
      1000 0000 \text{2} \\
      \end{array}
      \]
      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 overflow 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.
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 0111₂ and 1111 1111 1111 0111₂, 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.

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>32-bit Two's Complement Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0001₆</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001₁₆</td>
</tr>
<tr>
<td>256</td>
<td>0000 0100₁₆</td>
</tr>
<tr>
<td>5000</td>
<td>0000 1388₁₆</td>
</tr>
<tr>
<td>2147483647(2^{31}-1)</td>
<td>7FFF FFFF₁₆</td>
</tr>
<tr>
<td>-2147483647(-2^{31})</td>
<td>8000 0000₁₆</td>
</tr>
<tr>
<td>2147483647(-2^{31}+1)</td>
<td>8000 0000₁₆</td>
</tr>
<tr>
<td>-5000</td>
<td>FFFF EC78₁₆</td>
</tr>
<tr>
<td>-256</td>
<td>FFFF FFF0₁₆</td>
</tr>
<tr>
<td>-2</td>
<td>FFFF FFFE₁₆</td>
</tr>
<tr>
<td>-1</td>
<td>FFFF FFFF₁₆</td>
</tr>
</tbody>
</table>

Figure 6.1 Examples of Two's Complement Representation

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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0, carry 1</td>
</tr>
</tbody>
</table>

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:  
   \[ \begin{array}{c} 0000 \ 0101 \\ -0000 \ 0011 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 0000 \ 0101 \\ +1111 \ 1101 \end{array} \]  
   (carry lost)  
   \[ \begin{array}{c} 0000 \ 0010 \end{array} \]  
   = \[ 2_{10} \]

2. 3-5:  
   \[ \begin{array}{c} 0000 \ 0011 \\ -0000 \ 0101 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 0000 \ 0011 \\ +1111 \ 1011 \end{array} \]  
   (no carry)  
   \[ \begin{array}{c} 1111 \ 1110 \end{array} \]  
   = \[ -2_{10} \]

3. 25-( -17):  
   \[ \begin{array}{c} 0001 \ 1001 \\ -1110 \ 1111 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 0001 \ 1001 \\ +0001 \ 0001 \end{array} \]  
   (no carry)  
   \[ \begin{array}{c} 0010 \ 1010 \end{array} \]  
   = \[ 42_{10} \]

4. (-17)-25:  
   \[ \begin{array}{c} 1110 \ 1111 \\ -0001\ 1001 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 1110 \ 1111 \\ +1110 \ 0111 \end{array} \]  
   (carry lost)  
   \[ \begin{array}{c} 1101 \ 0110 \end{array} \]  
   = \[ -4_{210} \]

5. -17-( -25):  
   \[ \begin{array}{c} 1110 \ 1111 \\ -1110 \ 0111 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 1110 \ 1111 \\ +0001 \ 1001 \end{array} \]  
   (carry lost)  
   \[ \begin{array}{c} 0000 \ 1000 \end{array} \]  
   = \[ 8_{10} \]

6. 67-( -93):  
   \[ \begin{array}{c} 0100 \ 0011 \\ -1010 \ 0011 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 0100 \ 0011 \\ +0101 \ 1101 \end{array} \]  
   (no carry)  
   \[ \begin{array}{c} 1010 \ 0000 \end{array} \]  
   = \[ -96_{10} \] (overflow)

7. (-93)-67:  
   \[ \begin{array}{c} 1010 \ 0011 \\ -0100 \ 0011 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 1010 \ 0011 \\ +1011 \ 1101 \end{array} \]  
   (carry lost)  
   \[ \begin{array}{c} 0110 \ 0000 \end{array} \]  
   = \[ 96_{10} \] (overflow)

8. -128-( -93):  
   \[ \begin{array}{c} 1000 \ 0000 \\ -1010 \ 0011 \end{array} \]  
   becomes  
   \[ \begin{array}{c} 1000 \ 0000 \\ +0101 \ 1101 \end{array} \]  
   (no carry)  
   \[ \begin{array}{c} 1101 \ 1101 \end{array} \]  
   = \[ -35_{10} \]
9. 3-3: \[ \begin{array}{c}
0000 0011 \\
-0000 0012 \\
\hline
\end{array} \quad \text{becomes} \quad \begin{array}{c}
0000 0011 \\
+1111 1101 \\
\hline
\end{array} \\
\text{(carry lost)} \quad \begin{array}{c}
0000 0000 \\
= 0 \\
\end{array}\]

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)): \]
\[ \begin{array}{c}
0000 0001 \\
-1000 0000 \\
\hline
\end{array} \quad \text{becomes} \quad \begin{array}{c}
0000 0001 \\
+1000 0000 \\
\hline
\end{array} \\
\text{(no carry)} \quad \begin{array}{c}
1000 0001 \\
= \text{no overflow found} \\
\end{array}\]

11. \((-1-(-128)): \]
\[ \begin{array}{c}
1111 1111 \\
-1000 0000 \\
\hline
\end{array} \quad \text{becomes} \quad \begin{array}{c}
1111 1111 \\
+1000 0000 \\
\hline
\end{array} \\
\text{(carry lost)} \quad \begin{array}{c}
0111 1111 \\
\text{(overflow indicated)} \\
\end{array}\]

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.
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) $X = 2^{32} + x \pmod{2^{32}}$, 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 no 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

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>126</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplicand</td>
<td>x 213</td>
</tr>
<tr>
<td>Partial Products</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>126</td>
</tr>
<tr>
<td>Product</td>
<td>26838</td>
</tr>
</tbody>
</table>
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 multipli-
cand 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 end
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 binary 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 \(01001_2\) to obtain a 10-bit product in a double-length register. Then the sequence of steps shown below indicates the method.

<table>
<thead>
<tr>
<th>Step</th>
<th>Rightmost Bit</th>
<th>Action</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Add</td>
<td>(00110)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>No Add</td>
<td>(00011)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>No Add</td>
<td>(00001)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Add</td>
<td>(00110)</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>No Add</td>
<td>(00011)</td>
</tr>
</tbody>
</table>

Multiplier (in separate register) \(00110\)
Multiplicand in right half of double-length register \(01001\)

| Initialize | \(00110\) | \(01001\) |
| Shift right 1 | \(00011\) | \(00100\) |
| Step 2: rightmost bit = 0 | \(00001\) | \(10010\) |
| Shift right 1 | \(00000\) | \(11001\) |
| Step 4: rightmost bit = 1 | \(00110\) | \(11001\) |
| Shift right 1 | \(00011\) | \(01100\) |
| Step 5: rightmost bit = 0 | \(00001\) | \(10110\) |

Final product \(= 110110_2 = 54_{10}\)

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 complement 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 less 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 = \(10^3 \times 456 - 1\), 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

\[ 762 \times 213 + 537 = 162843. \]

On a machine, the process is almost identical. Using the above scheme of decimal registers, the division works as follows:
High-order part of dividend smaller than divisor, 
division may proceed.

Shift dividend left once; save leftmost digit in an 
"overflow digit" position. Since dividend $\geq$ divisor, 
subtract, and count up at right end.

dividend $\geq$ divisor; subtract again

dividend $< \text{divisor}$; no subtraction

shift dividend left again

dividend $\geq$ divisor; subtract and count up

dividend $< \text{divisor}$; no subtraction

shift left for last time-

dividend $\geq$ divisor; subtract

subtract and count up by 1

dividend $\geq$ divisor; subtract

subtract and count

dividend now $< \text{divisor}$; 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\textsubscript{10}, and let the divisor be 00110\textsubscript{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 31 times.)

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 001012 = 510 in the left half, and the quotient 010012 = 910 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 afterward 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.

![Figure 9.1 Simplified Schematic of Assembler Processing](image-url)
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.

![Diagram](image)

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 module 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.

![Diagram of program loading and execution]

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 \texttt{4316} 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 \texttt{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 \texttt{15} for some use, we can use the decimal number \texttt{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.
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.)

![Figure 9.5 A Machine Instruction Statement](image)

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.
Finally, an example of a macro-instruction statement in which only the operation field entry appears is given below.
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 \(2^4\) 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 strings and 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.
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  AP@PLEXY
JAMES  KBFØ  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 ØØ  (no blanks allowed)
235#  (does not start with a letter)
CHARACTER  (too many characters)
/ T£N*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 relocatable or absolute (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 `LOAD` is the name of the location at which the instruction (whose mnemonic is LR) begins. In the machine instruction statement

```
GETC$NST     L     0,4(2,7)
```

