These notes have been prepared for programmers using the System/360 Model 75 at SLAC. They give a simple introduction to the structure and use of System/360 at the machine-language level.

Any comments or suggestions regarding the material will be greatly appreciated.

John R. Ehrman
SLAC Computation Group
Bldg. J, x307
**TABLE OF CONTENTS**

I. Introduction  
   1. Introduction  
   2. Binary and Hexadecimal Numbers  

II. Some Basic Features of System/360  
   3. Structure of System/360  
   4. Instructions (I)  
   5. Addressing  

III. Number Representation and Arithmetic  
   6. Two's Complement Representation  
   7. Two's Complement Arithmetic  
   8. Binary Multiplication and Division  

IV. Assembler Language Programming  
   9. Assembler Language  
   10. Self-Defining Terms and Symbols  

   In preparation:  
   11. Instructions (II), Mnemonics and Operands  
   12. Establishing and Maintaining Addressability  

   and many other topics for which appropriate chapter and section headings have not yet been invented.
TABLES

I. Hexadecimal, Decimal, and Binary Digits 2-2
II. Multiples of Powers of 16 2-7
III. EBCDIC Character Representation 10-3
### Glossary of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Condition Code (PSW34-35)</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>Fn</td>
<td>Floating-Point Register n</td>
</tr>
<tr>
<td>IA</td>
<td>Instruction Address (PSW40-63)</td>
</tr>
<tr>
<td>LLC</td>
<td>Instruction Length Code (PSW32-33)</td>
</tr>
<tr>
<td>IR</td>
<td>Instruction Register</td>
</tr>
<tr>
<td>LC</td>
<td>Assembler Location Counter</td>
</tr>
<tr>
<td>MAR</td>
<td>Memory Address Register</td>
</tr>
<tr>
<td>MDR</td>
<td>Memory Data Register</td>
</tr>
<tr>
<td>PSW</td>
<td>Program Status Word</td>
</tr>
<tr>
<td>Rn</td>
<td>General-Purpose Register n</td>
</tr>
<tr>
<td>RR</td>
<td>Register-to-Register</td>
</tr>
<tr>
<td>RS</td>
<td>Register-to-Storage</td>
</tr>
<tr>
<td>RX</td>
<td>Register-to-Indexed-Storage</td>
</tr>
<tr>
<td>SI</td>
<td>Storage-Immediate</td>
</tr>
<tr>
<td>SS</td>
<td>Storage-to-Storage</td>
</tr>
</tbody>
</table>
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 operations, 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:

$$26_{10} = 11010_2, \quad 1_{10} = 1_2, \quad 10_{10} = 1010_2, \quad 1000_2 = 8_{10}.$$
As the decimal numbers being represented become larger, the number of binary digits required becomes larger also.

Thus,

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

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

<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:

# \[ \begin{align*}
26_{10} &= 11010_2 = 1A_{16}, & 26_{16} &= 100110_2 = 38_{10}, & 1_{10} &= 1_2 = 1_{16}, \\
10_{10} &= 1010_2 = A_{16}, & 100_2 &= 8_{10} = 8_{16}, & 10010_2 &= 64_{16} = 100100_2.
\end{align*} \]

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:

\begin{align*}
D5B_{16} &= 1101 \ 0101 \ 1011_2, & \text{(hexadecimal to binary)} \\
11101002 &= 3E8_{16} & \text{(binary to hexadecimal)}
\end{align*}

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

\[ 11_{16} = 10001_2 \text{ rather than } 0001 \ 0001_2. \]

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

In the positional notation we are accustomed to using, a string of digits \[ d_\ldots d_1 d_0 \] 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, and the remainder is \( b_0 \). That this is so can be seen from the definition of the quotient and remainder:

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

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.

   \[
   \begin{array}{cccccccc}
   9 & \frac{2}{2} & \frac{4}{2} & \frac{2}{2} & \frac{1}{2} & \frac{0}{2} \\
   & 18 & 8 & 4 & 2 & 0 \\
   \end{array}
   \]