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

```
TEN          DC     P'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 relative origin (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 his 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 assumed 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 $7B_{16}$ when the statement given in the first example above was scanned. Then the value of the symbol GETC#NST would be $7B_{16}$, 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 $7B_{16}$ 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 \( \text{4EC}_{16} \) respectively, we could write

(the details of writing the operand field will be discussed in the next section)

\[
L_{0,x} \cdot \text{4EC'(0,6)}
\]

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 convenient means 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 "EXE" 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 r1, s2, 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 0016 and 3F16 (in the column 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).
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, ",r1" is "7" and "r2" is "3". In fact, the quantities r1 and r2 must simply be absolute (i.e., non-relocatable) expressions 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. 11.2), as in the following figure.

![Figure 11.1 RR Instruction Showing Register Specification Digits](image)

The subscripts on the quantities "r1" and "r2" 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 r1 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 instruction which would be assembled from the statement as given would be $1373_{16}$ in hexadecimal.

For the case of the SRM instruction the digit labeled r2 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 \( \text{SRM 14} \) and \( \text{svc 255} \) 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 terms (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 \( \text{GETCONST} + X'4A' \) 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 expression in the operand field. Thus the statement

\begin{verbatim}
ABS425 EQU 425
\end{verbatim}

serves to define an absolute symbol with value 425. (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, so 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 08/360 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 procedural languages, 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 \( 5 \times 2/4 \) means \((5 \times 2)/4\), not \(5 \times (2/4)\).

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 \( 5 \times 2/4 \) has the value 2, and \(5 \times (2/4)\) 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 paired (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

\[
\text{LOAD LR C145'(79X'2A36')ABS425*E'11111'-235,18/(Q-Q)+3}
\]

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

\[
\begin{align*}
R7 & \text{ EQU 7} \\
R3 & \text{ EQU 3} \\
\text{LOAD LR} & \text{ R7, R3}
\end{align*}
\]
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.

\[
\begin{align*}
Z\text{\textregistered}CH & \text{ EQU } 3 & R7 & \text{ EQU } 3 \\
Z\text{ILCH} & \text{ EQU } 7 & R3 & \text{ EQU } 7 \\
L\#AD & \text{ LR } Z\text{ILCH},Z\text{\textregistered}CH & L\#AD & \text{ LR } R3,R7 \\
\end{align*}
\]

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 RO; the following three statements (whose properties will be discussed later) would suffice:

\[
\begin{align*}
\text{LR } 0,2 & \text{ MOVE CONTENTS OF R2 TO RO} \\
\text{AR } 0,14 & \text{ ADD CONTENTS OF R14} \\
\text{SR } 0,9 & \text{ SUBTRACT CONTENTS OF R9} \\
\end{align*}
\]

**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.)
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

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

**TABLE V.**
RX Instructions
forms for the operand field entry immediately, we note first that it is of
the general form

\[ r_1, \text{<address specification>} \]

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.

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IC</td>
<td>0,X'468'(10,7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IC</td>
<td>0,1128(0,7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IC</td>
<td>0,1128(7,0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 forms in which the second operand of an RX instruction may be written; these are given in Fig. 11.4 below.

<table>
<thead>
<tr>
<th>Explicit Address</th>
<th>Implied Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_2(x_2,b_2) )</td>
<td>( s_2(x_2) )</td>
</tr>
<tr>
<td>( d_2(x_2) )</td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( d_2(b_2) )</td>
<td></td>
</tr>
</tbody>
</table>

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 or equal to 4095\(_{10} = \text{FFF}_{16} \). Note that the second and third forms of explicit address implicitly specify \( b_2 = 0 \) and \( x_2 = 0 \), respectively, 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 \( 2^{84} \). This means that we may write instructions such as \( \text{L O, ANSWER} \) and leave it to the Assembler to compute the proper base and displacement; 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 \( \text{IC O, BYTE} \) is translated into \( \text{43 00 74 68} \) as in Fig. 11.3. Then if the index register to be used is R10, the instruction \( \text{IC O, BYTE(10)} \) would be translated into \( \text{43 0A 74 68} \).

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 most 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 LOAD FROM N INTO RO
A O,N ADD INTEGER CONSTANT 1
ST O,N STORE RESULT BACK AT N

where it is assumed that an addressable fullword area named ONE which contains the integer constant 41 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 instructions 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, SII, TII, HI, TCH, and "Diagnose", for which there is no mnemonic.

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

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 $\text{MVI .FLAG,0}$ which would cause the byte named $\text{FLAG}$ (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

\[
\begin{array}{|c|c|c|}
\hline
\text{Opcode (hex)} & \text{Mnemonic} & \text{Instruction} \\
\hline
D1 & MVN & \text{Move Numeric} \\
D2 & MVC & \text{Move} \\
D3 & MVZ & \text{Move Zone} \\
D4 & NC & \text{Logical AND} \\
D5 & CLC & \text{Compare Logical} \\
D6 & \phi C & \text{Logical \&R} \\
D7 & XC & \text{Exclusive \&R} \\
D8 & TR & \text{Translate} \\
D9 & TRT & \text{Translate and Test} \\
DE & ED & \text{Edit} \\
DF & EIMK & \text{Edit and Mark} \\
\hline
\end{array}
\]  

\[
\begin{array}{|c|c|c|}
\hline
\text{Opcode (hex)} & \text{Mnemonic} & \text{Instruction} \\
\hline
F1 & MVO & \text{Move with Offset} \\
F2 & PACK & \text{Pack} \\
F3 & UNPK & \text{Unpack} \\
F8 & ZAP & \text{Zero and Add} \\
F9 & CP & \text{Compare} \\
FA & AP & \text{Add} \\
FB & SP & \text{Subtract} \\
FC & MP & \text{Multiply} \\
FD & DP & \text{Divide} \\
\hline
\end{array}
\]

**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 $\text{AREA}$ to an area whose first byte is named $\text{FIELD}$ is $\text{MVC \: FIELD(5),AREA}$.

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.

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

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 r1,0 (so that the second operand register specification digit r2 is zero). The effect of this instruction when executed is to replace the contents of general register r1 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 500016.

<table>
<thead>
<tr>
<th>Location</th>
<th>Name</th>
<th>Operation</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>BALR</td>
<td>6,0</td>
<td></td>
</tr>
<tr>
<td>5002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,N LOAD CONTENTS OF N INTO R2</td>
</tr>
<tr>
<td>5006</td>
<td>A</td>
<td>2,ONE ADD CONTENTS OF ONE</td>
<td></td>
</tr>
<tr>
<td>500A</td>
<td>ST</td>
<td>2,N STORE CONTENTS OF R2 INTO N</td>
<td></td>
</tr>
<tr>
<td>5024</td>
<td>N</td>
<td>DC</td>
<td>F'8' FULLWORD INTEGER 8</td>
</tr>
<tr>
<td>5028</td>
<td>ONE</td>
<td>DC</td>
<td>F'S FULLWORD INTEGER1</td>
</tr>
</tbody>
</table>

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 "A" 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 \([58\ 0206022]\). Then when the instruction is executed, the computation of the effective address yields \( 022 + 005002 = 005204 \), 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembled Contents</th>
<th>Original Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>0560</td>
<td>BALR 6,0</td>
</tr>
<tr>
<td>5002</td>
<td>5A206026</td>
<td>BEGIN L 2,ME 2,N</td>
</tr>
<tr>
<td>500A</td>
<td>50206022</td>
<td>ST 2, N</td>
</tr>
<tr>
<td>5024</td>
<td>00000008</td>
<td>N DC F'8'</td>
</tr>
<tr>
<td>5028</td>
<td>00000001</td>
<td>ANE DC F'1'</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$84E8$</td>
<td>BALR 6, 0</td>
</tr>
<tr>
<td>$84EA$</td>
<td>BEGIN L 2, N</td>
</tr>
<tr>
<td>$84EE$</td>
<td>A 2, $\phi$NE</td>
</tr>
<tr>
<td>$84F2$</td>
<td>ST 2, N</td>
</tr>
<tr>
<td>$850C$</td>
<td>--- the same 22 bytes of odds and ends ---</td>
</tr>
<tr>
<td>$8510$</td>
<td>$\phi$NE DC F'8'</td>
</tr>
</tbody>
</table>

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 $850C - 84EA = 022_{16}$. Thus the assembled program would appear as in the figure below.

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembled Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$84E8$</td>
<td>0560</td>
</tr>
<tr>
<td>$84EA$</td>
<td>58206022</td>
</tr>
<tr>
<td>$84EE$</td>
<td>5A206026</td>
</tr>
<tr>
<td>$84F2$</td>
<td>50206022</td>
</tr>
<tr>
<td>$850C$</td>
<td>00000008</td>
</tr>
<tr>
<td>$8510$</td>
<td>00000001</td>
</tr>
</tbody>
</table>

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 $\phi$NE 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 memory 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 500016; 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 500016, and that it would be found in R6 at execution time. Similarly, in the second example, the relative origin would be 84EA16, and the contents of R6 that the Assembler should assume in order to calculate the correct displacements would be 84EA16. 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

\[
\text{USING } s, r_1
\]

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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BALR</td>
<td>6,0</td>
</tr>
<tr>
<td>USING</td>
<td>BEGIN,6</td>
</tr>
<tr>
<td>BEGIN</td>
<td>L</td>
</tr>
<tr>
<td>A</td>
<td>2,NE</td>
</tr>
<tr>
<td>ST</td>
<td>2,N</td>
</tr>
<tr>
<td>N</td>
<td>DC</td>
</tr>
<tr>
<td>NE</td>
<td>DC</td>
</tr>
</tbody>
</table>

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 NE are $2^{4_{16}}$ and $2^{8_{16}}$ respectively. To compute the addressing syllable of the ST instruction, the Assembler need only note that the difference between the value of the symbol N and the value that the USING instruction specifies will be present in R6, is $2^{4} - 2 = 2^{2_{16}}$; 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 L0AD CONTENTS OF N INT0 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembled Contents</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0560</td>
<td>BALR 6,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USING</td>
</tr>
<tr>
<td>2</td>
<td>58606022</td>
<td>BEGIN L</td>
</tr>
<tr>
<td>6</td>
<td>5A206026</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>50206022</td>
<td>ST</td>
</tr>
<tr>
<td>24</td>
<td>00000008</td>
<td>N</td>
</tr>
<tr>
<td>28</td>
<td>00000001</td>
<td>0NE</td>
</tr>
</tbody>
</table>

Figure 12.6 Sample Program Segment with Erroneous Statement

It is apparent that this program will assemble correctly, as did the one in Figure 12.5, since all quantities are properly specified. However, at execution time, 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 $xx009002_{16}$. 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 $00000081_{16}$, and not $xx005002_{16}$. When the next 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 $'8'$ 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 $48_{10}$)
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'18' 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 location 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 \(\text{DC } F'8'\) 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 \(L_n\)

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

\[
\text{DC } \text{FL3'8'} \quad \text{and} \quad \text{DC } \text{FL(2*4-5)'8'}
\]

would both cause the constant \(0000008_{16}\) 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 FL'4'8'
```

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 Ic is, $12E_{16}$ when each of the following statements is encountered.

```
IMPLIED DC F'8'
```

```
EXPLICIT DC FL'4'8'
```

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 9.3 the length attribute of the symbol L@AD 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 FFF616 to be assembled 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 LONG CHARACTER CONSTANT'
DIGITS      DC X'8462AFCB975310'
```

Figure 13.2 Examples of Character and Hexadecimal Constants

In the discussion of data converted according to types F and H it 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 bytes 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 Assembler. The rules used to determine the final values of the constants are given below.
Constant Assembled too Large Value ~'65537' 000116 FLL'-300' D416 CL3'SMITH' E2D40916 XL2'56789' 6789~

Constant too Small Value H'2' 000216 FLL'-6' FA16 CL3'S' E2404016 X'56789' 05678916 B'10l' 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 4016) 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 at the left end of the constants to fill out the partially-specified byte.

Before discussing literals and the definition of storage areas, we 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 \texttt{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 A(\texttt{H4(10)}) and F'\texttt{H4:10}'. 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 \texttt{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 \texttt{DC F'8,8,8'} which, as was indicated earlier, is equivalent to \texttt{DC F'8'} and \texttt{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 \texttt{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, \texttt{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.

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

(* 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 optional. 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:

```
STORAGE DS 100F
```

which reserves one hundred aligned fullwords and assigns to the symbol
STORAGE the location of the first. Note also that the two statements

\[ \text{AREA1 DS 80C} \quad \text{and} \quad \text{AREA2 DS CL80} \]

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

\[ \text{WORD DS OF C'WORD'} \quad \text{and} \quad \text{WORD DC C'WORD'} \]

both serve to define a four-byte character constant on a fullword boundary addressed by the symbol \( \text{WORD} \), which would not in general have been the case if \( \text{DC C'WORD'} \) or \( \text{DC CL4'WORD'} \) had been specified. Note that \( \text{DC A(C'WORD')} \) is incorrect: because the operand in parentheses must be an expression, and because \( \text{C'WORD'} \) 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 \( \text{constant} \) 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 \( j \) are both defined by DC instructions, but it is implicit in the use of the symbol "\( j \)" 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 \( \text{EIGHT DC \ 'F'5'} \) 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.

\[
\begin{align*}
=F'1' & \quad =C'=\text{LONGLITERAL'} & \quad =BL2'111101' \\
=H'1' & \quad =\text{CL7'}.BLANK' & \quad =X'765432A' \\
=A(1) & \quad =F'1,2,3,4' & \quad =AL3(5,X'D7'/C'.')
\end{align*}
\]

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 \( \text{IC \ '0'\ F'1'+3} \) is illegal);
2. an instruction may not store or modify a literal (thus \( \text{ST \ '7'\ F'1'} \) is illegal);
3. a literal may not be specified in an address constant (about which more later) (so that \( \text{A(\ 'F'1'} \) 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 \( \text{L \ '2'\ X'2B'} \) 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 $r_l$" where $r_l$ indicates the value supplied for an operand in the operand field entry of a machine instruction statement, we will use the abbreviation "$R_r_l$" instead. Thus if $r_l$ has the value 5, the register being referred to is $R_5$.

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

\[ L \quad r_1, d_2(x_2, b_2) \]

causes the fullword second operand to replace the contents of $R_{r_1}$. The original contents of $R_{r_1}$ 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

\[
L \quad 9, = F'0'
\]

and to set it to the maximum negative number,

\[
L \quad 9, = F'-2147483648'
\]

would suffice.

The inverse operation ST is written explicitly as

\[
ST \quad r1, d_2(x_2, b_2)
\]

and causes the contents of Rr1 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

\[
\begin{align*}
L \quad & 0, A \\
ST \quad & 0, B \\
\end{align*}
\]

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

\[
\begin{align*}
L \quad & 1, B \\
L \quad & 0, A \\
\text{ST} \quad & 0, B \\
\text{ST} \quad & 1, A \\
\end{align*}
\]

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, an optional feature available on System/360 which allows some degree of supervision over the areas of memory accessible 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

\[
\begin{align*}
L \quad & 1, A \\
L \quad & 2, A+4 \\
L \quad & 3, A+8 \\
\text{ST} \quad & 1, B \\
\text{ST} \quad & 2, B+4 \\
\text{ST} \quad & 3, B+8 \\
\end{align*}
\]
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 IM (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:

\[
\text{IM (or STM)} \quad r_1, r_3, d_2(b_2)
\]

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

<table>
<thead>
<tr>
<th>operation code</th>
<th>r_1</th>
<th>r_3</th>
<th>b_2</th>
<th>d_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \( R_{r_1} \), 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 \( R_{r_3} \) 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 \( R_{r_1} \) through \( R_{15} \) will be transmitted, followed by \( R_0 \) through \( R_{r_3} \); thus \( R_0 \) may be considered to follow after \( R_{15} \), 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, IM 2,6,=5F'0' would cause the contents of \( R_2 \), \( R_3 \), \( R_4 \), \( R_5 \), and \( R_6 \) 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 \( R_1 \) contains the address of a list of
four fullword constants, we could load them into R7 through R10 by executing the statement $IM \ 7,10,0(1)$ 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 examples illustrate certain conventions commonly used in communicating with subroutines, which will be treated in detail later. As a final example, suppose we wish to exchange the contents of R0 through R7, as a block, with the contents of R8 through R15. We could then write

\[
\begin{align*}
STM & \ 0,15,SAVE \\
IM & \ 8,7,SAVE \\
--- & \ \\
S&AVE \ DS \ 16F \\
\end{align*}
\]

or

\[
\begin{align*}
STM & \ 8,7,SAVE \\
IM & \ 0,15,SAVE \\
--- & \ \\
S&AVE \ DS \ 16F \\
\end{align*}
\]

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 Rr1 replaces the halfword at the effective address of the second operand, and Rr1 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 in precisely 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 and FFFFFFFFFF16 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(R0) 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(R0)=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 RL 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'00000000'
  STH 0,A         C(A)=X'FFFF'
   LH 1,A         C(RL)=X'FFFFFFFF'

A   DS   H
```

Figure 14.3 Loss of Significant Digits when Using STH

In this case the result in RL has a different sign 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 \texttt{STC r1,d2(x2,d2)} causes the rightmost byte of Rr1 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. As 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  C'L2   BECO£ES C'BA'
```
Occasionally when memory space is at a premium it is convenient to use a single byte to contain a small integer constant; its value may be placed in a register using the following instruction sequence.

```
L1,F'0' CLEAR REGISTER
L1,LITLCON INSERT CONSTANT
LITLCON DC F1153'
```

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 data 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), and 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 five 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 differ from \( r_1 \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>( C(R_{r_1}) \leftarrow C(R_{r_2}) )</td>
<td>not set</td>
</tr>
<tr>
<td>LTR</td>
<td>( C(R_{r_1}) \leftarrow C(R_{r_2}) )</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCR</td>
<td>( C(R_{r_1}) \leftarrow \overline{C(R_{r_2})} )</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>LPR</td>
<td>( C(R_{r_1}) \leftarrow</td>
<td>C(R_{r_2})</td>
</tr>
<tr>
<td>LNR</td>
<td>( C(R_{r_1}) \leftarrow -</td>
<td>C(R_{r_2})</td>
</tr>
</tbody>
</table>

Figure 14.4 Action of Certain General Register Instructions

The meanings of the CC settings are given below.

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Result is Zero</td>
</tr>
<tr>
<td>1</td>
<td>Result is Negative</td>
</tr>
<tr>
<td>2</td>
<td>Result is Positive</td>
</tr>
<tr>
<td>3</td>
<td>Result has Overflowed</td>
</tr>
</tbody>
</table>

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 than 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 LFR 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 NOT SET
LR  7,3        C(R7)=0,  CC NOT 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
LFR 0,4        C(R0)=+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 IM and SIM in the important respect that the R3 register specification 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 \( R_8 \) are \( 87654321_{16} \) and the instruction

\[-SLL \quad 8,1\]

is executed, the result in \( R_8 \) will be \( 0ECA8642_{16} \). Note that we
could have written \( SLL 8,1(0) \) also, because the explicit use of 0 as
a base register specification.dig3.t causes no base register to be used in
the calculation of an effective address. Again supposing \( R_8 \) to contain
\( 87654321_{16} \) and \( R_3 \) to contain \( 82F3A2B5_{16} \), execution of the instruction

\[ SRL \quad 8,16(3) \]

would cause the contents of \( R_8 \) to be shifted right
\( \ldots_{16} \) \( 10_{16} \) = \( 05_{16} \) (modulo \( 40_{16} \)) bit positions, leaving \( 043B2A_{16} \) as
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 \( R_5 \), 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 bits occupy the second and third bytes. Suppose \( R_5 \) contains \( \text{FFFA620B}_{16} \), and \( R_{12} \) 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 \( \text{XXYYZZ} \) are meant to represent the hex digits of the three characters in the data entry.)

\[
\begin{align*}
\text{STC} & \, 5,3(0,12) \quad \text{C(DATA ENTRY)} = \text{XXYYZZOB} \quad \text{----} \\
\text{SRL} & \, 5,8 \quad \text{C(R5)} = \text{00FFFA62} \\
\text{STH} & \, 5,4(0,12) \quad \text{C(DATA ENTRY)} = \text{XXYYZZOBFA62}
\end{align*}
\]

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

\[
\begin{align*}
\text{LH} & \, 5,4(0,12) \quad \text{C(R5)} = \text{FFFFFA62} \\
\text{SLL} & \, 5,8 \quad \text{C(R5)} = \text{FFFFFA6200} \\
\text{IC} & \, 5,3(0,12) \quad \text{C(R5)} = \text{FFFFFA620B}
\end{align*}
\]

This example also illustrates a situation where the need for efficient use of memory space 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 \( \text{L} \) and \( \text{ST} \) instructions could be used, with a considerable increase in speed (an approximate factor of 3) for this 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 bytes. 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 \( R_5 \) would be as follows, assuming that the data entry contained \( \text{FA620BXXYYZZ} \).

\[
\begin{align*}
\text{LH} & \, 5,0(0,12) \quad \text{C(R5)} = \text{FFFFFA62} \\
\text{SL} & \, 5,8 \quad \text{C(R5)} = \text{FFFFFA6200} \\
\text{IC} & \, 5,2(0,12) \quad \text{C(R5)} = \text{FFFFFA620B}
\end{align*}
\]

It is apparent that the particular arrangement of the data in memory may depend on the programmer's inclinations, 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 (Rr1) 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 Rr1 entering the left end of Rr1+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
SRDL 2,16
LH 2,A+2
SLDL
ST 2,A
LH 2,A
SRDL 2,16
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.

```
<table>
<thead>
<tr>
<th>1st integer</th>
<th>2nd integer</th>
<th>3rd integer</th>
<th>4th integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 bits long</td>
<td>4 bits long</td>
<td>13 bits long</td>
<td>6 bits long</td>
</tr>
<tr>
<td>0 8 9 12 13 25 26 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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, SECOND, THIRD, and FOURTH, 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.
Another code sequence to do the same task is:

L 2,=F*0'  GET A 0 CONSTANT FOR CLEARING RO
L 1,0(0,9). GET DATA FULLWORD
LR 0,2   CLEAR RO
SLDL 0,9  SHIFT 9 BITS INTO RO
ST 0,FIRST STORE FIRST INTEGER
LR 0,2   CLEAR RO
SLDL 0,4  SHIFT 4 BITS INTO RO
ST 0,SECOND
LR 0,2   CLEAR RO
SLDL 0,13 SHIFT 13 BITS INTO RO
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 code and slightly faster execution. The overall saving is quite small, but the choice serves as 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 right 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 $F	ext{FF}F_{16}$:

\[
\begin{align*}
L & 0,=F'-8' \\
SRL & 0,2 \\
SRA & 0,2
\end{align*}
\]

After the logical shift, C(RO)=$3	ext{FFFFP2}_{16}$, and after the arithmetic shift, C(RO)=$F	ext{FF}F	ext{FF}F_{216}$. 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 left 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.

\[
\begin{align*}
L & 0,=F'-8' \\
SRL & 0,2 \\
SLA & 0,4 \\
C & \text{RO}=TFFFFF_{20}, \text{CC SET TO 3, OVERFLOW}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CC = 0</th>
<th>CC = 1</th>
<th>CC = 2</th>
<th>CC = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIA</td>
<td>Result=0</td>
<td>Result&lt;0</td>
<td>Result&gt;0</td>
<td>Overflow</td>
</tr>
<tr>
<td>SRA</td>
<td>Result=0</td>
<td>Result&lt;0</td>
<td>Result&gt;0</td>
<td>Impossible</td>
</tr>
<tr>
<td>SLDA</td>
<td>Result=0</td>
<td>Result&lt;0</td>
<td>Result&gt;0</td>
<td>Overflow</td>
</tr>
<tr>
<td>SRDA</td>
<td>Result=0</td>
<td>Result&lt;0</td>
<td>Result&gt;0</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

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 RO and Rl contain zero.

An important characteristic of the arithmetic shift operations is that they provide a simple means for multiplying by positive and negative 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^n. 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 bits are lost; consider the two following code sequences.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>3, =F'5'</td>
<td>C(R3) = 00000005</td>
</tr>
<tr>
<td>SRA</td>
<td>3,1</td>
<td>C(R3) = 00000002</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>3, =F'5'</td>
<td>C(R3) = FFFFFFFF = -5</td>
</tr>
<tr>
<td>SRA</td>
<td>3,1</td>
<td>C(R3) = FFFFD = -3</td>
</tr>
</tbody>
</table>

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 lower 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. Both of the following code sequences achieve the desired result.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRA</td>
<td>9,4</td>
<td>SRL 9,4</td>
</tr>
<tr>
<td>SLA</td>
<td>9,4</td>
<td>SLL 9,4</td>
</tr>
</tbody>
</table>

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 O,0(0,9) GET DATA WORD
SRDA 0.6 SHIFT 6 BITS INTORI
SRA 1.26 EXTEND TO RIGHT
ST 1,FOURTH STORE FULLWORD RESULT
SRDA 0,13 SHIFT OFF 13 MOREBITS
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) 89900000
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.
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 r2 digit of the instruction unless r2 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 \( r_i \). Because this digit does not refer to \( R_{r_i} \), 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 01112 and 10012 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.