   \( b_0 = 1 \) \quad \( b_1 = 1 \) \quad \( b_2 = 0 \) \quad \( b_3 = 0 \) \quad \( b_4 = 1 \)

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

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

   \[
   \begin{array}{cccccccc}
   62 & \frac{3}{16} & \frac{0}{16} \\
   & 992 & 48 & 0 \\
   \end{array}
   \]

   \( b_0 = 8 \) \quad \( b_1 = 14 \text{ or } E_{16} \) \quad \( b_2 = 3 \)

   Hence \( 1000_{10} = 3B3_{16} \).
3. Convert $627_{10}$ to base 9.

$$
\begin{array}{ccc}
9 & 7 & 0 \\
\hline
627 & 69 & 9 \\
63 & 6 & 0 \\
bo = 6 & b_1 = 6 & b_2 = 7 \\
\end{array}
$$

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.)

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

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:

$$766_9 = 7 \times 81 + 6 \times 9 + 6 = 567 + 54 + 6 = 627_{10}$$

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

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,

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

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

2-5
6. Convert $3E8_{16}$ to base 10. In this case it is usually simplest to use the positional notation used earlier:

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

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

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

This type of conversion is considerably 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>2,952,790,016</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.

![Diagram of computer structure](image)

Figure 3.1 Structure of a Typical Computer

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

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

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

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.

```
address
... 701 702 703 704 705 706 707 708 709 ...
\[ byte \quad byte \quad byte \quad byte \quad byte \quad byte \quad byte \quad byte \]
```

Figure 3.4 A portion of memory

Many of the machine instructions which refer to bytes "in memory" (which is an abbreviation for "in the memory unit") actually refer to a group of consecutive bytes. In such a situation the group, or "operand", is always addressed by referring to its 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.

```
8E7 8E8 8E9 8EA 8EB 8EC 8ED 8EE 8EF 8FO 8F1 8F2 8F3
```

```
| ← halfword → | ← halfword → | ← halfword → | ← halfword → | ← halfword → |
| ← fullword → | ← fullword → | ← fullword → | ← fullword → |
| ← doubleword → |
```

Figure 3.5 A portion of memory
Note that if (for example) a halfword operand (that is, a group of two bytes whose address is divisible by 2) were specified for some operation, and the address of that 16-bit operand were $8\text{EA}_{16}$, then bit 0 of the byte at $8\text{EB}_{16}$ would be considered to follow immediately after bit 7 of the byte at $8\text{EA}_{16}$. 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.

![Floating-Point Registers](image)

Figure 3.8 Floating-Point Registers

In certain circumstances the floating-point registers are used to contain operands 32 bits long, in which case they use only the left half of the register, and the rightmost 32 bits of the registers are ignored; this will be discussed in the chapter on floating-point arithmetic. As in the figure above, we will use the abbreviations $F_0$, $F_2$, $F_4$, and $F_6$ 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>
</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.

![Diagram](fetch-decode-execute)

**Figure 4.1 Instruction Cycle**

In the "Fetch" portion of the cycle, the CPU causes the instruction in memory which begins at the byte whose address is contained in the rightmost 24 bits of the PSW (the Instruction Address or IA) to be brought into the CPU and placed in an internal holding register where it may be examined. Though this internal register is not accessible to the programmer, we will from time to time make reference to it, so we will simply call it the Instruction Register, or IR for short. There is a simple way for the CPU circuits to know the length of an instruction and therefore how many bytes to bring from memory; this will be explained at the end of this section.
To complete the Fetch portion of the cycle, the CPU adds the length in bytes of the instruction now in the instruction register to the IA in the PSW, so that it will contain the address of the next instruction to be fetched when the current instruction has completed its execution. This means of course that instructions are packed tightly in memory; there are no leftover bytes between instructions.