<table>
<thead>
<tr>
<th>Instruction Bit</th>
<th>Mask Bit Value</th>
<th>CC Value Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

**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 \( C(R12) = 0 \).
   
   \[
   \begin{align*}
   & \text{LTR } 12,12 \quad \text{or} \quad \text{SRA } 12,0 \\
   & \text{BC } 8, \times \quad \text{BC } 8, \times
   \end{align*}
   \]

2) Branch to X if \( C(R0) \neq 0 \).
   
   \[
   \begin{align*}
   & \text{LTR } 0,0 \quad \text{or} \quad \text{SLA } 0,0 \\
   & \text{BC } 6, \times \quad \text{BC } 7, \times
   \end{align*}
   \]

(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.
   
   \[
   \begin{align*}
   & \text{SIA } 5,2 \\
   & \text{BC } 14, \times
   \end{align*}
   \]

4) Branch to the address contained in R14.
   
   \[
   \begin{align*}
   & \text{BCR } 15,14 \quad \text{ (preferred) } \\
   & \quad \text{or} \quad \text{BC } 15,0(0,14) \quad \text{ (slower) } \\
   & \quad \text{or} \quad \text{BC } 15,0(14) \quad \text{ (slowest) }
   \end{align*}
   \]

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.
   
   \[
   \begin{align*}
   & \text{LCR } 8,2 \\
   & \text{BC } 5, \times
   \end{align*}
   \]

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:
   
   \[
   \begin{align*}
   & \text{L } 5, N \quad \text{GET INTEGER} \\
   & \text{L } 4, =F'0' \quad \text{CLEAR R4} \\
   & \text{SHIFT SLDL } 4, 1 \quad \text{SHIFT LEFT} \\
   & \text{LTR } 5, 5 \quad \text{TEST R5} \\
   & \text{BC } 7, \text{SHIFT} \quad \text{BRANCH IF NOT ZERO} \\
   & \text{ST } 4, N \quad \text{STORE RESULT}
   \end{align*}
   \]

15-3
b) Shift right, testing "lost" bits:

```
L    4,N    GET INTEGER
SHIFT SRDL 4,1    SHIFT RIGHT
LTR  5,5    TEST SIGN OF R5
BC   10, SHIFT BRANCH IF NOT -
SLDL 4,1    MOVE BIT BACK
ST   4,N    STORE 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'FF', 14 and BCR B'11111', 14 also), since the branch condition must always be met. There are occasions 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 NOP s and NOPR 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 NOPR O 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 NPAIR should be used or not.

To relieve the programmer of this worry, the Assembler provides an instruction CNOP (Condition No-Operation) which ensures the desired alignment. The operand field entry of a CNOP 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. No 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 NPAIR's to force the LC to the desired boundary.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Alignment Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNOP 0,4</td>
<td>Beginning of a fullword</td>
</tr>
<tr>
<td>CNOP 2,4</td>
<td>Middle of a fullword</td>
</tr>
<tr>
<td>CNOP 0,8</td>
<td>Beginning of a doubleword</td>
</tr>
<tr>
<td>CNOP 2,8</td>
<td>Second halfword of a doubleword</td>
</tr>
<tr>
<td>CNOP 4,8</td>
<td>Middle of a doubleword</td>
</tr>
<tr>
<td>CNOP 6,8</td>
<td>Fourth halfword of a doubleword</td>
</tr>
</tbody>
</table>

Figure 15.2 CNOP Alignments

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

```
CNOP 2,4
BALR 14,15
DC A(Anything)
```

Note that we could not write

```
DS 0
BALR 14,15
DC A(Anything)
```

because the alignment to a halfword 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 OF
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 next 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.
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.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>RR</td>
<td>Add Register</td>
</tr>
<tr>
<td>A</td>
<td>RX</td>
<td>Add</td>
</tr>
<tr>
<td>ALR</td>
<td>RR</td>
<td>Add Logical Register</td>
</tr>
<tr>
<td>AL</td>
<td>RX</td>
<td>Add Logical</td>
</tr>
<tr>
<td>AH</td>
<td>Rx</td>
<td>Add Half'word</td>
</tr>
<tr>
<td>SR</td>
<td>RR</td>
<td>Subtract Register</td>
</tr>
<tr>
<td>CR</td>
<td>RR</td>
<td>Compare Register</td>
</tr>
<tr>
<td>S</td>
<td>Rx</td>
<td>Subtract</td>
</tr>
<tr>
<td>C</td>
<td>Rx</td>
<td>Compare</td>
</tr>
<tr>
<td>SLR</td>
<td>RR</td>
<td>Subtract Logical Register</td>
</tr>
<tr>
<td>CLR</td>
<td>RR</td>
<td>Compare Logical Register</td>
</tr>
<tr>
<td>SL</td>
<td>RX</td>
<td>Subtract Logical</td>
</tr>
<tr>
<td>CL</td>
<td>Rx</td>
<td>Compare Logical</td>
</tr>
<tr>
<td>SH</td>
<td>RX</td>
<td>Subtract Halfword</td>
</tr>
<tr>
<td>CH</td>
<td>RX</td>
<td>Compare Halfword</td>
</tr>
<tr>
<td>MR</td>
<td>RR</td>
<td>Multiply Register</td>
</tr>
<tr>
<td>M</td>
<td>Rx</td>
<td>Multiply</td>
</tr>
<tr>
<td>MH</td>
<td>RX</td>
<td>Multiply Halfword</td>
</tr>
<tr>
<td>DR</td>
<td>RR</td>
<td>Divide Register</td>
</tr>
<tr>
<td>D</td>
<td>RX</td>
<td>Divide</td>
</tr>
</tbody>
</table>

*Figure 16.1 Fixed-Point Arithmetic Instructions*
There are several 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 MX instructions, and "HW_2" means the same for halfword operands.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>C(Rr_1) ← C(Rr_1)+C(Rr_2)</td>
<td>0: Result is zero</td>
</tr>
<tr>
<td>SR</td>
<td>C(Rr_1) ← C(Rr_1)-C(Rr_2)</td>
<td>1: Result is&lt; 0</td>
</tr>
<tr>
<td>A</td>
<td>C(Rr_1) ← C(Rr_1)+C(FW_2)</td>
<td>2: Result is&gt; 0</td>
</tr>
<tr>
<td>S</td>
<td>C(Rr_1) ← C(Rr_1)-C(FW_2)</td>
<td>3: Overflow</td>
</tr>
<tr>
<td>AH</td>
<td>C(Rr_1) ← C(Rr_1)+C(HW_2)</td>
<td></td>
</tr>
<tr>
<td>SH'</td>
<td>C(Rr_1) ← C(Rr_1)-C(HW_2)</td>
<td></td>
</tr>
<tr>
<td>ALR</td>
<td>C(Rr_1) ← C(Rr_1)+C(Rr_2)</td>
<td>0: Zero result, no carry</td>
</tr>
<tr>
<td>SLR</td>
<td>C(Rr_1) ← C(Rr_1)-C(Rr_2)</td>
<td>1: Nonzero result, no carry</td>
</tr>
<tr>
<td>AL</td>
<td>C(Rr_1) ← C(Rr_1)+C(FW_2)</td>
<td>2: Zero result, carry</td>
</tr>
<tr>
<td>SL</td>
<td>C(Rr_1) ← C(Rr_1)-C(FW_2)</td>
<td>3: Nonzero result, carry</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{L} & \quad 5, X \\
\text{A} & \quad 5, Y \\
\text{BC} & \quad 11, \text{ST} \\
\text{A} & \quad 5, 2 \\
\text{SH} & \quad 5, = \text{H}'2' \\
\text{ST} & \quad \text{ST} 5, \text{ANS}
\end{align*}
\]

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.

\[
\begin{align*}
\text{LH} & \quad 3, N \\
\text{LM} & \quad 6, 9, = \text{F}'0', 2, 1, 1' \\
\text{ADDUP} & \quad 6, 8 \\
\text{AR} & \quad 8, 7 \\
\text{SR} & \quad 3, 9 \\
\text{BC} & \quad 7, \text{ADDUP} \\
\text{ST} & \quad 6, \text{SUM}
\end{align*}
\]

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 compute NEWSTOCK from the formula

\[
\text{NEWSTOCK} = \text{OLDSTOCK} + \text{RECEIPTS} - \text{SALES}
\]
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
A 2,RECEIPTS
S 2,SALES
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.

<table>
<thead>
<tr>
<th>Operations</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>0: Operand 1 = Operand 2</td>
</tr>
<tr>
<td>C</td>
<td>1: Operand 1 &lt; Operand 2</td>
</tr>
<tr>
<td>CH</td>
<td>2: Operand 1 &gt; Operand 2</td>
</tr>
<tr>
<td>CLR</td>
<td></td>
</tr>
<tr>
<td>C L</td>
<td></td>
</tr>
</tbody>
</table>

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 Rr1 for the subtract instructions. Further analysis of the original operands is required in the CPU if the internal result overflows. The logical-comparisons do not 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.
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.

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.

\[
\begin{align*}
\text{SR} & \quad 7,7 & \text{CLEAR R7} \\
\text{SRDL} & \quad 6,3 & \text{SHIFT 3 BITS INTO R7} \\
\text{LTR} & \quad 7,7 & \text{SEE IF THE BITS ARE ZERO} \\
\text{BC} & \quad 8,A & \text{BRANCH IF YES} \\
\text{A} & \quad 6,'F'1' & \text{IF NOT, ADD 1 TO R6} \\
\text{A} & \quad \text{SLL} & \quad 6,3 & \text{MULTIPLY BY 8}
\end{align*}
\]

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 \( \text{L } 7,='0' \) 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 \( \text{SRL } 6,3 \) 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:

\[
\begin{align*}
\text{A} & \quad 7,'F'7' & \text{FORCE CARRY IF POSSIBLE} \\
\text{SRL} & \quad 7,3 & \text{DROP OFF 3 BITS} \\
\text{SLL} & \quad 6,3 & \text{MULTIPLY BY 8}
\end{align*}
\]

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 23 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(R0,R1) we mean the contents of the double-length register formed by RO and RL.