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

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

It is during this final execution phase that the actual operation is performed. The operation may be a simple one which could, for example, cause the contents of one general register to replace the contents of another, or it may involve many intermediate steps of complicated logic or arithmetic. If no errors are detected during the execution phase (such as attempting to divide something by zero), the CPU then begins the cycle again by returning to the "fetch" portion of the cycle. It should be noted that
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.

```
<table>
<thead>
<tr>
<th>Base Register Specification</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>
```

Figure 5.1 Structure of an Addressing Syllable

The 4-bit field at the left of the addressing syllable contains a single hex digit which can take values from 0 to 15, and which specifies a general purpose register. The 12-bit field in the rest of the addressing syllable contains a number called the displacement which can take values from 0 to 4095.

To generate the address of 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:

$$\begin{array}{cccc}
\text{Op Code} & \text{4 bits} & \text{4 bits} & \text{16 bits} \\
\hline
0 & 7 & 8 & 11 12 15 16 31 \\
\end{array}$$

Figure 5.2 RX Instruction Showing Index Register Specification

Step 3) If the instruction is of type RX, and the 4-bit index register specification digit is not zero, then the rightmost 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 $1011\ 0010\ 1101\ 10101$ in binary (or $\text{E2DF}_{16}$ in hex) and suppose that the contents of general purpose register is $1100\ 0111\ 1110\ 1001\ 0000\ 1010\ 1111$ in binary (or $\text{C73E90AF}_{16}$ in hex). Then the effective address of the instruction is (giving both binary and hex):

$$
\begin{align*}
0000 \ 0000 \ 0000 \ 0010 \ 1101 \ 0101 \quad &0002\text{DF}_{16} \quad \text{displacement} \\
+ \ 0011 \ 1110 \ 1001 \ 0000 \ 1010 \ 1111 \quad &+ \ 3\text{E90AF}_{16} \quad \text{base (from R11)} \\
0011 \ 1110 \ 1001 \ 0011 \ 1000 \ 0100_{2} \quad &3\text{E938}_{16}
\end{align*}
$$
2. Suppose the addressing syllable of the same instruction is \( 0_{16}^{468} \). Then the effective address is \( 000468_{16} \), since R0 cannot be used for a base.

3. Suppose an RX-type instruction is \( 14_{16}^{307_{16}} \), and that the contents of R7 is \( 12345678_{16} \) and the contents of R10 is \( \text{FEDCBAA9}_{16} \). (Note that the base register specification digit, namely \( A_{16} \), means that R10 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 R0.) Then the effective address is

\[
\begin{align*}
\text{0000 0000 0000 0100 0110 1000} & \quad \text{000468 displacement} \\
\text{+ 0011 0100 0101 0110 0111 1000} & \quad \text{+ 345678 base (from R10)} \\
\text{0011 0100 0101 1010 1110 0000} & \quad \text{345A0} \\
\text{1101 1100 1011 1010 1001 1000} & \quad \text{+ DCBA98 index (from R7)} \\
\text{+ 0001 0001 0010 0111 0111 0000} & \quad \text{111578}_{16} \text{ effective address}
\end{align*}
\]

(The carry out the left end is ignored.)

4. Suppose an RX-type instruction is \( 14_{16}^{310071468} \) and that the contents of register 7 is as in example 3. Then the effective address is

\[
\begin{align*}
\text{0000 0000 0000 0100 0110 1000} & \quad \text{000468 displacement} \\
\text{+ 0011 0100 0101 0110 0111 1000} & \quad \text{+ 345678 base} \\
\text{0011 0100 0101 1010 1110 0000} & \quad \text{345AB0}_{16} \text{ effective address}
\end{align*}
\]