```
L   0,=F'-1'
LR  1,0       C(R0,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  s TM 0,1,ARG STORE COMPLEMENTED RESULT

DS  OD       ALIGN ON DOUBLEWORD BOUNDARY
ARG cc FL8'123456787654321'
```

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-ORDER WORD
BC  8,X      JUMP IF C(R1) = 0
S   0,=F'1'  SUBTRACT 1 FROM RO
X  STM 0,1,ARG STORE RESULT

DS  OD       ALIGN
ARG - DC FL8'987654356789'
```

In this case, we use the first LCR instruction to form the two's complement of C(R0) immediately; that is, we have already added a low-order 1 bit to the one's complement of C(R0). 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 $C(R_0)$ by subtracting 'of 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,='1'           PROPAGATE CARRY BIT TO HIGH-ORDER PART
NC  A 0,3           ADD HIGH-ORDER PARTS
STM 0,1,C           STORE DOUBLE-LENGTH SUM
C  DS 0             FRESERVE 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 LCW-ORDER PART OF SECOND GPERAND
BC 3,CAR           BRANCH IF THERE'S A CARRY
S 0,='1'           REDUCE $C(R_0)$ BY 1 (BORROW 1)
CAR  S 0,3           SUBTRACT HIGH-ORDER PART OF SECOND OPERAND
STM 0,1,C           STORE DOUBLE-LENGTH DIFFERENCE
C  DS 0             FRESERVE BYTES, ALIGNED
B   DC FL8'123456787654321'
A   'cc FL8'23456789765432'
```

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 result cannot 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 $'1'$ 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.

\[
\begin{align*}
\text{MR} & \quad 2.7 \quad \text{C}(R2,R3) = \text{C}(R3) \times \text{C}(R7) \\
\text{MR} & \quad 0.1 \quad \text{C}(R0,R1) = \text{C}(R1) \times \text{C}(R1) \\
\text{MR} & \quad 8.8 \quad \text{C}(R8,R9) = \text{C}(R8) \times \text{C}(R9) \\
\text{M} & \quad 4,X \quad \text{C}(R4,R5) = \text{C}(R5) \times \text{C}(X) \\
\text{M} & \quad 12,=F '932' \quad \text{C}(R12,R13) = \text{C}(R13) \times 932 \\
\text{LR} & \quad 5,4 \quad \text{MOVE MULTIPLICAND TO R5} \\
\text{MR} & \quad 4,4 \quad \text{C}(R4,R5) = \text{C}(R5) \times \text{C}(R4) \\
\end{align*}
\]

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 \( \text{MR} 4,5 \) also, giving \( \text{C}(R5) \times \text{C}(R5) \). 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 \( 65536 = 2^{16} \))

\[
\begin{align*}
\text{L} & \quad 1,=A('X'10000') \quad \text{C}(R1) = 65536 \\
\text{MR} & \quad 0,1 \quad \text{SQUARE IT} \\
\text{ST} & \quad 1,\text{PRODUCT} \\
\text{PRODUCT} & \quad \text{C} \quad \text{S} \quad \text{F}
\end{align*}
\]

we would find that the fullword stored at \text{PRODUCT} was zero and that \( \text{C}(R0) = 1 \); and if we executed the instruction sequence (note that \( 32768 = 2^{15} \))
we would find that \( C(\text{PRODUCT}) = -2^{31} \). 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 not a sign bit, but contains an arithmetically significant digit.

The \texttt{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 \texttt{fullword} result is retained, \( r_1 \) need not be even, and a specification exception will occur only if the effective address of the \texttt{halfword} operand is odd. Because fewer shifts and adds are needed during multiplication, some small economies may be achieved by the use of \texttt{MH}, particularly on the smaller models of \texttt{System/360}. Thus, \texttt{MH 5,=H'100'} is a simple way to multiply the contents of \( R5 \) by 100. If \( X \) and \( Y \) are both \texttt{halfword} operands, their product may be found by writing

\[
\begin{align*}
\text{LH} & \quad 9,X \\
\text{MH} & \quad 9,Y 
\end{align*}
\]

and \( R8 \) is undisturbed. And to square the \texttt{halfword} integer \( n \) at \( N \) we could write

\[
\begin{align*}
\text{LH} & \quad 6,N \\
\text{MH} & \quad 6,N 
\end{align*}
\]

Note that because both operands are \texttt{halfwords} of at most 15 significant bits, the product will fit in a single register; the only \texttt{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 \( A = B + G \times D \), where all quantities are \texttt{fullword} integers, and it is assumed that all results are small enough so that no overflows occur.
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 \[ \sum_{k=1}^{n} k^3. \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 7, G</td>
<td>C(R7) := C(G)</td>
</tr>
<tr>
<td>M 6, D</td>
<td>C(R6) := G * D</td>
</tr>
<tr>
<td>A 7, B</td>
<td>C(R7) := B + G * D</td>
</tr>
<tr>
<td>S1 7, A</td>
<td>STCRE RESULT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 5, 5</td>
<td>SUM CARRIED IN R5</td>
</tr>
<tr>
<td>L 4, =F '1'</td>
<td>K CARRIED IN R4</td>
</tr>
<tr>
<td>RPT</td>
<td>C(R1) := K</td>
</tr>
<tr>
<td>LR 1, 4</td>
<td>C(R0, R1) := K * K</td>
</tr>
<tr>
<td>MR 0, 1</td>
<td>C(R0, R1) := K * KUBED</td>
</tr>
<tr>
<td>AR 5, 1</td>
<td>ACCUMULATE SUM</td>
</tr>
<tr>
<td>A 4, =F '1'</td>
<td>INCREMENT K</td>
</tr>
<tr>
<td>C 4, NBR</td>
<td>COMPARE TO UPPER LIMIT</td>
</tr>
<tr>
<td>BC 12, RPT</td>
<td>BRANCH IF K NOT BIGGER</td>
</tr>
<tr>
<td>ST 5, SUM</td>
<td>STORE SUM OF CUBES</td>
</tr>
</tbody>
</table>

16-11
Division is always performed using a double-length dividend and remainder. As was the case for the fullword multiply instructions, the r 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 low-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.

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 rounded quotient at Q. This means that if the remainder is 5 or larger the quotient must be increased by 1.

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 \texttt{OKAY} independent of the magnitude of the remainder. To obtain a correctly represented dividend it is simplest to use the \texttt{SRDA} instruction, as shown.

\begin{verbatim}
L   1,=F'1'    SETUP ROUNDING BIT
L   6,N
SRDA 6,32  C(R6) = C(N)
BC   11,0DIV  C(R6, R7) = 64-BIT DIVIDEND
      1,1     JUMP IF NON-NEGATIVE DIVIDEND
      LCR3      OTHERWISE SET ROUNDOFF TO -1
CIV  D 6,F'10'  DIVIDE BY 10
LPK 6,6
C  6,F'5'     ABSOLUTE VALUE OF REMAINDER
BC  4,OKAY    COMPARE TO 5
AU  7,1     BRANCH IF SMALLER THAN 5
OKAY ST 7,Q   ADD CORRECTLY-SIGNED ROUNDOFF
STORE ROUNDED QUOTIENT
\end{verbatim}

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

$$|C(R1)| < \frac{1}{2}$$

second operand is satisfied, the quotient can be computed correctly.
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 instruction: 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
A 0,B
BC 14,ST
LA 0,0
ST ST 0,ANSWER

L 0,A
A 0,B
BC 14,ST
L 0,=F'0'
ST ST 0,ANSWER
```
Because the LA instruction computes an effective address, it also provides a simple way to increment the contents 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 COUNT contain the number of positions shifted.

```
L 4,N
L 3,=F'-1'
SHIFT S R D L 4,1
LTR 5,5
LA 3,1(0,3)
BC 10,SHIFT
SLDL 491
ST 4,N
STH 3,COUNT
GET INTEGER
INITIAL SHIFT COUNT
SHIFT A BIT INTO R5
TEST SIGN OF R5
INCREMENT R3 BY 1
BRANCH IF R5 NOT NEGATIVE
MOVE BIT BACK IN PLACE
STORE SHIFTED INTEGER
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 ERR if no overflow occurs, and to ERR1 if one or more overflows occur. Let the integers be stored in successive fullwords beginning at Q.
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 instruction 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 `NOERR` 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.

```
\begin{array}{c|c|c}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
\hline
\end{array}
```

\(\wedge\) AND

```
\begin{array}{c|c|c}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
\hline
\end{array}
```

\(\lor\) OR

```
\begin{array}{c|c|c}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 1 & 1 \\
\hline
\end{array}
```

\(\oplus\) EXCLUSIVE OR

\textit{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, "FW2" means the fullword second operand specified by the effective address.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Type</th>
<th>Action</th>
<th>CC Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR</td>
<td>RR</td>
<td>C(Rr1) → C(Rr1) ∧ C(Rr2)</td>
<td>0: all result bits are zero</td>
</tr>
<tr>
<td>N</td>
<td>Rx</td>
<td>C(Rr1) → C(Rr1) A C(FW2)</td>
<td></td>
</tr>
<tr>
<td>ØR</td>
<td>RR</td>
<td>C(Rr1) → C(Rr1) v C(Rr2)</td>
<td>1: result bits are not all zero</td>
</tr>
<tr>
<td>Ø</td>
<td>RX</td>
<td>C(Rr1) → C(Rr1) v C(FW2)</td>
<td></td>
</tr>
<tr>
<td>XR</td>
<td>RR</td>
<td>C(Rr1) → C(Rr1) ⊕ C(Rr2)</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Rx</td>
<td>C(Rr1) → C(Rr1) ⊕ C(FW2)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17.2 Logical Instructions

Suppose C(R4) = 0123456716, and C(R9) = EDA9652116. Then if the instructions indicated are executed, the final contents of R4 will be as shown below the instruction.

<table>
<thead>
<tr>
<th>NR</th>
<th>4,9</th>
<th>ØR</th>
<th>4,9</th>
<th>XR</th>
<th>4,9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123452116</td>
<td></td>
<td>EDAB656716</td>
<td></td>
<td>EC8A204616</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>3</th>
<th>0011</th>
<th>3</th>
<th>0011</th>
<th>3</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧9</td>
<td>∧1001</td>
<td>∧9</td>
<td>v1001</td>
<td>∧9</td>
<td>⊕1001</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>B</td>
<td>1011</td>
<td>A</td>
<td>1010</td>
</tr>
</tbody>
</table>

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 R1
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  AAAAAAAAAAABBBBCCCCCCCCCCCCDDDDDDDD
N  1,MASK     00000000000000000000000000000000
SRL 1,6   00000000000000000000000000000000
ST 1,THIRD

- - -
DS   OF  ALIGN TO FULLWORD BOUNDARY
MASK  D  C  X'00007FFC0'
```

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 AND 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 INTERGERS  |
| S R D L | 0,6 | MOVE FOURTH INTO R1  |
| L | 0,NEWTHIRD | GET NEW VALUE OF THIRD INTEGER  |
| S R D L | 0,913 | MOVE IT IN WITH FOURTH  |
| L | 0,DATAWORD | GET INTERGERS AGAIN  |
| S R L | 0,19 | DROP OFF THIRD AND FOURTH  |
| S R D C | 0,13 | MOVE FULL WORD INTO R1  |
| ST | 1,DATAWORD | STORE NEW DATAWORD  |

Alternatively, we can use the logical AND and $\&$R to do the same:

| L | 0,DATAWORD | GET INTERGERS  |
| N | 0,MASKA | CLEAR SPACE FOR THIRD  |
| L | 1,NEWTHIRD | GET NEW VALUE OF THIRD INTEGER  |
| S L L | 1,6 | SHIFT INTO PROPER POSITION  |
| OR | 001 | 'OR' INTO PLACE  |
| ST | 0,DATAWORD | STORE NEW DATAWORD  |

In this case, the N causes all the bit positions into which the third integer will be placed to be set to zero. The $\&$R instruction then forms the logical OR of all the bits of RO 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 RO, 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 RO. 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 $\&$R to insure that no extraneous bits are ORed into RO.