5. Suppose an RX-type instruction is \( 14_{16}^{30701468} \) and that the contents of register 7 is as in example 3. Then the effective address is

\[
\begin{align*}
\text{0000 0000 0000 0100 0110 1000} & \quad \text{000468 displacement} \\
\text{+ 0000 0000 0000 0000 0000 0000} & \quad \text{+ 000000 base} \\
\text{0000 0000 0000 0100 0110 1000} & \quad \text{000468} \\
\text{+ 0011 0100 0101 0110 0111 1000} & \quad \text{+ 345678 index} \\
\text{0011 0100 0101 1010 1110 0000} & \quad \text{345AB0}_{16} \text{ effective address}
\end{align*}
\]

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. On 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 2^4 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_{10} 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 \[58\quad 40\quad 0\quad 123\] specifies that a fullword operand is to be transmitted from memory and placed in R^4. 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 $FFFFFFF_{16}$,
- $2^{32}+1$ 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

$$X = \sum_{i=0}^{31} b_i \cdot 2^i.$$  

(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 $\Sigma b_i 2^i$ gives the value of $x$, because the leftmost bit, being zero, does not contribute to the sum. If $x$ is negative, the sum of the rightmost 31 bits is $2^{31} + x$ and the leftmost bit is always a one, so that we can combine these to obtain

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

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

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

$$X = +2^{31} b_{31} + \sum_{i=0}^{30} b_i 2^i.$$
If \( b_{31} \) is zero, the logical and two's complement representations give the same value, and \( x = X \). If \( b_{31} \) is one, then \( X = x + 2 \times 2^{31} = x + 2^{32} \).

But because we can only represent numbers less than \( 2^{32} \) in the logical representation, \( x + 2^{32} \) for positive \( x \) is the same as \( X \), with the extra bit being lost. Thus, for \( 0 \leq X \leq 2^{32}-1 \) and \( -2^{31} \leq x \leq 2^{31}-1 \), we have

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

(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 00102
   2) One's Complement: 1111 1101
   3) Add one: +1
      1111 11102

2. Find the two's complement of +75.
   1) Representation of +75: 0010 10112
   2) One's Complement: 1101 0100
   3) Add one: +1
      1101 01012

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 11102.
   1) One's Complement: 0000 0001
   2) Add one: +1
      0000 00102

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 0002.
   1) One's Complement: 1111 1111
   2) Add one: +1
      (carry one) 0000 0000
To the 8-bit accuracy chosen, the result is zero, and the carry of a 1 bit out the left-hand end is lost. Thus the negative of zero is still zero, which is a mathematically satisfying result; there is no such quantity as a negative zero, which can be the case in some other representations.

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

1) One’s Complement: 0111 1111
2) Add one: 1000 0000₂

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

Thus we see that the two unusual cases which arise during complementation are those for which all the bits except the sign bit are zero, and it is found that the complemented result is the same as the original operand. For a zero operand this is desirable, but for the negative case we have a situation in which there is no corresponding positive value available for a representable negative value. Such a situation is described by saying that we have generated an 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.
<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>32-bit Two's Complement Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>256</td>
<td>0000 0100&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>5000</td>
<td>0000 1388&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>2147483647(2&lt;sup&gt;31&lt;/sup&gt;-1)</td>
<td>7FFFFFFF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-2147483648(-2&lt;sup&gt;31&lt;/sup&gt;)</td>
<td>8000 0000&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-2147483647(-2&lt;sup&gt;31&lt;/sup&gt;+1)</td>
<td>8000 0001&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-5000</td>
<td>FFFF EC78&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-256</td>
<td>FFFF FFOO&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-2</td>
<td>FFFF FFFE&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>-1</td>
<td>FFFF FFFF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Figure 6.1 Examples of Two's Complement Representation