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'=l which does the same thing. We can rewrite the example above to use an X instruction (though in a somewhat roundabout way) as follows:
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'  FORC E 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 NORM the number of positions-shifted.

```
SR  8,8  SET SHIFT COUNT TO ZERO
SHIFT  SLA  6,1  SHIFT LEFT ONEBITPOSITION
       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,DI GIT  RESTORE THE LOST BIT
       ST  8,NORM  STORE SHIFT COUNT

NORM  DS  F
      .DIGIT DC 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.
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 some typical uses involving arrays of data.

As a simple example which will be used to illustrate some 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 \(193_{10} = X'1C'\)) 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 R 0
LR    1,0    CHARACTER COUNT IN R1, INITIALLY 0
LA    2,C'A'    C(R2) = X'0000000C1'
LA    3,C'0'    C(R3) = X'000000040'
LA    4,STR    FIRST BYTE ADDRESS IN R 4
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.

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 CKAY 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 N$ERR is to be taken if no overflows occurred, to . ERR1 if exactly one overflow occurred, and to ERR2 if two.
When the instruction named A2 is reached, Rl 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 Rl 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 Rr2 for BCTR (unless r2=0, in which case no branch can be taken) or from the effective address for BCT. In this case, after the branch address is computed, the branching condition is determined by first algebraically reducing the contents of Rr1 by one, and then branching unless C(Rr1)=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.
The use of the expression STR-1 in the second operands of the TC and STC instructions is dictated by the fact that the possible values of C(R1) run between 80 and 1, rather than 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
MR 0,1
MR 0,4
AR 5,1
BCT 4,NEXT
ST 5,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
Lh 1,N
LGP LA 2,0(1,1)
BCTR 2,0
AR 0,2
BCT 1,LOOP
ST 0,NSQ
```

`CLEAR SUM TC ZERO
GET N FROM MEMORY
(COUNT+COUNT) IN R2
2 * COUNT - 1
ADD TO SUM
DECREASE COUNTER BY 1, LOOP
REDUCE COUNT A N D BRANCH`
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 $r2 = 0$; 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
LCR 0,0
LCR 1,1
BC 8X
BCTR 0,0
STM 0,1,ARG
```

This is identical to the example in Section 16 except that the BCTR replaces $S_0=S'1'$ and thus the CC setting may be different when the STM is executed. The BCTR instruction with $r2=0$ 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
MULT LR 3,4
MR 2,3
MR 2,4
CR 1,4
SLL 1,2
ST 3,CUBE-4(1)
BCT 4,MULT
```

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.
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 comparison, 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 Rr1, 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 IM 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 we write $\text{BXLE } 0,4,\text{NEXT}$ then the index is in $R0$, the increment is in $R4$, and the comparand is in $R5$, whereas if we write $\text{BXLE } 0,5,\text{NEXT}$ the index is again in $R0$, but both the increment and the comparand are in $R5$. There is a simple notational device which illustrates the fact that the comparand is always 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 $Rr_3v1$ to indicate that the register containing the comparand may be determined by ORing a 1 digit into the $r_3$ digit. Thus $R8v1$ refers to $R9$, and $R9v1$ 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 not 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 before 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 before 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.
Note that the values of the index run from 0 to 79; when control reaches the EXLE instruction, the increment in R2 (namely +1) is added to C(RO), and because R2 is an even-numbered register, the sum is compared to the comparator 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 EXLE, the contents of R1 will be 8010.

To give an example where the use of EXLE 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.

```
CA 7,1 INITIAL INTEGER = 1
LR 8,7 C(R8) = 1 FOR INCREMENTING N
SR 4,4 SET INDEX TO ZERO
CA 2,4 INCREMENT OF +4 FOR INDEX
LA 3,36 CUBED = 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 BY 1
BXLE 4,2,MULT INCREASE INDEX BY 4 AND COOP
```

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.
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.

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.
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              |
| 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 `r3` digit is odd, so `R3V 1` is the same register as `Rr3`; the `BXH` will increment the index in `R4` by `-4` and branch until the resulting sum becomes `-4` also, when control will pass to the instruction following.

Some specialized uses of `BXH` and `EXLE` 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 `EXLE 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 `EXLE` 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's some negative integer in `R5`, `- BXH 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 comparison 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 `EXLE` 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 \( k_0, k_1, \ldots, k_{10} \) 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 \( K + 8n \) and \( K + 16n \) 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 \( k_m \) 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's merely customary to store arrays this way.) An example will help to illustrate this.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 0,0</td>
<td>INITIALIZE SUM</td>
</tr>
<tr>
<td>L 1,\text{LOWER}</td>
<td>INITIALIZE SUBSCRIPT ( n ), LOWER BOUND = 5</td>
</tr>
<tr>
<td>A 2,\text{LOWER}</td>
<td>INDEX CALCULATED IN R2</td>
</tr>
<tr>
<td>S 2.2</td>
<td>( 4\times(n-m) )</td>
</tr>
<tr>
<td>SAC 0,2</td>
<td>( \text{SUM} = \text{SUM} + X(N) )</td>
</tr>
<tr>
<td>LA 1,1(0,1)</td>
<td>INCREMENT ( N ) BY 1</td>
</tr>
<tr>
<td>LG 1,\text{UPPER}</td>
<td>CMPAREN TO UPPER BOUND</td>
</tr>
<tr>
<td>BC 12,4</td>
<td>IF NOT GREATER, BRANCH</td>
</tr>
<tr>
<td>ST 0,1</td>
<td></td>
</tr>
</tbody>
</table>

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 $L^*(-m)$.

It can be seen if $C(\text{LOWER}) = 5$ and $C(\text{UPPER}) = 17$ that the same result will be obtained; the first element to be added will be at $X-20(4\times 5) = X$, 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](image)

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 $a_{1j}$ is as follows, assuming that $a_{11}$ is stored at $A$: to obtain the address of

```
SR   0,0
LA   4,4
L    2,LOWER
SLL  2,2
L    5,UPPER
SLL  5,2

ACC  0,X-20(2)
XLE  2,4,ADD
ST   0,1
```
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 $I*(i-1)$ to the partially computed address, giving $A+I*(2*(j-1)+(i-1))$. 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 $I*(r*(j-1)+(i-1))$. 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
BCTR 6,0 FORM J - 1
MH 6,*H5' MULTIPLY BY NUMBER OF ROWS
A 6,I ADD ROW INDEX
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
```

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,*H5' *(NUMBER OF ROWS)
L 6,I + ROW INDEX
SLL 6,2 (ALL)*(ELEMENT LENGTH)
L 3,A-4*(5+1)(6) G(R3) = A(I,J)
ST 3,X STORE 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 "a00" (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 $C\phi LADDR$, as indicated in Figure 18.4. The notation $A(x)$ means "address of x".

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C\phi LADDR$</td>
<td>$A(a_{11})$</td>
</tr>
<tr>
<td>$C\phi LADDR + 4$</td>
<td>$A(a_{12})$</td>
</tr>
<tr>
<td>$C\phi LADDR + 8$</td>
<td>$A(a_{13})$</td>
</tr>
</tbody>
</table>

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.

![Table](image)

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

```
L 2,I GET ROW INDEX
L 3,J GET COLUMN INDEX
SLDL 2,2 M U L T I P L Y B O T H B Y 4
L 4,ADDRTAB-4(3) GET COLUMNADDRESS
L 0,0(2,4) GET A[I,J]
ST 0,X STORE A[ X
```

The expressions in the address constants are written in such a way that the programmer need only specify the value to be given to NRWS 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 $R_{r1}$ 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 immediate 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.

![Table of 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.
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) NI X,255 equivalent to (2), except CC = 1
(6) NI 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, NI, 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*XZ* 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 from memory. We can rewrite the sample program from 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.

<table>
<thead>
<tr>
<th>NEXT</th>
<th>LA</th>
<th>INITIALIZE LOOP COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1,80</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2,STR-1(1)</td>
<td>CONSTRUCT CHARACTER ADDRESS WITH INDEXING</td>
</tr>
<tr>
<td>CL1</td>
<td>0(2),C'A'</td>
<td>COMPARE ADDRESSED CHARACTER TO LETTER 'A'</td>
</tr>
<tr>
<td>BC</td>
<td>10,ANUM</td>
<td>BRANCH IF NOT LESS THAN 'A'</td>
</tr>
<tr>
<td>MVI</td>
<td>0(2),C'A'</td>
<td>BLANK OUT IF NON-ALPHANUMERIC</td>
</tr>
<tr>
<td>ANUM</td>
<td>8CT</td>
<td>COUNT DOWN AND LOOP</td>
</tr>
</tbody>
</table>

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 193₁₀, 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.

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bits examined are all zeros or mask is zero</td>
</tr>
<tr>
<td>1</td>
<td>Bits examined are mixed zero and one</td>
</tr>
<tr>
<td>3</td>
<td>Bits examined are all ones</td>
</tr>
</tbody>
</table>

Figure 19.3 CC Settings after TM Instruction

19-3
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.)
   \[
   \begin{align*}
   &TM \quad NUM, X'80' \\
   &BC \quad 8, MINUS
   \end{align*}
   \]

2) Branch to EVEN if the fullword integer stored at NUM is even.
   \[
   \begin{align*}
   &TM \quad NUM+3, 1 \\
   &BC \quad 8, EVEN
   \end{align*}
   \]

3) Branch to MIXED if the bits in the byte at B are not all zero or all one.
   \[
   \begin{align*}
   &TM \quad B, 255 \\
   &BC \quad 4, MIXED
   \end{align*}
   \]

4) Branch to SMALL if the value of the halfword integer at HNUM is between -512 and 511.
   \[
   \begin{align*}
   &TM \quad HNUM, X'FE' \\
   &BC \quad 9, SMALL
   \end{align*}
   \]

When used in conjunction with the NI, \( \phi \), 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 ERR if no overflows occurred and to ERROR if one or more overflows occurred.

\[
\begin{align*}
&MI \quad FLAG, 0 \quad \text{SET INDICATOR CR NO OVERFLOWS} \\
&L \quad 0, Q \quad \text{GET FIRST INTEGER} \\
&A \quad 0, Q+4 \quad \text{ADD SECOND INTEGER} \\
&SC \quad 14, NEXTA \quad \text{BRANCH IF NO OVERFLOW} \\
&OX \quad FLAG, 1 \quad \text{SET OVERFLOW FLAG ON (T O 1)} \\
&NEXTA \quad A \quad 0, Q+8 \quad \text{ADD THIRD INTEGER} \\
&BC \quad 1, ERROR \quad \text{BRANCH IF OVERFLOW TO ERROR} \\
&TM \quad FLAG, 1 \quad \text{OTHERWISE EXAMINE OVERFLOW FLAG BIT} \\
&5C \quad 8, NOERR \quad \text{IF BIT WAS ZERO, NO OVERFLOWS} \\
&BC \quad 1, ERROR \quad \text{IF ONE OVERFLOW OCCURRED} \\
&--- \quad \text{OVERFLOW FLAG BYTE} \\
&DS \quad X \quad \text{OVERFLOW FLAG BYTE} \\
&DS \quad 3F \quad \text{INTERGERS JO BE ACGE}
\end{align*}
\]
The \( \phi I \) 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.

\[
\begin{align*}
&\text{LA} & 4, \text{LIST} & \text{INITIAL LIST ADDRESS IN R 4} \\
&\text{L} & 3, N & \text{NUMBER OF ELEMENTS IN R 3} \\
&\text{SR} & 6, 6 & \text{INITIALIZE SUM TO ZERO} \\
&\text{Camb} & 5, 0(0,4) & \text{GET A HALFWORD LIST ELEMENT IN R5} \\
&\text{AR} & 6, 5 & \text{ADD TO SUM ONCE} \\
&\text{TM} & \text{SWITCH,} 1 & \text{TEST SWITCH BIT} \\
&\text{BC} & 8, \text{ONCE} & \text{BRANCH IF O, ADD ONLY ONCE} \\
&\text{AR} & 6, 5 & \text{ADD A SECOND TIME} \\
&\text{Camb} & 4, 2(0,4) & \text{INCREMENT LIST ADDRESS BY 2} \\
&\text{XI} & \text{SWITCH,} 1 & \text{INVERT SWITCH BIT} \\
&\text{BCT} & 3, \text{LOAD} & \text{BRANCH TO GET NEXT ELEMENT IF NOT DONE}
\end{align*}
\]

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 it alternately 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.
<table>
<thead>
<tr>
<th>L</th>
<th>1,N</th>
<th>GET NUMBER OF ELEMENTS TO BE ADDED</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>0,2</td>
<td>SET UP INCREMENT OF 2 IN R0</td>
</tr>
<tr>
<td>AR</td>
<td>1,1</td>
<td>2*N</td>
</tr>
<tr>
<td>SR</td>
<td>1,0</td>
<td>2*(N-1) IN R1 = COMPARAND FOR BXLE LOOP</td>
</tr>
<tr>
<td>SR</td>
<td>2,2</td>
<td>INITIALIZE INDEX IN R2 TO ZERO</td>
</tr>
<tr>
<td>LR</td>
<td>3,2</td>
<td>SAME FOR SUM IN R3</td>
</tr>
<tr>
<td>Of</td>
<td>BRNCH+1, X<em>FO</em></td>
<td>SET SWITCH FOR SINGLE ADD ON FIRST PASS</td>
</tr>
<tr>
<td>TM</td>
<td>SWITCH,1</td>
<td>CHECK SWITCH TO SEE IF SETUP IS CORRECT</td>
</tr>
<tr>
<td>BC</td>
<td>8,ADD</td>
<td>JUMP IF BRANCH HAS BEEN SET CORRECTLY</td>
</tr>
<tr>
<td>NI</td>
<td>BRNCH+1, X<em>OF</em></td>
<td>OTHERWISE SET UP TO ADD TWICE QN 1ST PASS</td>
</tr>
<tr>
<td>ACC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRNCH BC</td>
<td>0,FLIP</td>
<td>MASK FIELD HERE IS ALTERNATED BY XI</td>
</tr>
<tr>
<td>AH</td>
<td>3,LIST(2)</td>
<td>ADD A TERM</td>
</tr>
<tr>
<td>FLIP XI</td>
<td>BRNCH+1, X<em>FO</em></td>
<td>INVERT BRANCH MASKBITS</td>
</tr>
<tr>
<td>BXLE 2,0,ADD</td>
<td></td>
<td>COUNT AND LOOP</td>
</tr>
<tr>
<td>ST</td>
<td>3,RESULT</td>
<td>STORE ANSWER APPROPRIATELY</td>
</tr>
</tbody>
</table>

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 not 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 RO
SR 3, 9 3  INITIALIZE SUM TO ZERO
LR 2, 3  SAEH FOR INDEX
TM  SWITCH, 1  TEST WHETHER FIRST TERM ADDS TWICE
BC 1, TWICE  BRANCH IF BIT-1, MEANING YES
ONCE  A 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
Ah 3, LIST(2)  COMPARAND = 2(N-1)
LA 0, 2  INCREMENT = 2
SR 393  INITIALIZE SUM TO ZERO
SR 2, 2  INITIALIZE INDEX TO ZERO
ACC1  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 SINGLE
BC 8, ADD2  BRANCH IF YES
SR 2, 2  OTHERWISE RESET INITIAL INDEX TO ZERO
ACC2  Ah 3, LIST(2)  ADD AN ALTERNATE TERM FOR SECOND TIME
BXLE 2, 0, ADD2  INCREMENT INDEX BY 4 AND LOOP
```

19-7
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.

   \[ \text{NI STATUS,B'}01111111' \]

2) The policy holder has married.

   \[ \text{TM STATUS,B'}00010000' \]
   \[ \text{Bc 1,BIGAMY} \]
   \[ \phi \text{ STATUS,B'}00010000' \]

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

   \[ \text{TM STATUS,B'}1000' \]
   \[ \text{BC 1,TSKTSK} \]
   \[ \text{BC 15,TSK} \]

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

   \[ \text{JM STATUS,X'}30' \text{ TEST MARRIED AND TRAINING} \]
   \[ \text{BC 7, NEXT} \]
   \[ \text{JM STATUS,X'}CO' \text{ TEST AGE AND SEX} \]
   \[ \text{BC 1,HIGHCOST} \text{ IF YOUNG MACE, BRANCH} \]

   \[ \text{NEXT -- -- --} \]

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

   \[ \text{TM STATUS,X'}4' \]
   \[ \text{BC 8,NEXT} \]
   \[ \text{TM STATUS,X'}20' \]
   \[ \text{BC 1,NEXT} \]
   \[ \phi \text{ STATUS,X'}8' \]

   \[ \text{NEXT -- -- --} \]
6) If the policy holder is married or has completed driver training, branch to L&RISK.

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 digits which 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.

```
EA 2,10
LA 3,NBR-1(2)
MVI O(3),C'
BCT 2,BLANK
L 1,N
LPR 11
LA 3,NBR+10
CNVTLP SR 0.0
0 0 0=F90
STC 0 0(3,0)
CI 0(3),C+0
BGCTR 3,0
LTR 1,1
BC 2,CNVTLP
MVI 0(3),C++
TM N,X80
BC 8,ALLDONE
MVI 0(3),C--
ALLDONE L--
NBR OS- CL11
N GS F
```

- Set up to blank gut result area
- Construct byte address
- Stokes blanks in first 10 bytes
- Branch back 9 times
- Get number to be converted
- Take its magnitude
- Set up address of rightmost digit
- Clear high-order register
- Generate a digit by division
- Store the remainder byte
- Give digit CGRRECJ EBCDIC representation
- Move character pointer 1 byte to the left
- See if done, quotient goes to zero
- If zero, generate more digits
- Assume sign x 5, store that character
- Check actual sign of argument
- Branch if it was indeed positive
- Otherwise plant a - sign in the string
- Restuff program

- Output character string
- Number to be converted

19-9
20. SS Instructions

As the name implies, Storage-to-Storage instructions work with operands which are entirely in memory; except for TRT and EIMK, 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 packed decimal format, will be discussed later.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mvc</td>
<td>Move</td>
<td>NC</td>
<td>AND</td>
</tr>
<tr>
<td>MVN</td>
<td>Move Numerics</td>
<td>XC</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>MVZ</td>
<td>Move Zones</td>
<td>CLC</td>
<td>Compare</td>
</tr>
<tr>
<td>TR</td>
<td>Translate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRT</td>
<td>Translate and Test</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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](image-url)
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 a 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.

<table>
<thead>
<tr>
<th>Explicit Length</th>
<th>Implied Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁(L), s₂</td>
<td>s₁, s₂</td>
</tr>
<tr>
<td>d₁(L, b₁), s₂</td>
<td>d₁(b₁), s₂</td>
</tr>
<tr>
<td>s₁(L), d₂(b₂)</td>
<td>s₁, d₂(b₂)</td>
</tr>
<tr>
<td>d₁(L, b₁), d₂(b₂)</td>
<td>d₁(b₁), d₂(b₂)</td>
</tr>
</tbody>
</table>

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 12₅₁₆ and 4₇₁₆ 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*470(23,9),A
MVC B(23),293(S)
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 TO BLANKS
B D S CI23
```

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, is 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

```
MVC X'47D'(.9),A
```

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.
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

\[ \text{MVC } B-B'+X'47D'(,9),A \]

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 \( \text{MVC } B(L'^*),A \)

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 \( \text{MVC } B(L'=C'RAY'),=C'RAY' \) are incorrect.

Thus we can rewrite our simple example above, which uses an explicit base and displacement, as \( \text{MVC } X'47D'(L'B,9),A \)

---

**Figure 20.4 Determination of Length Specification Byte**

<table>
<thead>
<tr>
<th>Form of First Operand</th>
<th>Address Specification</th>
<th>Length Specification</th>
<th>Length Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>Implied</td>
<td>implied</td>
<td>length attribute of ( s_1 )</td>
</tr>
<tr>
<td>( s_1(L) )</td>
<td>implied</td>
<td>explicit</td>
<td>L</td>
</tr>
<tr>
<td>( d_1(L,b_1) )</td>
<td>explicit</td>
<td>implied</td>
<td>length attribute of ( d_1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>explicit</td>
<td>L</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>ASSEMBLED FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC B(23),A</td>
<td>0216 9470 9125</td>
</tr>
<tr>
<td>MVC B(1),A</td>
<td>0200 9470 9125</td>
</tr>
<tr>
<td>MVC B(0),A</td>
<td>0200 9470 9125</td>
</tr>
<tr>
<td>MVC O(L*),29(12)</td>
<td>D205 0000 0010</td>
</tr>
<tr>
<td>MVC 15(L'B-4,3),B</td>
<td>D212 300F 9470</td>
</tr>
<tr>
<td>MVC B,A</td>
<td>0216 9470 9125</td>
</tr>
<tr>
<td>MVC H(L'H,H)',H</td>
<td>D208 8008 0008</td>
</tr>
<tr>
<td>MVC H(H',H)',H(H)'</td>
<td>D207 8008 8008</td>
</tr>
<tr>
<td>MVC H+8-A(9),A</td>
<td>0200 9360 9125</td>
</tr>
<tr>
<td>MVC T,8-4</td>
<td>0216 9470 9479</td>
</tr>
<tr>
<td>MVC B-A+4(9),A</td>
<td>D208 035C 9125</td>
</tr>
</tbody>
</table>

E OS CL23
T EQU B
H EQU B

20-5
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'0'  
MVC LINE+1(119), LINE  PROPAGATE THROUGH REMAINING AREA
```

This requires less storage space than

```
MVC LINE(120), =120C'0'
```

(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'0'  
```

(3) Exchange the contents of the halfword integers at A and B.

```
MVC TEMP, A  MOVE TO TEMPORARY LOCATION
MVC A, B  MOVE B TO A
MVC B, TEMP  MOVE OLD C(A) FROM T E M Q TO 8
```

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.