As was mentioned earlier, it is implicit in the representation of positive numbers that an arbitrary number of zero bits may be added onto the left end of a number without affecting its value. For example, the 8-bit and 16-bit representations of the decimal value +9 are 0000 1001<sub>2</sub> and 0000 0000 0000 1001<sub>2</sub>, respectively. Similarly, the 8-bit and 16-bit two's complement representations of -9 are 1111 0111<sub>2</sub> and 1111 1111 1111 0111<sub>2</sub>, 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>Length of Representation</th>
<th>Representation of +1</th>
<th>Representation of -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>01&lt;sub&gt;16&lt;/sub&gt;</td>
<td>FF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>16 bits</td>
<td>0001&lt;sub&gt;16&lt;/sub&gt;</td>
<td>FFFF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>32 bits</td>
<td>0000 0001&lt;sub&gt;16&lt;/sub&gt;</td>
<td>FFFF FFFF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>64 bits</td>
<td>0000 0000 0000 0001&lt;sub&gt;16&lt;/sub&gt;</td>
<td>FFFF FFFF FFFF&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Figure 6.2 Examples of Sign Extension

Sign extension will appear later in the discussion of instructions which perform shifting, and which do arithmetic with halfword operands.
7. TWO'S COMPLEMENT ARITHMETIC

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

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

```
<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<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)  
   \[
   \begin{array}{c}
   0000 0010
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   0000 0101
   \end{array}
   \]
   +1111 1101
   \[
   \begin{array}{c}
   0000 0010
   \end{array}
   \]
   = 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)  
   \[
   \begin{array}{c}
   1111 1110
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   1111 1110
   \end{array}
   \]
   +1111 1011
   \[
   \begin{array}{c}
   1111 1110
   \end{array}
   \]
   = -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)  
   \[
   \begin{array}{c}
   0010 1010
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   0010 1010
   \end{array}
   \]
   +0001 0001
   \[
   \begin{array}{c}
   0010 1010
   \end{array}
   \]
   = 4_{210}

4. (-17)-25:  
   \[
   \begin{array}{c}
   1110 1111 \\
   -0001 1001
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   1110 1111 \\
   +1110 0111
   \end{array}
   \]
   (carry lost)  
   \[
   \begin{array}{c}
   1110 0110
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   1110 0110
   \end{array}
   \]
   +1110 0111
   \[
   \begin{array}{c}
   1110 0110
   \end{array}
   \]
   = -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)  
   \[
   \begin{array}{c}
   0000 1000
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   0000 1000
   \end{array}
   \]
   +0001 1001
   \[
   \begin{array}{c}
   0000 1000
   \end{array}
   \]
   = 8_{10}

6. 67-(-93):  
   \[
   \begin{array}{c}
   0100 0011 \\
   -1010 0011
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   0100 0011 \\
   +1010 1101
   \end{array}
   \]
   (no carry)  
   \[
   \begin{array}{c}
   1010 1000
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   1010 1000
   \end{array}
   \]
   +0101 1101
   \[
   \begin{array}{c}
   1010 1000
   \end{array}
   \]
   = -9_{610} \ (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)  
   \[
   \begin{array}{c}
   0110 0000
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   0110 0000
   \end{array}
   \]
   +1011 1101
   \[
   \begin{array}{c}
   0110 0000
   \end{array}
   \]
   = 9_{610} \ (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)  
   \[
   \begin{array}{c}
   1101 1101
   \end{array}
   \]
   becomes  
   \[
   \begin{array}{c}
   1101 1101
   \end{array}
   \]
   +0101 1101
   \[
   \begin{array}{c}
   1101 1101
   \end{array}
   \]
   = -3_{510}
9. 3-3: 

\[
\begin{align*}
0000 \ 0011 & \quad \text{becomes} \quad 0000 \ 0011 \\
-0000 \ 00112 & \quad \text{(carry lost)} \quad +1111 \ 1101 \\
& \quad \text{expected result:} \quad 0000 \ 0000
\end{align*}
\]

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): 

\[
\begin{align*}
0000 \ 0001 & \quad \text{becomes} \quad 0000 \ 0001 \\
-1000 \ 0000 & \quad \text{(no carry)} \quad +1000 \ 0000 \\
& \quad \text{(no overflow found)} \quad 1000 \ 0001
\end{align*}
\]

11. -1-(-128): 

\[
\begin{align*}
1111 \ 1111 & \quad \text{becomes} \quad 1111 \ 1111 \\
-1000 \ 0000 & \quad \text{(carry lost)} \quad +1000 \ 0000 \\
& \quad \text{overflow indicated} \quad 0111 \ 1111
\end{align*}
\]

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 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 3-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

\[
\begin{array}{c}
\text{multiplier} & 126 \\
\text{multiplicand} & \times 213 \\
\text{partial products} & 378 \\
& 126 \\
& 252 \\
\text{product} & 26838
\end{array}
\]
broken down even more, by writing the sequence of operations in a different way.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial register contents</td>
<td>000 213</td>
</tr>
<tr>
<td>Add multiplier to upper end</td>
<td></td>
</tr>
<tr>
<td>That's 1 time</td>
<td>+126</td>
</tr>
<tr>
<td>Add multiplier</td>
<td>126 212</td>
</tr>
<tr>
<td>That's 2 times</td>
<td>+126</td>
</tr>
<tr>
<td>Add multiplier</td>
<td>252 211</td>
</tr>
<tr>
<td>That's 3 times</td>
<td>+126</td>
</tr>
<tr>
<td>Shift right 1 place</td>
<td>378 210</td>
</tr>
<tr>
<td>Add multiplier</td>
<td></td>
</tr>
<tr>
<td>That's 1 time</td>
<td>163 820</td>
</tr>
<tr>
<td>Shift right 1 place</td>
<td>016 382</td>
</tr>
<tr>
<td>Add multiplier</td>
<td>+126</td>
</tr>
<tr>
<td>That's 1 time</td>
<td>142 381</td>
</tr>
<tr>
<td>Add multiplier</td>
<td>+126</td>
</tr>
<tr>
<td>That's 2 times</td>
<td>268 380</td>
</tr>
<tr>
<td>Shift right 1 place</td>
<td>026 838</td>
</tr>
</tbody>
</table>

We place the multiplicand in the right half of the double-length register and clear the left half to zero. Then by examining the rightmost digit of the multiplicand we know how many times to add the multiplier to the left half of the double-length register. When the rightmost digit has been counted down to zero, the partial product of that digit and the multiplier has been added to the accumulating result. Then the entire double-length register is shifted to the right one digit position, at which time the zero digit at the right-hand end is lost and a zero digit is inserted in the vacated position at the left. The process of adding the multiplier and counting down on the multiplicand digit then continues until the proper partial product has been added to the accumulated result. This process is repeated for as many steps as there are multiplicand digits. When completed, the result in the double-length register is the product, and all the multiplicand digits have been shifted off the right-hand end. The main
points to observe are that (1) the multiplicand is placed in the right half of the double-length register, (2) the left half is initially cleared to zero, (3) the multiplier is added to the left end depending on the multiplicand digit at the far right, and (4) the decimal point of the result (that is, the position of the least significant digit) is at the right-hand 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 \((126 \times 9 = 1134)\) 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 5-digit binary numbers and registers and wish to multiply 00110₂ by 01001₂ to obtain a 10-bit product in a double-length register. Then the sequence of steps shown below indicates the method.