```
BCIR 9,0  DECREASE LENGTH BY 1 FOR CPU
STC 9, MVC+1  STORE A 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" position of a character) and leftmost 4 bits (the "gone" portion of a character) are 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
LA 2,5
GET NUMBER TO BE CONVERTED
LA 2,5
COUNT NUMBER OF DIGITS IN R2
X SR 0,0
CLEAR HIGH-ORDER REGISTER
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
LA 3,TEMP
RETRIEVE NUMERIC PORTIONS OF DIGITS
LA 2,5
ADDRESS OF CURRENT DIGIT IN R3
LA 2,5
NUMBER OF DIGITS
SR 0,0
CLEAR REGISTER
LR 1,0
AND FOR NUMBER BEING GENERATED
MULT NH 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 ii = XL5,'0'
LZUNESPRESEN'T TO ZERO, DIGITS MOVED IN
```

20-7
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, φC, 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.

\[ \text{xc \hspace{1cm} LINE(120),LINE} \]

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.

\[ \phi C \hspace{1cm} LUMP(4),LUMP \hspace{1cm} \text{or} \hspace{1cm} \text{NC \hspace{1cm} LUMP(4),LUMP} \]

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 four 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.

\[
\begin{align*}
\text{MVC} & \hspace{1cm} \text{TEMP,Z} & \text{MVC NEW VALUE TO Temporary Location} \\
\text{NC} & \hspace{1cm} \text{TEMP,MASK} & \text{Eliminate All But Second Integer} \\
\phi C & \hspace{1cm} \text{X,MASK} & \text{Set All Bits 1 in 2D Integer Position} \\
\text{XC} & \hspace{1cm} \text{X,MASK} & \text{Now Set Them JO ZERO} \\
\phi C & \hspace{1cm} \text{X,TEMP} & \text{Insert New Value Into Word at X} \\
\text{TEMP DS} & \hspace{1cm} \text{XL4} \\
\text{MASK DC} & \hspace{1cm} \text{XL4'00780000} & \text{MASK \ Bits For Second Integer Position}
\end{align*}
\]
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 WHD, branch to FOUND.

```
LA   1,LIST      INITIALIZE TO ADDRESS OF FIRST BLOCK
LA   2,100       SET COUNT TO NUMBER OF BLOCKS
TEST  CLC  0(60,1),WHD       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
      BC   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 first 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 from 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'CBEDC'. 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 INTO NUMBER OF BYTES
    SR 1,1 SET FIRST OPERAND INDEX TO 0
    SR 2,2 FOR INDEXING INTO TABLE AT 2ND OPERAND ADDRESS
    GETARG IC 2,D1(1,B1) GET ARGUMENT BYTE, USE AS X NO EX
    IC 2,D2(2,B2) REPLACE IT BY FUNCTION BYTE FROM TABLE
    STC 2,D1(1,B1) STORE IN STRING AT FIRST OPERAND LOCATION
    BCT 0,GETARG INCREMENT FIRST OPERAND INDEX BY 1
    LOOP UNTIL 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 U NCHANGED.
DC 7CC' ' BLANK THEN NON-PRINTING CHARACTERS BETWEEN
DC C'JKLNMOPQR' PRINT LETTERS AS IS
DC C'BL8' ' BLANK OUT NON-PRINTING CHARACTERS
DC C'STUVWXYZ' 
DC 6C' ' BLANKS FOR ANYTHING BETWEEN C'Z' A N D C'O'
DC C'0123456789' DIGITS PRIN T AS IS
DC 6C' ' T AIL-ENDERS ARE BLANKED TO 0
```

20-10
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 FOR SHIFTING
SLDL 0,4   SHIFT A HEX DIGIT INTO R0
STC 0,0(0,2) STORE IN STRING A 1 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
SHFT SRC 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.
<table>
<thead>
<tr>
<th>CC Setting</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All accessed function bytes were zero.</td>
</tr>
<tr>
<td>1</td>
<td>Nonzero function byte was accessed before the last argument byte was reached.</td>
</tr>
<tr>
<td>2</td>
<td>The nonzero function byte accessed corresponds to the last argument byte.</td>
</tr>
</tbody>
</table>

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 NOPUNCT. First, we will write a program segment using CLI instructions.

```assembly
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 TO 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

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 FOR 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 TRT instruction allows us to do the same processing much more rapidly but at a cost of more memory space.

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 \text{NOPUNCT} 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 LONNAME if no punctuation is found within \(256\) characters of the start of a name.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 393</td>
<td>R3 contains index for LIST</td>
</tr>
<tr>
<td>LR 2,3</td>
<td>Clear function byte in R2</td>
</tr>
<tr>
<td>LA 1,NAMES</td>
<td>Initialize scan address</td>
</tr>
<tr>
<td>LR 4,1</td>
<td>Save initial character address in R4</td>
</tr>
<tr>
<td>TRT 0(256,1),TRTB</td>
<td>SCAN FOR PERIOD OR COMMA</td>
</tr>
<tr>
<td>BC 8,LONGNAME</td>
<td>Branch if something funny happened</td>
</tr>
<tr>
<td>ST 4,LIST(3)</td>
<td>Store address of name in LIST</td>
</tr>
<tr>
<td>SR 1,4</td>
<td>Compute name length</td>
</tr>
<tr>
<td>STC 1,LIST(3)</td>
<td>Stroke length of name in first byte</td>
</tr>
<tr>
<td>LA 3,4(0,3)</td>
<td>Increment list address</td>
</tr>
<tr>
<td>LA 1,1(4,1)</td>
<td>Move address to start of next name</td>
</tr>
<tr>
<td>BCT 2,SCAN</td>
<td>Branch if a comma was encountered</td>
</tr>
<tr>
<td>SRL 32</td>
<td>IF PERIOD, no branch. Compute and store</td>
</tr>
<tr>
<td>ST 3,NBRNMS</td>
<td>Number of names found</td>
</tr>
</tbody>
</table>

**TRTB**

- DC \((C',*)\times'00',\times'01'\) Function = 1 for period
- DC \((C','-C',\cdot-I)\times'00',\times'02'\) Function = 2 for comma
- DC 1255-\(C',.)\times'00' Zero otherwise

**NAMES**

- DC 'BROWN, GREEN, WUNKA, OF STRAND, JONES, SMEDLEY, DOE, APPLE'
- DC '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 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 over the list.

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 "SMITH", and shorter names will sort ahead of longer ones with the same beginning letters. If two identical 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 START, again separated by commas and terminated with 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: first, 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, since it allows the programmer to specify that the execution of another instruction should be performed. It is an RX-type instruction with mnemonic EX which works as follows:

1. The effective address is computed, and the r₁ digit of the EX instruction 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 instruction 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 r₁ digit which was saved is zero, proceed to step 5. Otherwise, the rightmost byte of R₁ is ORed into the second byte of the IR; R₁ 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 r₁ digit is zero, so 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 TO 0
'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  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 $\text{INST}$ 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 212</td>
<td>CLEAR OVERFLOW COUNTER</td>
</tr>
<tr>
<td>L 0,Q</td>
<td>GET FIRST INTEGER</td>
</tr>
<tr>
<td>A 0,Q+4</td>
<td>ADD SECOND INTEGER</td>
</tr>
<tr>
<td>BC 14,**+8</td>
<td>BRANCH IF NO OVERFLOW</td>
</tr>
<tr>
<td>LA 2,4</td>
<td>INDICATE ONE OVERFLOW</td>
</tr>
<tr>
<td>A 0,Q+8</td>
<td>ADD THIRD INTEGER</td>
</tr>
<tr>
<td>BC 14,**+8</td>
<td>BRANCH IF NO OVERFLOW</td>
</tr>
<tr>
<td>LA 2,4(0,2)</td>
<td>INDICATE ANOTHER OVERFLOW</td>
</tr>
<tr>
<td>EX 0,**+4(2)</td>
<td>EXECUTE A BRANCH INSTRUCTION</td>
</tr>
<tr>
<td>BC 15,NOERR</td>
<td>0-ERROR BRANCH</td>
</tr>
<tr>
<td>BC 15,ERR1</td>
<td>1-ERROR BRANCH</td>
</tr>
<tr>
<td>BC 15,ERR2</td>
<td>2-ERROR BRANCH</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC</td>
<td>MOVE RX INSTRUCTION TO WORK AREA</td>
</tr>
<tr>
<td>BLDLA(4),0(14)</td>
<td>CLEAR OLD R1 DIGIT POSITION</td>
</tr>
<tr>
<td>NI</td>
<td>SET R1 DIGIT TO 6</td>
</tr>
<tr>
<td>OI</td>
<td>INSERT LA_OPCODE INTO INSTRUCTION</td>
</tr>
<tr>
<td>MVC</td>
<td>EXECUTE THE CONSTRUCTED 'LOAD ADDRESS'</td>
</tr>
<tr>
<td>EX 0,BLDLA</td>
<td>R6 NOW CONTAINS THE DESIRED ADDRESS</td>
</tr>
<tr>
<td>BLDLA DS 2H</td>
<td>4 BYTES ON HALFWORD BOUNDARY</td>
</tr>
</tbody>
</table>
The above instruction sequence changes no registers (even though RO was available) and illustrate a technique that can be used when all register content 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 $R_8$ and $R_9$ respectively, as 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 proper 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.

As another example, suppose we wish to branch to YES if the rightmost byte of $R_3$ contains 00011111~.

```
EX 3,CLI          EXECUTE THE COMPARISON
BC 8,YES          BRANCH IF EQUALITY IS FOUND

CLI CLI CHECK,0   EXECUTED INSTRUCTION
CHECK D C 8'00011111' 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 c
```

(4) Store at T the sum of the contents of registers $R_0$ through $R_{10}$.

```
LA 11,10          COUNT IN R 11
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      R 2 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 $R_5$ contains an 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 $R_7$.

```
LTR 5,5  CHECK NUMBER OF BYTES TO BE MOVED
BC  12,FINIS  EXIT IF NOT GREATER THAN ZERO
LA  1,A  R1 CONTAINS 'FROM' ADDRESS
TEST c 5,=F'256'  SEE IF BYTE COUNT EXCEEDS 256
BC  4,LAST  IF NOT, DO LAST MOVE
MVC 0(256,7),0(1) MOVE 256 BYTES
LA  1,256(0,1)  INCREMENT 'FROM' ADDRESS
LA  7,256(0,7)  INCREMENT 'TO' ADDRESS
S  5,=F'256'  DECREASE BYTE COUNT BY 256
BC  7,TEST  IF NOT ZERO, TEST FOR FINISH
BC  8,FINIS  IF COUNT 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 BYTESTRING
```

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 for SS instructions, in which two Length Specification Digits are provided, will be discussed later.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>$r_1,r_2$</td>
</tr>
<tr>
<td>Rx</td>
<td>$r_1,d_2(x_2,b_2)$</td>
</tr>
<tr>
<td>Rs</td>
<td>$r_1,r_2,d_2(b_2)$</td>
</tr>
<tr>
<td>ST</td>
<td>$d_1(b_1),d_2$</td>
</tr>
<tr>
<td>SS</td>
<td>$d_1(L_1,b_1),d_2(b_2)$</td>
</tr>
<tr>
<td></td>
<td>$d_1(L_1,b_2),d_2(L_2,b_2)$</td>
</tr>
</tbody>
</table>

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.