| Initialize | 00110 | multiplier (in separate register) |
| Shift right 1 | 00000 0100₁ | multiplicand in right half of double-length register |
| Step 1: rightmost bit = 1, add multiplier | 00110 0100₁ |
| Step 2: rightmost bit = 0, no add. Shift right 1 | 00001 0010₀ (1 bit lost) |
| Step 3: rightmost bit = 0, no add. Shift right 1 | 00000 1100₁ |
| Step 4: rightmost bit = 1, add multiplier | 00110 1100₁ |
| Shift right 1 | 00011 0110₀ (1 bit lost) |
| Step 5: rightmost bit = 0, no add. Shift right 1 | 00001 10110 final product = 110110₂ = 5₄₁₀ |

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 be added 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 ≥ divisor, subtract, and count up at right end.
Dividend ≥ divisor; subtract again
Dividend < divisor; no subtraction
Shift dividend left again
Dividend ≥ divisor; subtract and count up
Dividend < divisor; no subtraction
Shift left for last time
Dividend ≥ divisor; subtract
Subtract and count up by 1
Dividend ≥ divisor; subtract
Subtract and count
Dividend ≥ divisor; subtract
Dividend now < divisor; stop

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

As was the case for multiplication, binary division is simplified by the fact that at most one subtraction need be made for each quotient digit generated. To illustrate, consider this example using a five-bit divisor and a ten-bit dividend. Let the dividend be 0000111011₂ = 59₁₀, and let the divisor be 00110₂. 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.)

Thus the remainder $0010_2 = 5_{10}$ in the left half, and the quotient $0100_2 = 9_{10}$ in the right half are as expected.

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

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

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

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

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

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

![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.

![Figure 9.2 Simplified Schematic of Linkage-Editor Processing](image)

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.

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

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

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

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

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

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

**Figure 9.6 An Assembler Instruction Statement**

**Figure 9.7 A Macro-Instruction Statement**
10. SELF-DEFINING TERMS AND SYMBOLS

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

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

A decimal self-defining term is simply an unsigned string of decimal digits. 12345, 98, and 007 are examples of decimal self-defining terms. The size of a decimal self-defining term is limited by the fact that 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 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 less than 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'111111100001100' are examples of binary self-defining terms. Because 2^4 bits are allotted for the value of self-defining terms, at most 24 digits may be specified between the apostrophes. Note also that the value of the term is assumed positive even though the leftmost position contains a one bit.

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

<table>
<thead>
<tr>
<th>Character</th>
<th>Representation</th>
<th>Character</th>
<th>Representation</th>
<th>Character</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>40</td>
<td>C</td>
<td>C3</td>
<td>T</td>
<td>E3</td>
</tr>
<tr>
<td>.</td>
<td>4B</td>
<td>D</td>
<td>C4</td>
<td>U</td>
<td>E4</td>
</tr>
<tr>
<td>(</td>
<td>4D</td>
<td>E</td>
<td>C5</td>
<td>V</td>
<td>E5</td>
</tr>
<tr>
<td>+</td>
<td>4E</td>
<td>F</td>
<td>C6</td>
<td>W</td>
<td>E6</td>
</tr>
<tr>
<td>&amp;</td>
<td>50</td>
<td>G</td>
<td>C7</td>
<td>X</td>
<td>E7</td>
</tr>
<tr>
<td>$</td>
<td>5B</td>
<td>H</td>
<td>C8</td>
<td>Y</td>
<td>E8</td>
</tr>
<tr>
<td>*</td>
<td>5C</td>
<td>I</td>
<td>C9</td>
<td>Z</td>
<td>E9</td>
</tr>
<tr>
<td>)</td>
<td>5D</td>
<td>J</td>
<td>D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>60</td>
<td>K</td>
<td>D2</td>
<td>0 (digit)</td>
<td>F0</td>
</tr>
<tr>
<td>/</td>
<td>61</td>
<td>L</td>
<td>D3</td>
<td>1</td>
<td>F1</td>
</tr>
<tr>
<td>,</td>
<td>6B</td>
<td>M</td>
<td>D4</td>
<td>2</td>
<td>F2</td>
</tr>
<tr>
<td>#</td>
<td>7B</td>
<td>N</td>
<td>D5</td>
<td>3</td>
<td>F3</td>
</tr>
<tr>
<td>@</td>
<td>7C</td>
<td>$ (letter)</td>
<td>D6</td>
<td>4</td>
<td>F4</td>
</tr>
<tr>
<td>:</td>
<td>7D</td>
<td>P</td>
<td>D7</td>
<td>5</td>
<td>F5</td>
</tr>
<tr>
<td>=</td>
<td>7E</td>
<td>Q</td>
<td>D8</td>
<td>6</td>
<td>F6</td>
</tr>
<tr>
<td>A</td>
<td>C1</td>
<td>R</td>
<td>D9</td>
<td>7</td>
<td>F7</td>
</tr>
<tr>
<td>B</td>
<td>C2</td>
<td>S</td>
<td>E2</td>
<td>8</td>
<td>F8</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Relocatability</th>
<th>Length</th>
<th>Type</th>
<th>Scaling</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AGENT007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#235</td>
<td>O0H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAMES</td>
<td>KSPφ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$746295</td>
<td>WÑK'A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- $7462.95 (decimal point not allowed)
- BÑND/007 (no division sign allowed)
- SET Gφ (no blanks allowed)
- 235# (does not start with a letter)
- CHARACTER (too many characters)
- TEN*FVE (contains the special character *)
- C'WÑK'A (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 LAD 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    F'10'
```

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

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

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

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

When a program is ready to be assembled, one of the first steps the Assembler must perform is the assignment of a 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 \(7B6_{16}\) when the statement given in the first example above was scanned. Then the value of the symbol GETC\(\text{INST}\) would be \(7B6_{16}\), and because the instruction whose mnemonic is L is an RX-type instruction of length 4 bytes, the LC is incremented by four and will be \(7BA_{16}\) when the scan of the following statement is begun. In this way the Assembler scans all the statements of the program and assigns values to all symbols appearing as name field entries. It should be noted that there are other methods for assigning values to symbols, but the method described is what will most often be used, and that there are also assembler instruction statements which allow the programmer to change the value of the Location Counter. This usual method of symbol definition provides the simplest definition of a relocatable symbol: suppose the relative origin is changed by some fixed amount; if the value of the symbol changes by the same amount, then that symbol is relocatable. We will see later that it is also possible to define symbols whose values either do not change or which change in different ways. (The reader should also note that there is a definite difference between the LC, which is maintained by the Assembler program in the course of processing the statements of the source program, and the Instruction Address in the PSW, which gives the location in memory of the next instruction to be executed during the execution step of the program. They are not at all the same.)

After this brief discussion of how symbols get their values, we turn to the question of their utility. Suppose we want to write an instruction which will load the integer constant ten into R0 (remember that this is an abbreviation for general register 0). Suppose also that we also know that
some other general register will contain an address which will provide addressability for the fullword area of memory containing the constant. Then we could calculate what the exact displacement would have to be and write the instruction with the base and displacement given explicitly. If, for example, these were 6 and $4EC_{16}$ respectively, we could write (the details of writing the operand field will be discussed in the next section)

\[
L 0,X'4EC'(0,6)
\]

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

\[
L 0,TEN
\]

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

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

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

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

RR instructions are always two bytes long.

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

SS instructions are always six bytes long.

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

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

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Instruction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxxxxx</td>
<td>RR</td>
</tr>
<tr>
<td>01xxxxxx</td>
<td>RX</td>
</tr>
<tr>
<td>10xxxxxx</td>
<td>RS,SI</td>
</tr>
<tr>
<td>11xxxxxx</td>
<td>SS</td>
</tr>
</tbody>
</table>

Figure 4.3 Bit Patterns for Each Instruction Type

Thus if the first two bits are 00 the instruction is two bytes long; if the bits are 01 or 10 the instruction is four bytes long; and if the bits are 11 the instruction is six bytes long. Before proceeding with the decoding phase of the instruction cycle, the CPU places the number of 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 ILC) what instruction caused the error. (This is of course precisely what is done; we will note for now that if the ILC were not saved, it would not be possible to determine the exact location of the offending instruction, since the location of the next instruction to be executed is what appears in the PSW and the length of the bad instruction is variable. This is a subject with many ramifications, to be covered later.)